Towards a privacy-preserving web

Iqbal, Umar

TOWARDS A PRIVACY-PRESERVING WEB

by

Umar Iqbal

A thesis submitted in partial fulfillment of the
requirements for the Doctor of Philosophy
degree in Computer Science
in the Graduate College of
The University of Iowa

August 2021

ACKNOWLEDGEMENTS

I have been incredibly fortunate to have helpful and supportive mentors, collaborators, friends, and family. This thesis would not have been possible without their guidance and support.

First and foremost, I would like to thank my advisor, Zubair Shafiq, for his invaluable mentorship. I am especially thankful to him for his openness to ideas, encouragement to go out of my comfort zone, teaching me to critically question my research, and to pick important research problems. I also enjoyed our informal discussions, especially while playing cricket and walking around cities during conference travels. Reflecting back, it has been a very rewarding experience.

I would also like to thank several folks at the University of Iowa, who have made my Ph.D. an incredible experience. Especially my comprehensive, proposal, and thesis defense committee members, Omar Chowdhury, Rishab Nithyanand, Juan Pablo, and Supreeth Shastri for their valuable feedback, that helped me improve my research. I am particularly thankful to Omar Chowdhury for providing immense support and writing tens of letters. I cannot forget my weekly meetings with my lab members and friends, Shehroze Farooqi, Adnan Ahmad, Hammad Mazhar, John Cook, and Huyen Le. They provided invaluable critical feedback and a friendly environment for me to learn and excel. Last, but not the least, I want to thank Sheryl Semler and Catherine Till for providing swift administrative support and tolerating my last minute requests.

I would also like to thank my undergraduate research advisor, Fareed Zaffar, for introducing me to research, providing me an opportunity to work with talented individuals, and encouraging me to pursue a Ph.D. I would not have been where I am today, if it was not for the encouragement and advise of my middle school principal, Khalida Parveen.

I have been fortunate to collaborate and learn from many talented people, both in academia and in industry. I would especially like to thank, Steven Englehardt, Zhiyun Qian, Carmela Troncoso, Shitong Zhu, Sandra Siby, John Hazen, and John Wilander.

I am thankful to have friends who were always there for me through the ups and downs of graduate school. A special thank you to Bilal Khan, Bilal Bhatti, Hamza Naveed, Bilal Asif, Usman Rana, and Ankur Parupally.

I owe everything to my parents, Iqbal and Farhana, who provided their utmost support in all my endeavors. I thank them for their unwavering love and guidance. I want to thank my wife, Jeehan, for supporting me through thick and thin. Especially for tolerating my busy work schedule and helping me get better at maintaining my work life balance :)

I dedicate this thesis to my parents and my wife.

ABSTRACT

Modern web applications are built by combining functionality from external third parties; with the caveat that the website developers trust them. However, third parties come from various sources and website developers are often unaware of their origin and their complete functionality. Thus, the presence of "trusted" third parties has lead to many security and privacy abuses on the web, with one of the most severe consequence being privacy-invasive cross-site tracking without the knowledge or consent of users.

In this thesis, I aim to tackle the cross-site tracking menace to make the web more secure and private. Specifically, I build novel privacy-enhancing systems using system instrumentation, machine learning, program analysis, and internet measurements techniques. At a high level, my research process involves: instrumenting web browsers to capture detailed execution of webpages, conducting rigorous measurements of privacy and security abuses, and using insights from the measurement studies to build machine learning based approaches that counter the privacy and security abuses.

In the first half of this thesis, I build privacy-enhancing systems that counter third party cross-site tracking by blocking both stateful tracking, commonly referred to as cookie based tracking, and stateless tracking, commonly referred to as browser fingerprinting. In the second half of this thesis, I build privacy-enhancing systems that counter retaliation by third party circumvention services that evade blocking, by either detecting and removing them or by stealthily concealing the traces of blocking. At the end of the thesis, I highlight the research contributions made by this thesis and some of the open research problems.

PUBLIC ABSTRACT

Modern web applications are built by combining functionality from external third parties; with the caveat that the website developers trust them. However, third parties come from various sources and website developers are often unaware of their origin and their complete functionality. Thus, the presence of “trusted” third parties has lead to many security and privacy abuses on the web, with one of the most severe consequence being privacy-invasive cross-site tracking without the knowledge or consent of users.

In this thesis, I aim to tackle the cross-site tracking menace to make the web more secure and private. Specifically, I build novel privacy-enhancing systems using system instrumentation, machine learning, program analysis, and internet measurements techniques. At a high level, my research process involves: instrumenting web browsers to capture detailed execution of webpages, conducting rigorous measurements of privacy and security abuses, and using insights from the measurement studies to build machine learning based approaches that counter the privacy and security abuses.

In the first half of this thesis, I build privacy-enhancing systems that counter third party cross-site tracking by blocking both stateful tracking, commonly referred to as cookie based tracking, and stateless tracking, commonly referred to as browser fingerprinting. In the second half of this thesis, I build privacy-enhancing systems that counter retaliation by third party circumvention services that evade blocking, by either detecting and removing them or by stealthily concealing the traces of blocking. At the end of the thesis, I highlight the research contributions made by this thesis and some of the open research problems.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

The web's modular nature is responsible for its tremendous success as well as its chronic insecurity. The modular nature of building web applications allows publishers to add new functionality to their websites by embedding third parties as needed. However, the modularity, by its very nature, also requires that publishers implicitly trust the embedded third parties. The browsers impose only modest restrictions on third party resource embedding, leaving users susceptible to be exploited by malicious third parties that often blatantly bypass these restrictions. One of the most important consequence of this design is privacy-invasive, cross-site tracking by third parties without knowledge or consent of users. In this thesis, I aim to build privacy-enhancing systems that can be deployed in web browsers to rein in cross-site tracking.

1.2 Research Challenges

It is challenging to detect and mitigate cross-site tracking on the web. First, each website is unique from the other and embeds different third parties, where third parties may further include other third party resources on the fly. Such dynamism requires privacy-enhancing tools to be **scalable** and adapt to changing websites. Second, given the adversarial nature of the problem, malicious third parties often use evasive tactics to circumvent restriction. Privacy-enhancing tools need to be **robust** to adversarial evasion by malicious third parties. Third, countermeasures imposed by privacy-enhancing tools are susceptible to breaking legitimate functionality and hinder user experience. Thus, to be suitable for real-world deployment, privacy-enhancing countermeasures should not degrade **usability**.

1.3 Research Approach

To address the **scalability** issues, I leverage machine learning (ML) techniques to detect *tracking* behaviors instead of *tracker* entities. To effectively apply ML, I gather labeled data, estimate the cost of incorrect predictions, and fight adversarial actors. To address the **robustness** issues, I capture execution of trackers across all three layers (HTML, JavaScript, and network) of the web stack. Capturing the cross-layer context allows me to trace the provenance of trackers and reveal any evasion attempts. To address the **usability** issues, I rely on user feedback and associate user perceived breakage to the execution patterns of the trackers and encode it into ML models that automatically reduce the breakage events.

1.4 Research Contributions

The research problems solved in this thesis can be divided into two thrusts: (1) Content blocking and (2) Anti circumvention. Content blocking aims to counter third party cross-site tracking by blocking advertisers and trackers. Anti circumvention aims to counter retaliation by intrusive third parties, that detect content blockers, by either detecting and removing them or by stealthily concealing the traces of content blocking. At a high level, my research approach starts off with instrumenting web browsers to meticulously study cross-site tracking and evasion techniques and building ML-based content blocking and anti circumvention privacy-enhancing tools. In the following, I will briefly describe these two thrusts in detail.

1.4.1 Content Blocking

Trackers rely on stateful and stateless techniques to conduct cross-site tracking. In stateful tracking, trackers store unique identifiers in the browsers, usually in the form of cookies. Whereas in stateless tracking, unique identifiers are computed at run time based on the values exposed by JavaScript APIs. Content blocking has become the most popular approach to counter cross-site stateful and stateless tracking. In my research, I aim to improve content blocking by tackling both stateful and stateless tracking.

1.4.1.1 Stateful tracking (AdGraph [216], WebGraph [288], Khaleesi [217])

The first sub-thrust of my research in content blocking tackles stateful tracking. Content blockers rely on filter lists to block advertisers and trackers. Filter lists are manually curated with a philosophy of blocking advertising and tracking “endpoints”. Due to their manual nature and endpoint blocking approach, filter lists suffer from scalability, robustness, and usability issues.

To bridge this gap, I proposed AdGraph, WebGraph, and Khaleesi, which are graph and sequence-based ML approaches to detect advertising and tracking. Intuitively, AdGraph, WebGraph, and Khaleesi detect advertising and tracking resources because their execution patterns differ from functional resources. AdGraph is designed as a general purpose content blocking tool, whereas WebGraph is designed for robust content blocking, and Khaleesi is designed to detect emerging tracking threats. Below, I give a brief overview of these approaches.

1. *Generic Content Blocking.* AdGraph models the execution of a webpage across HTML, network, and JavaScript layers as a graph and featurizes the cross-layer resource interaction and resource content to train an ML classifier that detects advertising and tracking.

2. *Robust Content Blocking.* WebGraph extends AdGraph, by incorporating storage layer in the graph representation and by eliminating the reliance on resource content. Extending AdGraph, allows WebGraph to be robust against adversarial evasions that manipulate resource content to avoid detection.

3. *Targeted Content Blocking.* Khaleesi models the network requests, across network and JavaScript layer, in a sequential structure and capitalizes on the sequential context to train an ML model that detects request chains based emerging tracking threats (e.g., bounce tracking [312]).

All, AdGraph, WebGraph, and Khaleesi suffer from the lack of reliable ground truth to label ads and trackers at scale, as it is not readily available, which leaves these approaches to train on unreliable ground truth. However, these approaches generalizes well on advertising and tracking behavior and are able to identify mistakes in the ground truth. Further, due to fewer mistakes in detection, they causes less website breakage in comparison to the state-of-the-art content blockers.

Overall AdGraph, WebGraph, and Khaleesi improve the state-of-the-art by: (1) proposing ML-based approaches to detect ads and trackers at scale, (2) using cross-layer context to make detection robust against single layer evasion attacks, and (3) generalizing on advertising and tracking to cause least disruptions on usability.

1.4.1.2 Stateless tracking (FP-Inspector [214])

The second sub-thrust of my research in content blocking tackles stateless tracking. As mainstream browsers have implemented countermeasures against stateful tracking, there are concerns that trackers will migrate to more opaque stateless tracking techniques, such as browser fingerprinting. Browser fingerprinting is largely unexplored and a little is known about it. My research aims to provide an informed perspective about browser fingerprinting by rigorously studying it. Specifically, I proposed FP-Inspector– a syntactic semantic ML based approach to detect and counter browser fingerprinting.

Intuitively, FP-Inspector detects browser fingerprinting because fingerprinting scripts have abnormal usage of JavaScript APIs. FP-Inspector combines static analysis, to featurize JavaScript API keywords, and dynamic analysis, to featurize JavaScript execution, to train an ML classifier that detects browser fingerprinting. Static analysis complements dynamic analysis, in case scripts are dormant, and dynamic analysis complements static analysis, in case scripts are obfuscated; overall, making the classifier robust. Browser fingerprinting is an under-studied problem and there is no consensus on a single definition, which makes it extremely hard to attain a reliable ground truth at scale. FP-Inspector, uses manually curated, high precision and low recall, heuristics to "build" its own ground truth by iteratively training a classifier. Further, FP-Inspector evaluates the usability of several fingerprinting countermeasures and discovers that JavaScript API restriction is the least disruptive mitigation to browser fingerprinting. Moreover, deploying FP-Inspector in the wild uncovered a recent surge in use of browser fingerprinting for cross-site tracking in new ways. Overall, FP-Inspector improves the state-of-the-art by: (1) proposing an ML based approach to detects browser fingerprinting at scale, (2) using both static and dynamic analysis for robust detection, (3) evaluating the usability of countermeasures, and (4) measuring the prevalence of browser fingerprinting.

1.4.2 Anti Circumvention

Content blocking curtails the advertisers' ability to do cross-site tracking and serve targeted advertisements. As a consequence, advertisers and publishers retaliate against the users of content blocking tools by using "anti-adblockers" that force users to disable their content blockers. Anti-adblockers are JavaScript snippets that leverage the content blockers' interaction with bait ad/tracking requests or HTML elements to detect their users and deploy warning messages or paywalls. My research aims to counter circumvention by anti-adblockers by detecting and removing them or by stealthily concealing the actions of content blockers to fool them.

1.4.2.1 Active blocking (AdWars [215])

Active content blocking counters circumvention by anti-adblockers by detecting and blocking anti-adblocking scripts. They key idea is to detect and block anti-adblockers before they get a chance to detect content blockers.

I proposed an ML approach that featurizes anti-adblocking scripts to detect anti-adblocking. The features capture the concentrated usage of web requests, DOM interaction, and timing specific JavaScript APIs – which are disproportionately used by anti-adblockers. Detecting anti-adblockers is challenging because they stay dormant before getting triggered by content blockers.We sidestep the problem of activating anti-adblockers and instead use static analysis to featurize scripts. Further, anti-adblocking detection suffers from the lack of ground truth due to their limited and adversarial usage, making it challenging to train a robust classifier. We address this issue by training an ensemble classifier that aims to create a strong meta-classifier by iteratively training multiple weak classifiers, where each subsequent model attempts to correct the errors from the previous model. Overall, my approach improves the state-of-the-art by: (1) proposing a static analysis based ML based approach that detects anti-adblocking at scale and by (2) using a classifier that incrementally improves itself to be more robust.

1.4.2.2 Stealthy blocking (ShadowBlock [286])

Active content blocking can lead to an arms-race, where publishers and websites attempt to detect blocking of anti-adblockers and content blockers attempt to block the detection of blocking of anti-adblockers. To avoid that arms-race, I proposed an orthogonal approach to counter circumvention through stealthy content blocking, that completely sidesteps the problem of detecting anti-adblockers.

Specifically, I proposed a stealthy content blocker, called ShadowBlock, which instruments the Chromium web browser to conceal the traces of content blocking from anti-adblockers. ShadowBlock has a two step process to stealthy content blocking: (1) it first uses filter lists to remove ads from the webpage and (2) then it creates a shadow copy of ad DOM elements in the browser to fake their presence to anti-adblockers. Browser instrumentation allows ShadowBlock to remove the static and as well as the dynamic ad elements by tracing the execution of scripts, making stealthy content blocking effective. Further, browser instrumentation also allows ShadowBlock to hook the JavaScript APIs, that can be probed to check the presence of ad elements, at an abstraction that is invisible to anti-adblockers, making stealthy content blocking robust. Overall, ShadowBlock provides (1) a stealthy content blocker that is robust against circumvention by anti-adblockers and (2) improves the usability, by avoiding circumvention altogether which flummoxes popular content blocking extensions.

1.4.2.3 Research Impact

My research has had a direct impact on the state-of-the-art privacy-enhancing tools. My research has been incorporated by various industry partners in their privacy-focused products. Most notably, Brave is integrating AdGraph [216] in their production browser [18]. Popular filter lists such as DuckDuckGo (used by Safari), Disconnect (used by Firefox and Edge), and Easylist/EasyPrivacy (used by uBlock Origin and many other anti-tracking tools) have incorporated trackers detected by FP-Inspector [211]. Firefox is considering to reduce the fingerprint-ability of several JavaScript APIs that were reported [212] to be abused by FP-Inspector [214].

I have also contributed to open-source projects such as WebKit's Intelligent Tracking Protection (ITP) [213]. My research can also inform standards bodies, such as the World Wide web Consortium (W3C), in evaluating recent high-profile proposals, such as Privacy Budget, which is part of Google's Privacy Sandbox project [166]. Overall, my research has directly benefited hundreds of millions of web users.

My research has been published in top-tier academic conferences such as IEEE S&P *(Oakland)* [214,216], USENIX Security [288], ACM SIGCOMM *IMC* [215], and IW3C2/W3C's The Web Conference *(WWW)* [286]. My research has also garnered significant interest from the press including coverage in *CNET* [169], *ZDNet* [320], *VentureBeat* [306], *Forbes* [191], etc.

1.4.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 discusses mitigation of generic stateful tracking. Chapter 3 discusses mitigation of adversarial stateful tracking. Chapter 4 discusses mitigation of emerging stateful tracking threats. Chapter 5 discusses mitigation of stateless tracking. Chapter 6 discusses countering circumvention by anti-adblockers by detecting and removing them. Chapter 7 discusses countering circumvention by anti-adblockers by stealthily concealing the actions of content blockers. Chapter 8 discusses some of the open research problems in web privacy space. Chapter 9 concludes this thesis.

CHAPTER 2

AdGraph: A GRAPH-BASED APPROACH TO AD AND TRACKER BLOCKING

2.1 Introduction

The need for content blocking on the web is large and growing. Prior research has shown that blocking advertising and tracking resources improves performance [80, 197, 274], privacy [141, 186, 242], and security [198, 255], in addition to making the browsing experience more pleasant [74]. Browser vendors are increasingly integrating content blocking into their browsers [180, 277, 310], and user demand for content blocking is expected to grow in future [105, 106].

While existing content blocking tools are useful, they are vulnerable to practical, realistic countermeasures. Current techniques generally block unwanted content based on URL patterns (using manually-curated filter lists which contain rules that describe suspect URLs), or patterns in JavaScript behavior or code structure. Such approaches fail against adversaries who rotate domains quickly [168], proxy resources through trusted domains (e.g. the first party, CDNs) [61], or restructure or obfuscate JavaScript [237], among other common techniques.

As a result, researchers have proposed several alternative approaches to content blocking. While these approaches are interesting, they are either incomplete or susceptible to trivial circumvention from even mildly determined attackers. Existing proposals suggest filter lists, pre-defined heuristics, and machine learning (ML) approaches that leverage network or code analysis for identifying unwanted web content, but fail to consider enough context to avoid trivial evasions.

This work presents AdGraph, an accurate and performant graph-based ML approach for detecting and blocking unwanted (advertising and tracking) resources on the web. AdGraph makes blocking decisions using a novel graph representation of a webpage's past and present HTML structure, the behavior and interrelationships of all executed JavaScript code units, and the destination and cause of all network requests that have occurred up until the considered network request. This contextually-rich blocking approach allows AdGraph to both identify unwanted resources that existing approaches miss, and makes AdGraph more robust against simple evasions that flummox existing approaches.

AdGraph is designed for both online (i.e. in-browser, during page execution) and offline (i.e. for filter list construction) deployment. AdGraph is performant enough for online deployment; its performance is comparable to stock Chromium and better than Adblock Plus. AdGraph can also be used offline to create or augment filter lists used by extension-based content blocking approaches. This dual deployment strategy can benefit users of AdGraph directly as well as users of extension-based content blocking approaches.

This work makes the following contributions to the problem of identifying and blocking advertising and tracking resources on the web.

1. A **graph-based ML approach** to identify advertising and tracking resources in websites based on the HTML structure, JavaScript behavior, and network

requests made during execution.

2. A **large scale evaluation** of AdGraph's ability to detect advertising and tracking resources on popular websites. We find that AdGraph is able to replicate the labels of human-generated filter lists with 95.33% accuracy. Further, AdGraph is able to outperform existing filter lists in many cases, by correctly distinguishing ad/tracker resources from benign resources in cases where existing filter lists err.

3. A **performant implementation** of AdGraph as a patch to Chromium.[1] Our approach modifies the Blink and V8 components in Chromium to instrument and attribute document behavior in a way that exceeds existing practical approaches, without significantly affecting browser performance. AdGraph loads pages faster than stock Chromium on 42% of pages, and faster than AdBlock Plus on 78% of pages.

4. A **breakage analysis** of AdGraph's impact on popular websites. AdGraph has a noticeable negative affect on benign page functionality at rates similar to filter lists (affecting 15.0% versus 11.4% of websites respectively) and majorly affects page functionality less than filter lists (breaking 5.9% versus 6.4% websites, respectively).

Chapter organization: The rest of this chapter is structured as follows. Sec-

[1]Since AdGraph is designed and implemented in Chromium, it can be readily deployed on other Chromium based browsers (e.g. Chrome, Brave).

tion 2.2 presents existing work on the problem of ad and tracker blocking, and discusses why existing approaches are insufficient as comprehensive blocking solutions. Section 2.3 describes the design and implementation of AdGraph. Section 2.4 presents an evaluation of AdGraph's effectiveness as a content blocking solution, in terms of blocking accuracy, performance, and effect on existing websites. Section 2.5 describes AdGraph's limitations, how AdGraph can be further improved, and potential uses for AdGraph in offline scenarios. Section 2.6 concludes the chapter.

2.2 Background and Related Work

2.2.1 Problem Difficulty

Ad and tracker blocking is a well studied topic (e.g. [147,149,203,209,225,287, 316,318]). However, existing work is insufficient to form a comprehensive and robust blocking solution.

Many existing approaches (e.g. [149,203]) are vulnerable to commonly deployed countermeasures, such as evading domain-based blocking through domain generation algorithms (DGA) [168], hosting tracking related code on the first-party domain [61], spreading tracking related behavior across multiple code units, and code obfuscation [237]. Much related work in the area is unable to reason about domains that host both "malicious" (ads and tracking) and "benign" (functional or user desireable) content, and end up over or under labeling resources.

Other existing work (e.g. [147,225]) lacks realistic evaluations. Sometimes this takes the form of an ambiguous comparison to ground truth (making it challenging to ascertain the usefulness of the technique as a deployable solution). Other cases target advertising or tracking, but not both together. Still other cases target only a subset of advertising or tracking related resources (e.g. scripts or images), but fail to consider other ways advertising or tracking can be carried out (e.g. iframes and CSS styling rules).

Further existing work (e.g. [209,316]) presents a strategy for blocking resources, but lacks an evaluation of how much benign (i.e. user desirable) functionality the approach would break. This leaves a proposal for preventing a subset of an application's code from executing, without an understanding of how it effects the functioning of the overall application (user-serving or otherwise). These approaches may fail to separate the wheat from the chaff; they may prevent advertising and tracking, but at the expense of breaking desirable functionality.

The rest of this section reviews existing work on blocking advertising and tracking content on the web. Emphasis is given both on the contributions of each work, and why each work is incomplete as a deployable, real-world blocking solution.

2.2.2 Existing Blocking Techniques

This subsection describes existing tracking and advertising blocking work, categorized by the types of evasions each approach is vulnerable to. Our goal is not to lessen the contributions of existing work (which are many and significant), but merely to highlight the kinds of practical and deployed evasions each is vulnerable to, to further motivate the need for a more comprehensive solution.

Note that many blocking approaches discussed here are vulnerable to multiple

Approach	Ad/Tracker Blocking	Domain/URL Blocking	1st,3rd Party Blocking	DGA Susceptibility	Code Structure Susceptibility	Cross JS Collaboration Susceptibility	Breakage Analysis
Bau et al. [147]	Tracker	Domain	3rd party	Yes	-	-	No (-)
Yu et al. [318]	Tracker	Domain	3rd party	Yes	No	No	Yes (25%)
Wu et al. [316]	Tracker	Domain	3rd party	Yes	No	Yes	No (-)
Shuba et al. [287]	Ads	URL	1st,3rd party	Yes	No	No	No (-)
Kaizer and Gupta [225]	Tracker	Domain	3rd party	Yes	Yes	Yes	No (-)
Ikram et al. [209]	Tracker	URL	1st,3rd party	No	Yes	Yes	No (-)
Gugelmann et al. [203]	Ads,Tracker	Domain	3rd party	Yes	No	Yes	No (-)
Bhagavatula et al. [149]	Ads	URL	1st,3rd party	Yes	No	No	No (-)

Table 2.1. Comparison of the related work, including the practical evasions and countermeasures each is vulnerable to. Ad/Tracker Blocking column represents blocking of ads, trackers, or both. Domain/URL Detection column represents blocking at domain or URL level. 1st,3rd Party Blocking column, represents blocking of third-party requests, first-party requests, or both. In DGA Susceptibility, Code Structure Susceptibility, and Cross JS Collaboration Susceptibility columns, Yes and No represent that the approach's susceptibility to specified countermeasure. The Breakage Analysis column represents whether the breakage analysis was performed by the approach and their results.

evasions. In these cases, we discuss only one category of evasion the work is vulnerable to. Table 2.1 summarily compares the strengths and weaknesses of existing approaches.

Domain Based Blocking. Many existing content blocking approaches attempt to prevent advertising and tracking by identifying suspect domains (eTLD+1), and blocking all requests to resources on such domains. These approaches are insufficient for several reasons. First, determined advertising and tracking services can use DGA to serve their content from quickly changing domains that are unpredictable to the client, but known to the adversary. Such evasions trivially circumvent approaches that depend primarily, or only, on domain blocking strategies [168]. Similarly, in many cases, domain-focused approaches are easily circumvented by proxying the malicious resource through the first-party domain [61]. A comprehensive blocking solution should be able to account for both of these evasion strategies.

AdBlock Plus [5], uBlock Origin [94], Ghostery [51], and Disconnect [31] are all popular and deployed solutions that depend solely or partially on the domain of the request, and are thus vulnerable to the above discussed approaches. These approaches use filter lists, which describe hosts, paths, or both of advertising and tracking resources.

Gugelmann et al. [203] developed a ML-based approach for augmenting filter lists, by using existing filter lists as ground truth, and training a classifier based on the HTTP and domain-request behavior of additional network requests. Bhagavatula et al. [149] developed a ML-based approach for generating future domain-and-path based filter lists, using the rules in existing filter lists as ground truth. These approaches may be useful in identifying additional suspect content, but are easily circumvented by an attacker willing to take any of the domain hiding or rotating measures discussed earlier.

Yu et al. [318] described a method for detecting tracking related domains by looking for third-parties that receive similar unique tokens across a significant number of first-parties. This approach hinges on an attacker using the same receiving domain over a large number of hosting domains. Apple's Safari browser includes a similar technique called Intelligent Tracking Protection [310], that identifies tracking related domains by looking for third-party contexts that access state without user interaction. Privacy Badger [76] also identifies tracking related domains by looking for third-party domains that track users (e.g., by setting identifying cookies) on three or more sites. These techniques do not attempt to block advertising, and also require that the attacker use consistent domains. Bau et al. [147] proposed building a graph of resource-hosting domains and training a ML classifier based on commonalities of third-party hosted code, again relying on hosting domains being distinct, consistent, and long lasting.

JavaScript Code Unit Classification. Other blocking approaches attempt to identify undesirable code based on the structure or behavior of JavaScript code units. Such approaches take as input a single code unit (and sometimes the resulting behavior of that code unit), and train ML classifiers for identifying undesirable code.

Blocking approaches that rely solely on JavaScript behavior or structure are vulnerable to several easy to deploy countermeasures. Most trivially, these approaches do not consider the interaction between code units. An attacker can easily avoid detection by spreading the malicious behavior across multiple code units, having each code unit execute a small enough amount of suspicious behavior to avoid being classified as malicious, and then using a final code unit to combine the quasi-identifiers into a single exfiltrated value. Examples of such work includes the approaches given by Wu et al. [316] and Kaiser et al. [225], both of whom propose ML classifiers that take as input the DOM properties accessed by JavaScript (among other things) to determined whether a code unit is tracking related.

Other approaches attempt to identify tracking-related JavaScript based on the static features of the code, such as names of cookie values, or similar sub-sections in the code. Such approaches are vulnerable to many obfuscation techniques, including using JavaScript's dynamic nature to break identifying strings and labels up across a code base, using dynamic interpretation facilities in the language (e.g. `eval`, `new Function`) to confuse static detection, or simply using different parameters for popular JavaScript post-processing tools (e.g. JSMin [65], Browserify [19], Webpack [101], RequireJS [82]). Ikram et al. [209] proposed one such vulnerable technique, by training a ML classifier to identify static features in JavaScript code labeled by existing filters lists as being tracking related, and using the resulting model to predict whether future JavaScript code is malicious.

Evaluation Issues. Much related work lacks a comprehensive and realistic evaluation. Examples include ambiguous or unstated sources of ground truth comparison (e.g. [147]), unrealistic metrics for what constitutes tracking or non-tracking JavaScript code (e.g. [209] makes the odd assumption that JavaScript code that tracks mouse or keyboard behavior is automatically benign, despite the most popular tracking libraries including the ability to track such functionality [54]), or the decision to (implicitly or explicitly) whitelist all first-party resources (e.g. [147], [318], [316], [225], [203]).

More significantly, much related work proposes resource blocking strategies, but without an evaluation of how their blocking strategy would affect the usability of the web. To name some examples, [147], [316], [287], [225], [209], [203], and [149], all propose strategies for automatically blocking web resources in pages, without determining whether that blocking would harm or break the user-serving goals of websites ([318] is an laudable exception, presenting an indirect measure of site breakage by way of how often users disabled their tool when browsing). Work that presents how much *bad* website behavior an approach avoids, without also presenting how much *beneficial* behavior the approach breaks, is ignoring one half of the ledger, making it difficult to evaluate each work as a practical, deployable solution.

2.2.3 JavaScript Attribution

We next present existing work on a related problem of attributing DOM modifications to responsible JavaScript code units. JavaScript attribution is a necessary part of the broader problem of blocking ads and trackers, as its necessary to trace DOM modifications and network requests back to their originating JavaScript code units. Without attribution, it is difficult-to-impossible to understand which party (or element) is responsible for which undesired activity.

While there have been several efforts to build systems to attribute DOM modifications to JavaScript code units, both in peer-reviewed literature and in deployed software, all existing approaches suffer from completeness and correctness issues. Below we present existing JavaScript attribution approaches and discuss why they are lacking.

JavaScript Stack Walking. The most common JavaScript attribution technique is to interpose on the prototype chain of the methods being observed, throw an exception, and walk the resulting stack object to determine what code unit called the modified (i.e. interposed on) method. This technique is used, for example, by Privacy Badger [76]. The technique has the benefit of not requiring any browser modifications, and of being able to run "online" (e.g. the attribution information is available during execution, allowing for runtime policy decisions).

Unfortunately, stack walking suffers from correctness and completeness issues. First, there are many cases where calling code can mask its identity from the stack, making attribution impossible. Examples include eval'ed code and functions the JavaScript runtime decides to inline for performance purposes. Malicious code can be structured to take advantage of these shortcomings to evade detection [172].

Second, stack walking requires that code be able to modify the prototype objects in the environment, which further requires that the attributing (stack walking) code run before any other code on the page. If untrusted code can gain references to unmodified data structures (e.g. those not interposed on by the attributing code), then the untrusted code can again avoid detection. Browsers do not currently provide any fool-proof way of allowing trusted code to restrict untrusted code from accessing unmodified DOM structures. For example, untrusted code can gain access to unmodified DOM structures by injecting subdocuments and extracting references to from the subdocument, before the attributing code can run in the subdocument.

AdTracker. Recent versions of Chromium include a JavaScript attribution system called AdTracker [55], which attributes DOM modifications made in the Blink rendering system to JavaScript code execution in V8, the browser's JavaScript engine. AdTracker is used by Chromium to detect when third party code modifies the DOM in a way that violates Google's ad policy [277], such as when JavaScript code creates large overlay elements across the page. The code allows the browser to determine which code unit on the page is responsible for the violating changes, instead of holding the hosting page responsible.

AdTracker achieves correctness but lacks completeness. In other words, the cases where AdTracker can correctly do attribution are well defined, but there are certain scenarios where AdTracker is not able to maintain attribution. At a high level, AdTracker can do attribution in *macrotasks*, but not in *microtasks*. Macrotasks are a subset of cases where V8 is invoked by Blink or when one function invokes another within V8. Microtasks can be thought of as an inlining optimization used by V8 to save stack frames, and is used in cases like callback functions in native JavaScript APIs (e.g. callback functions to `Promises`). Effectively, AdTracker trades completeness for performance,[2] which means that a trivial code transformation can circumvent AdTracker.

JSGraph. JSGraph [243] is designed for offline JavaScript attribution. At a high level, JSGraph instruments locations where control is exchanged between Blink and V8, noting which script unit contains the function being called, and treating all subsequent JavaScript functionality as resulting from that script unit. At the next point of transfer from Blink to V8, a new script unit is identified, and following changes are attributed to the new script.

JSGraph writes to a log file, which makes it potentially useful for certain types of offline forensic analysis, but not useful for online content blocking. More significantly, JSGraph suffers from correctness and completeness issues. First, like AdTracker, JSGraph does not provide attribution for functionality optimized into microtasks. Second, JSGraph's attribution provides incorrect results (e.g. unable to link eval'ed created script in a callback to its parent script) in the face of other V8 optimizations, such as deferred parsing, where V8 compiles different sections of a single script unit at different times. Third, JSGraph mixes all frames and subframes loaded in a page together, causing confusion as to which script is making which changes (the script unit identifier used by JSGraph is re-used between frames, so different scripts in different frames can have the same identifier in the same log file).

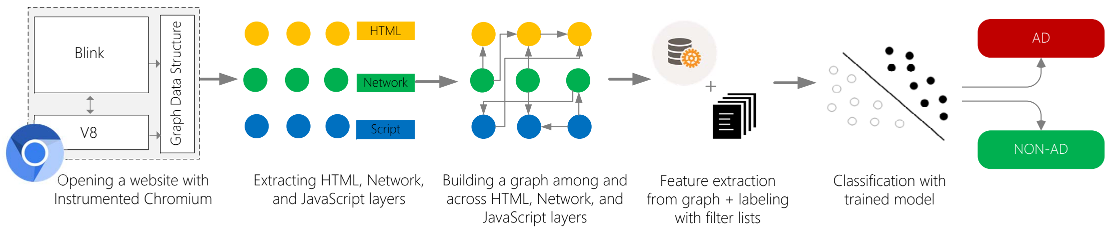

Blink
V8
Graph Data Structure
Opening a website with Instrumented Chromium
HTML
Network
Script
Extracting HTML, Network, and JavaScript layers
Building a graph among and across HTML, Network, and JavaScript layers
Feature extraction from graph + labeling with filter lists
Classification with trained model
AD
NON-AD

Figure 2.1. AdGraph: Our proposed approach for ad and tracking blocking. We instrument Chromium to extract information from HTML structure, network, and JavaScript behavior of a webpage execution as a graph representation. We then extract distinguishing structural and content features from the graph and train a ML model to detect ads and trackers.

2.3 AdGraph Design

In this section we present the design and implementation of AdGraph, an in-browser ML-based approach to block ad and tracking related content on the web. We first describe a novel graph representation of the execution of a website that tracks changes in the HTML structure, behavior and interaction between JavaScript code, and network requests of the page over time. This graph representation allows for tracing the provenance of any DOM change to the responsible party (e.g. JavaScript code, the parser, a network request). Second, we discuss the Chromium instrumentation needed to construct our graph representation. Third, we describe the features AdGraph extracts from our graph representation to distinguish between ad/tracker and benign resources. Finally, we explain the supervised ML classifier and how AdGraph enforces its classification decisions at runtime. Figure 2.1 gives an architectural overview of AdGraph.

[2]These shortcomings are known to the Chromium developers, and are an intentional tradeoff to maximize performance.

2.3.1 Graph Representation

Webpages are parsed and represented as DOM trees in modern browsers. The DOM tree captures relationships among HTML elements (e.g. parent-child, sibling-sibling). In AdGraph, we enrich this existing tree-representation with additional information about the execution and communication of the page, such as edges to capture JavaScript's interactions with HTML elements, or which code unit triggered a given network request. These edge additions transform the DOM tree to a graph. AdGraph uses this graph representation to capture the execution of a webpage.

AdGraph's graph representation of page execution tracks changes in the website's HTML structure, network requests, and JavaScript behavior. The unique graph structure brings several benefits. First, because the graph contains information about the cause and content of every network request and DOM modification during the page's life cycle, the graph allows for tracing the provenance of any change or behavior back to either the responsible JavaScript code unit, or, in the case of initial HTML text, the browser's HTML parser. Second, the graph representation allows for extraction of context-rich features, which are used by AdGraph to identify advertising and tracking related network requests. For example, the graph allows for quick determinations of the source script sending an `AJAX` request, the position, depth, and location of an image request, and whether a subdocument was injected in a page from JavaScript code, among many others. The contextual information captured by these features in AdGraph far exceeds what is available to existing blocking tools, as discussed in Section 2.2.

Next, we explain how AdGraph represents information during a page load as nodes and edges in a graph.

Nodes. AdGraph depicts all elements in a website as one of four types of node: *parser*, *HTML*, *network*, or *script*.

The parser node is a single, special case node that AdGraph uses to attribute document changes and network requests to the HTML parser, instead of script execution. Each graph contains exactly one parser node.

HTML nodes represent HTML elements in the page, and map directly onto the kinds of tags and markup that exist in websites. Examples of HTML nodes include image tags, anchor tags, and paragraph tags. HTML nodes are annotated to store information about the tag type and the tags HTML attributes (e.g. `src` for image tags, `class` and `id` for all tags, and `value` for input tags). HTML text nodes are represented as a special case HTML node, one without a tag type.

Network nodes represent remote resources, and are annotated with the type of resource being requested. Requests for sub-documents (i.e. `iframes`), images, `XMLHTTPRequest` fetches, and others are captured by network nodes.

Script nodes represent each compiled and executed body of JavaScript code in the document. In most cases, these can be thought of as a special type of HTML node, since most scripts in the page are tied to script tags (whether inline or remotely fetched). AdGraph represents script as its own node type though to also capture the other sources of script execution in a page (e.g. `javascript:` URIs).

Edges. AdGraph uses edges to represent the relationship between any two nodes in the graph. All edges in AdGraph are directed. Depending on the execution of pages, the graph may contain cycles. All edges in AdGraph are of one of three types, *structural*, *modification*, and *network*.

Structural edges describe the relationship between two HTML elements on a page (e.g. two HTML nodes). Mirroring the DOM API, edges are inserted to describe parent-child node relationships, and the order of sibling nodes.

Modification edges depict the creation, insertion, removal, deletion, and attribute modification of each HTML node. Each modification edge notes the type of event (e.g. node creation, node modification, etc) and any additional information about the event (e.g. the attributes that were modified, their new values, etc). Each modification edge leaves a script or parser node, and points to the HTML element being modified.

Network edges depict the browser making a request for a remote resource (captured in the graph as a network node). Network edges leave the script or HTML node responsible for the request being made, and point to the network node being requested. Network edges are annotated with the URL being requested.

Composition Examples. These four node types and three edge types together depict changes to DOM state in a website. For example, AdGraph represents an HTML tag `<img src="/example.png">` as an HTML node depicting the `img` tag, a network node depicting the image, and a network edge, leaving the former and pointing to the latter, annotated with the "`/example.png`" URL. As another example, a script modifying the value of a form element would be represented as a script node depicting the relevant JavaScript code, an HTML node describing the form element being modified, and a modification edge describing a modification event, and the new value for the "value" attribute.

2.3.2 Graph Construction

AdGraph's graph representation of page execution requires low level modifications to the browser's fetching, parsing, and JavaScript layers. We implement AdGraph as a modification to the Chromium web browser.[3] The Chromium browser consists of many sub-projects, or modules. The Blink [23] module is responsible for performing network requests, parsing HTML, responding to most kinds of user events, and rendering pages. The V8 [24] module is responsible for parsing and executing JavaScript. Next, we provide a high level overview of the types and scope of our modifications in Chromium for constructing AdGraph's graph representation.

Blink Instrumentation. We instrument Blink to capture anytime a network request is about to be sent, anytime a new HTML node is being created, deleted or otherwise modified (and noting whether the change was due to the parser or JavaScript execution), and anytime control was about to be passed to V8. We further modify each page's execution environment to bind the graph representation of the page to each page's document object. This choice allows us to easily distinguish scripts executing in different frames/sub-documents, a problem that has frustrated prior work (see discussion of JSGraph in Section 2.2.3). Finally, we add instrumentation to allow

[3]The source code of our Chromium implementation is available at: `https://uiowa-irl.github.io/AdGraph/`.

us to map between V8's identifers for script units, and the sources of script in the executing site (e.g. script tags, eval'ed scripts, script executed by extensions).[4]

V8 Instrumentation. We also modify V8 to add instrumentation points to allow us to track anytime a script is compiled, and anytime control changes between script units. We accomplish this by associating every function and global scope to the script they are compiled from. We then can note every time a new scope is entered, and attribute any document modifications or network requests to that script, until the scope is exited.

V8 contains several optimizations that make this general approach insufficient. First, V8 sometimes defers parsing of subsections of JavaScript code. A partial list of such cases includes eval'ed code, code compiled with the `Function` constructor, and anonymous functions provided as callbacks for some built in functions (e.g. `setTimeout`). To handle these cases, AdGraph not only maps functions to script units but also sub-scripts to scripts.

Second, V8 implements microtasks that make attribution difficult. Microtasks allow for some memory savings (much of the type information and vtable look-up overhead is skipped) and reduce some book-keeping overhead. Tracking attribution of DOM changes in microtasks is difficult because, at this level, V8 no longer tracks functions as C++ objects, but as compiled bytecode, requiring a different approach to determining which script unit "owned" any given execution. AdGraph solves

[4]The architectural independence between the V8 and Blink projects made this an unexpectedly difficult problem to solve, with many unanticipated corner cases that were not discovered until we subjected AdGraph to extensive automatic and manual testing.

this problem through additional instrumentation, and some runtime stack scanning, yielding completeness at the cost of a minor performance overhead.

JavaScript Attribution Example. AdGraph is able to attribute DOM modifications and network events to script units in cases where existing techniques fail. We give a representative example in code snippet 2.1.

This code uses `eval` to parse and execute a string as JavaScript code. The resulting code uses a `Promise` in a `setTimeout` callback. This `Promise` callback is optimized in V8 as a microtask, which evades the attribution techniques used in current work (e.g. PrivacyBadger / stack walking, AdTracker, JSGraph, discussed in Section 2.2.3). Existing tools would not be able to recognize that this code unit was responsible for the image fetched in the `Promise` callback.

```
<html>
   <script>
       ...
       eval("setTimeout(function xyz() {
       const p = Promise.resolve('A');
       p.then(function abc(_) {
         var img = document.createElement('img');
         img.setAttribute('id','ad_image');
         img.src = 'adnetwork.com/ad.png';
       }) }, 5) ");
       ...
   </script>
</html>
```

Script 2.1. A microtask in an eval created script loading an ad.

AdGraph, though, is able to correctly attribute the image request to this code unit. Figure 2.2 shows how this execution pattern would be stored in AdGraph. Specifically, the edge between nodes 2 and 4 records the attribution of the `eval` call to the responsible JavaScript code unit, and the edge between nodes 7 and 9 in record that the image request is a result of code executed in the microtask. Existing approaches would either miss the edge between 2 and 4, or 7 or 9.

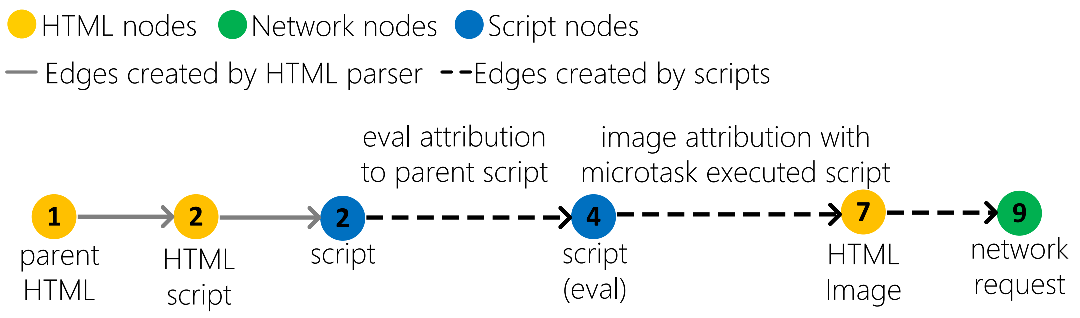


Figure 2.2. AdGraph's representation of example code snippet 2.1. Node numbers correspond to line numbers in code snippet 2.1. This example highlights connections and attributions not possible in existing techniques.

2.3.3 Feature Extraction

Next, we present the features that AdGraph extracts from the graph to distinguish ads and trackers from functional resources. These features are designed based on our domain knowledge and expert intuition. Specifically, we manually analyze a large number of websites and try to design features that would distinguish ad/tracking related resources from functional (or benign) resources.

The extracted features broadly fall into two categories: "structural" (features that consider the relationship between nodes and edges in the graph) and "content" (features that depend on the values and attributes of nodes in isolation from their connections). In total we extract 64 structural and content features. Table 2.2 gives a summary and representative examples of features from each category. Below we provide a high-level description of structural and content features. More detailed analysis of features and their robustness is presented in Section 2.4.4.

Structural Features
Graph size (# of nodes, # of edges, and nodes/edge ratio)
Degree (in, out, in+out, and average degree connectivity)
Number of siblings (node and parents)
Modifications by scripts (node and parents)
Parent's attributes
Parent degree (in, out, in+out, and average degree connectivity)
Sibling's attributes
Ascendant's attributes
Descendant of a script
Ascendant's script properties
Parent is an eval script
Content Features
Request type (e.g. `iframe`, `image`)
Ad keywords in request (e.g. banner, sponsor)
Ad or screen dimensions in URL
Valid query string parameters
Length of URL
Domain party
Sub-domain check
Base domain in query string
Semi-colon in query string

Table 2.2. Summarized feature set used by AdGraph.

Structural Features. Structural features target the relationship between elements in a page (e.g. the relationship between a network request and the responsible script unit, or a HTML nodes' parents, siblings and cousin HTML nodes). Examples of structural features include whether a node's *parents* have ad-related values for the `class` attribute, the tag names of the node's *siblings*, or how deeply nested in the document's structure a given node is.

Structural features also consider the interaction between JavaScript code, and the resource being requested. These features rely on AdGraph's instrumentation of Blink and V8. Examples of JavaScript features include whether the node initiating a network request was inserted by JavaScript code, the number of scripts that have "touched" the node issuing the request, and, in the case of requests that are not directly related to HTML elements (e.g. AJAX), whether the JavaScript code initiating the request was inlined in the document or fetched from a third-party.

Content Features. Content features relate to values attached to individual nodes in the graph (and not the connections *between* nodes in the graph). The most significant value considered is the URL of the resource being requested. These content features are similar to what most existing content blocking tools use. AdGraph's specific set of features though is unique. Examples of AdGraph's content features include whether the origin of the resource being requested is first-or-third party, the number of path segments in the URL being requested, and whether the URL contains any ad-related keywords.

2.3.4 Classification

AdGraph uses random forest [155], a well-known ensemble supervised ML classification algorithm. Random forest combines decisions from multiple decision trees, each constructed using a different bootstrap sample of the data, by choosing the mode of the predicted class distribution. Each node for a decision tree is split using the best among the subset of features selected at random. This feature selection mechanism provides robustness against over-fitting issues. We configure random forest as an ensemble of 100 decision trees with each decision tree trained using $int(\log M + 1)$ features, where M is the total number of features.

AdGraph's random forest model classifies network requests based on the provenance (creation and modification history) of a node and the context around it. These classification decisions are made before network request are sent, so that AdGraph can prevent network communication with ad and tracking related parties. A single node may initiate many network requests (either due to it being a script node, or being modified by script to reference multiple resources). As a result, any node may be responsible for an arbitrary number of network requests. AdGraph classifies three categories of network requests:

1. Requests initiated by the webpage's HTML (e.g. the image referenced by an `<img>` tag's `src` attribute).

2. Requests initiated by a node's attribute change (e.g. a new background image being downloaded due to a new CSS style rule applying because of a mouse hover).

3. Requests initiated directly by JavaScript code (e.g. `AJAX` requests, image objects not inserted into the DOM).

2.4 AdGraph Evaluation

In this section we evaluate the accuracy, usability, and performance of AdGraph when applied to live, real-world, popular websites.

2.4.1 Accuracy

We first evaluate how accurately AdGraph is able to distinguish advertising and tracking content from benign web resources.

Ground Truth. To evaluate AdGraph's accuracy, we first need to gather a ground truth to label a large number of ad/tracking related network requests. We generate a trusted set of ground truth labels by combining popular crowdsourced filter lists that target advertising and/or tracking, and applying them to popular websites. Table 2.3 lists the 8 popular filter lists we combine to form our ground truth. These lists collectively contain more than a hundred thousand crowdsourced rules for determining whether a URL serves advertising and/or tracking content.

List	# Rules	Citation
EasyList	72,660	[36]
EasyPrivacy	15,507	[39]
Anti-Adblock Killer	1,964	[11]
Warning Removal List	378	[97]
Blockzilla	1,155	[15]
Fanboy Annoyances List	38,675	[41]
Peter Lowe's List	2,962	[75]
Squid Blacklist	4,485	[87]

Table 2.3. Crowd sourced filter lists used as ground truth for identifying ad and tracking resources. Rule counts are as of Nov. 12, 2018.

Advertising resources include audio-visual promotional content on a website. Tracking resources collect unique identifiers (e.g., cookies) and sensitive information (e.g., browsing history) about users. In practice, there is no clear division between ad and tracking resources. Many resources on the web not only serve advertising images and videos but also track the users who view it. It is also noteworthy that EasyList (to block ads) and EasyPrivacy (to block trackers) have a significant overlap. Because of this overlap, we do not attempt to distinguish between advertising and tracking resources.

Note that while these crowdsourced filter lists suffer from well-known shortcomings [295], we treat them as “trusted” for three reasons. First, they are reasonably accurate for top-ranked websites even though they suffer on low-ranked websites [186, 255]. Second, a more accurate alternative, building a web-scale, manually generated, expert set of labels would require labor and resources far beyond what is feasible for a research project. Third, we use several filter lists together to maximize their coverage and reduce false negatives.

We visit the homepages of the Alexa top-10K websites with our instrumented Chromium browser. We expect that the top-10K websites is a diverse and large enough set to contain most common browsing behaviors. We limit our sample of websites to the 10K most popular sites to avoid biasing our sample; previous work has found that popular filter lists work reasonably well for popular sites [186, 255]. Applying crowdsourced filter lists to unpopular sites (sites that, almost by definition, the curators of filter lists are less likely to visit) risks skewing our data set to include a large number of false negatives (i.e. advertising and tracking resources that filter list authors have not encountered).

We apply filter lists to websites in the following manner. We visit the homepage of each site with our instrumented version of Chromium and wait for each page to finish loading (or 120 seconds, whichever occurs first). Next we record every URL of every resource fetched when loading and rendering each page. We then label each fetched resource URL as AD and NON-AD, based on the whether they are identified as ad or tracking related by any of a set of filter lists. Our final labeled dataset consists of 540,341 URLs, fetched from 8,998 successfully crawled domains.[5]

Results. We use the random forest model to classify each fetched URL. We then compare each predicted label with the label derived from our ground truth data set, the set of filter lists described above. We then evaluate how accurately our model can reproduce the filter list labels through a stratified 10-fold cross validation, and report the average accuracy. AdGraph classifies AD and NON-AD with a high degree of accuracy, achieving 95.33% accuracy, with 89.1% precision, and 86.6% recall.

As Table 2.4 shows, AdGraph classifies web resources with a high degree of accuracy. We note that AdGraph is more accurate in classifying visual resources such as images (98.95% accuracy) and CSS (96.32% accuracy) than invisible resources like JavaScript (90.52% accuracy) and `AJAX` requests (93.55% accuracy). This suggests an interesting possibility, that AdGraph's labels are correct, and filter lists

[5]The success rate of about 90% in our crawl is in line with those of previous studies [186, 255].

Resource	# Resources	Blocked by Filter Lists	Blocked by AdGraph	Precision	Recall	FPR	FNR	Accuracy
Image	201,785	11,584	10,228	93.09%	88.29%	0.39%	11.71%	98.95%
Script	167,533	67,959	60,030	88.32%	88.33%	7.97%	11.67%	90.52%
CSS	124,207	9,255	5,834	83.61%	63.03%	0.99%	36.97%	96.32%
AJAX	24,365	8,305	7,442	91.31%	89.60%	4.40%	10.40%	93.55%
iFrame	20,091	7,745	7,244	92.31%	93.53%	4.88%	6.47%	94.50%
Video	2,360	23	14	93.33%	60.86%	0.04%	39.14%	99.57%
Total	540,341	104,871	90,792	89.1%	86.6%	2.56%	13.4%	95.33%

Table 2.4. Number of resources, broken out by type, encountered during our crawl, and incidence of ad and tracking content, as determined by popular filter lists and AdGraph.

miss-classify invisible resources due to their reliance on human crowdsourced feedback. We investigate this possibility, and more broadly the causes of disagreements between AdGraph and filter lists in the next subsection.

2.4.2 Disagreements Between AdGraph and Filter Lists

We now manually analyze the cases where AdGraph disagrees with filter lists to determine which labeling is incorrect, AdGraph's or filter lists'. Overall, we find that AdGraph is able to identify many advertising and tracking resources missed by filter lists. We also find that AdGraph correctly identifies many resources as benign which filter lists incorrectly block. These findings imply that AdGraph's actual accuracy is higher than 95.33%.

Methodology. To understand why AdGraph disagrees with existing filter lists, we perform a manual analysis of a sample of network requests where AdGraph identifies a resource as ad/tracking related but filter lists identify as benign (i.e. false positives) and where filter lists identify a resource as ad/tracking related but AdGraph identifies as benign (i.e. false negatives). We select these "false positives" and "false negatives" from the most frequent advertising and tracking related resource types: JavaScript code units and images. We manually analyze all of the 282 distinct images and a random sample of 100 script URLs that AdGraph classifies as AD but filter lists label as NON-AD and a random sample 300 images and 100 script URLs that AdGraph classifies as NON-AD but filter lists label as AD. The goal of our manual analysis is to assign each JavaScript unit or image to one of the following labels:

1. **True Positive**: AdGraph's classification is correct and the filter lists are incorrect; the resource is related to advertising or tracking.

2. **False Positive**: The label by filter lists is correct and AdGraph's classification is incorrect; the resource is not related to advertising or tracking.

3. **True Negative**: AdGraph's classification is correct and the filter lists are incorrect; the resource is not related to advertising or tracking.

4. **False Negative**: The label by filter lists is correct and AdGraph's classification is incorrect; the resource is related to advertising or tracking.

5. **Mixed**: The resource is dual purpose (i.e. both ad/tracker and benign). This label is only used for script resources.

6. **Undecidable**: It was not possible to determine whether the resource is an ad/tracker.

We decide whether an image was advertising or tracking related through the following three steps. First, we label all tracking pixels (1×1 sized images used to initiate a cookie or similar state-laden communication) as "true positive" if AdGraph classified it as AD and "false negative" if AdGraph classified it as NON-AD. Second, we consider the content of each image and look for text indicating advertising, such as the word "sponsored", prices, or mentions of marketers. If the image has such text, we consider the image as an advertisement and label it "true positive" if AdGraph classified it as AD and "false negative" if AdGraph classified it as NON-AD. If the case is ambiguous, such as an image of a product that could either be advertising or a third-party discussion of the product, we use the “undecidable” label. Third, we label all remaining cases as “false positive” if AdGraph classified them as AD and “true negative” if AdGraph classified them as NON-AD.

Deciding the labels for the sampled script resources is more challenging. Determining the purpose of a JavaScript file requires inspecting and understanding large amounts of code, most of which has no documentation, and which is in many cases minified or obfuscated. We label a script as “true positive” (advertising or tracking related) if most of the script performs any of the following functionality: cookie transmission, passive device fingerprinting, communication with known ad or tracking services, sending beacons, or modifying DOM elements whose attributes are highly indicative of an ad (e.g. creating an image carousel with the `id` “ad-carousel”); and AdGraph classified it as AD and “false negative” if AdGraph classified it as NON-AD. If the script primarily includes functionality distinct from the above (e.g. form validation, non-ad-related DOM modification, first-party `AJAX` server communication), we label it as “false positive” if AdGraph classified it as AD and “true negative” if AdGraph classified it as NON-AD. If the script contains significant amounts of both categories of functionality, we label the script as “mixed”. In cases where the functionality is not discernable, we use the “undecidable” label.

False Positive Analysis. Table 2.5 presents the results of our disagreement analysis for false positives. In cases where AdGraph identifies a resource as suspect, and filter lists label it as benign, AdGraph’s determination is correct 11.0%–33.0% of the time for JavaScript and 46.8% of the time for images.

AdGraph is often able to detect advertising and tracking resources that are missed by filter lists. For example, AdGraph blocks a 1x1 pixel on `cbs.com` that includes a tracking identifier in its query string. In another example, AdGraph blocks a script (`js1`) on `nikkan-gendai.com` that performs browser fingerprinting. Filter lists likely missed these resources because they are often slow to catch up when websites introduce changes [215].

There are however several false positives that are actual mistakes by AdGraph. For example, AdGraph blocks a third-party dual purpose script (`avcplayer.js`), a video player library that also serves ads, on `inquirer.net`. Interestingly, AdGraph detects many such dual-purposed scripts that are beyond the ability of binary-label filter lists.

These results demonstrate that AdGraph is able to identify many edge case resources (e.g. mixed-use) that can be used to refine future versions of AdGraph. As discussed in Section 2.5.2, AdGraph can be extended to handle such mistakes by implementing more fine-grained blocking.

	Image		Script	
	#	%	#	%
True Positive	132	46.8%	11	11.0%
False Positive	129	45.7%	63	63.0%
Mixed	0	0%	22	22.0%
Undecidable	21	7.4%	4	4.0%

Table 2.5. Results of manual analysis of a sample of cases where AdGraph classifies a resource as AD and filter lists label it as NON-AD.

False Negative Analysis. Table 2.6 presents the results of our disagreement analysis for false negatives. In cases where AdGraph identifies a resource as benign, and filter lists label it as suspect, AdGraph's determination is correct 22%–32% of the time for JavaScript and 27.7% of the time for images.

Again, AdGraph is often able to identify benign content that is incorrectly over-blocked by filter lists. For example, AdGraph does not block `histats.com` when visited as a first-party in our crawl, but this domain is blanketly blocked by the Blockzilla filter list even when visited as a first-party. In another example, AdGraph does not block a social media icon `facebook-gray.svg` (served on `postimees.ee` as a first-party resource) and a privacy-preserving analytics script `piwik.js` (served on `futbol24.com` as a first-party resource). It can be argued that many of these resources are neither ads nor pose a tracking threat [40,229]. Filter lists over-block in such cases because of the inclusion of overly broad rules (e.g. blocking entire domains, or any URL containing a given string).

There are however several false negatives that are actual mistakes by AdGraph. For example, AdGraph misses `fingerprint2.min.js` served by a CDN `cloudflare.com` on `index.hr`. AdGraph likely made this mistake because a popular third-party CDN, which is typically used to serve functional content, is used to serve a fingerprinting script. As discussed in Section 2.5.2, AdGraph can be extended to handle such mistakes by extracting new features from JavaScript APIs.

	Image		Script	
	#	%	#	%
True Negative	83	27.7%	22	22%
False Negative	180	60.0%	55	55%
Mixed	0	0%	10	10%
Undecidable	37	12.3%	13	13%

Table 2.6. Results of manual analysis of a sample of cases where AdGraph classifies a resource as NON-AD and filter lists label it as AD.

2.4.3 Site Breakage

Content blocking tools carry the risk of breaking benign site functionality. Content blockers prevent resources that the website expects to be in place from being retrieved, which can have the carry over effect of harming desireable site functionality, especially when tools mistakenly block benign resources [60]. Thus assessing the usefulness of a content blocking approach must also include an evaluation of how many sites are "broken" by the intervention.

Next we evaluate how often, and to what degree, AdGraph breaks benign (i.e. user desired) website functionality. We do so by having two human reviewers visit a sample of popular websites using AdGraph, and having them independently record their assessment of whether the site worked correctly. We find that AdGraph only affects benign functionality on a small number of sites, and at a rate equal to or less than popular filter lists.

Methodology. We estimate how many sites AdGraph breaks by having two evaluators use AdGraph on a sample of popular websites and independently record their determination of how AdGraph impacts the site's functionality. Because of the time consuming nature of the task, we select a smaller sample of sites for this breakage evaluation than we use for the accuracy evaluation.

Our evaluators use AdGraph on two sets of websites: first the Alexa top-10 websites, and second on a random sample of 100 websites from the Alexa top-1K list, resulting in a total of 110 sites for breakage evaluation.

Automatic site breakage assessment is challenging due to the complexity of modern web applications [260, 318]. Unfortunately, manual inspection for site breakage assessment is not only time-consuming but also likely to lose completeness as the functionalities of a website are often triggered by certain events that may be hard to manually cover exhaustively. As a tradeoff, we adopt the approach from [294], which is a manual analysis but focuses on *the user's perspective*. In other words, we intentionally ignore the breakages that only affect the website owner as they do not have any impact on user experience.

For each website, our evaluators independently perform the following steps.

1. Open the website with stock Chromium, as a control, and perform as many actions as possible within two minutes. We instruct our evaluators to exercise the kinds of behaviors that would be common on each site. For example, in a news site this might be browsing through an article; on a e-commerce site this might include searching for a product and proceeding to checkout etc.

2. Open the website with AdGraph, repeat the actions performed above, and assign a breakage level of

   (a) **no breakage** if there is no perceptible difference between AdGraph and stock Chromium;

   (b) **minor breakage** if the browsing experience is altered, but objective of the visit can still be completed; or

   (c) **major breakage** if objective of the visit cannot be completed.

3. Open the website with Adblock Plus[6], repeat the actions, and assign a breakage level as above.

To account for the subjective nature of this analysis, we have each evaluator visit the same sites, at similar times, and determine their "breakage" scores independently. Our evaluators give the same score 87.7% of the time, supporting the significance of their analysis.

Tool	No breakage		Major		Minor		Crash	
	#	%	#	%	#	%	#	%
AdGraph	93.5	85.0%	6.5	5.9%	7.5	6.8%	2.5	2.3%
Filter lists	97.5	88.6%	7	6.4%	4	3.6%	1.5	1.4%

Table 2.7. Breakdown of breakage analysis results (# columns are the average of two independent scores.)

Results. Table 2.7 reports the site breakage assessments as the average of two reviewers. The evaluation shows that AdGraph and filter lists are comparable in

[6]Adblock Plus is configured with the same 8 filter lists that are used to train AdGraph.

terms of site breakage. AdGraph and filter lists do not cause any breakage on 85.0% and 88.6% of the sites, respectively. The major breakage rate (5.9%) is also on par with the filter lists (6.4%). We also note that AdGraph's breakage is much lower than other commonly used privacy oriented browsers (e.g. 16.3% for Tor Browser [294]).

2.4.4 Feature Analysis

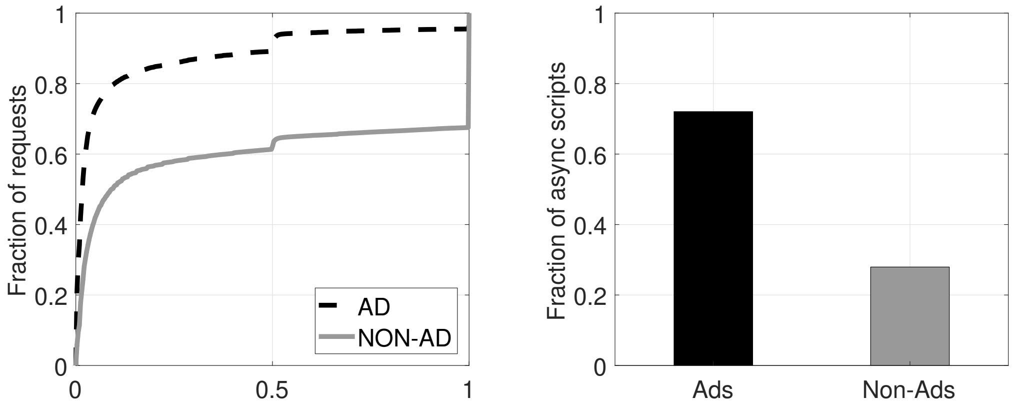
Next, we discuss the intuitions behind some of the features used in AdGraph, and evaluate their ability to distinguish ad/tracking content from benign content. We describe some of the features that are most useful (in terms of information gain [219]) in AdGraph's predictions.

Structural Features. Two of the structural features that provided the highest information gain are a node's *average degree connectivity* and its *parents' attributes.*

We expect AD nodes to have lower *average degree connectivity*, since the interaction of these nodes is confined to only ad/tracking content, and thus appear in less connected cliques. Conversely, we expect that NON-AD nodes appear alongside, and interact with, functional content more, and thus have higher *average degree connectivity.* Our results in Figure 2.3a support this intuition. AD nodes do indeed have have lower *average degree connectivity* than NON-AD nodes.

We also expect the parents of AD nodes to have different attributes than NON-AD nodes. This intuition came from the expectation that AD nodes are more likely to follow common practices and standards, such as those proposed by the Interactive Advertising Bureau (IAB) [63]. For example, IAB's LEAN standard [58] requires ad related scripts to load asynchronously (indicated by the presence of the `async` attribute on a script node). We capture this intuition in a feature by considering the attributes of each network requests' parent nodes (in our graph representation, the parent of a network request might be the script element that initiates the network request). Our results in Figure 2.3b support this intuition. The parents of AD nodes with `script` tag name were 3 times more likely to have the `async` attribute than NON-AD nodes.

We note that some structural features are more robust to obfuscation than others. For example, to flummox the classifier, it would be more challenging for an adversary to manipulate a node's average degree connectivity (which depends on all of the node's neighbors) than it would be to manipulate the attributes of a parent node.



(a) average degree connectivity (b) async script

Figure 2.3. Conditional distributions for structural features.

Content Features. Two of the content features that provided the highest information gain are a node's *domain party* and its *URL length.*

We expect AD nodes to be more likely to come from third-party domains than NON-AD nodes. We capture this intuition in a boolean feature, recording whether the domain of a network request differs from domain of the first party document. Figure 2.4a shows this intuition to be correct. More than 90% of the ads came from third-party domains.

We also expect AD nodes to include a large number of query parameters in their URLs. We capture this intuition by using a request's *URL length* as a numeric feature. Figure 2.4b shows this intuition to be correct. AD node URLs were on average longer than NON-AD node URLs.

We again note that some content features are more robust to obfuscation than others. For example, to flummox the classifier, it would be more challenging for an adversary to switch ads/trackers from third-party to first-party that it would be to manipulate the length of a URL.

Ablation Analysis. Next, we separately evaluate structural and content features in terms of their contribution to AdGraph's accuracy. To this end, we train additional classifiers separately, one using only structural features, the other using only content features. While structural features and content features have comparable accuracy they provide complementary information, which when used together improve AdGraph's accuracy. For example, excluding structural features results in a decrease of 6.6% in precision, 8.7% in recall, and 2.7% in accuracy.

We also expect structural features to be more robust than content features.

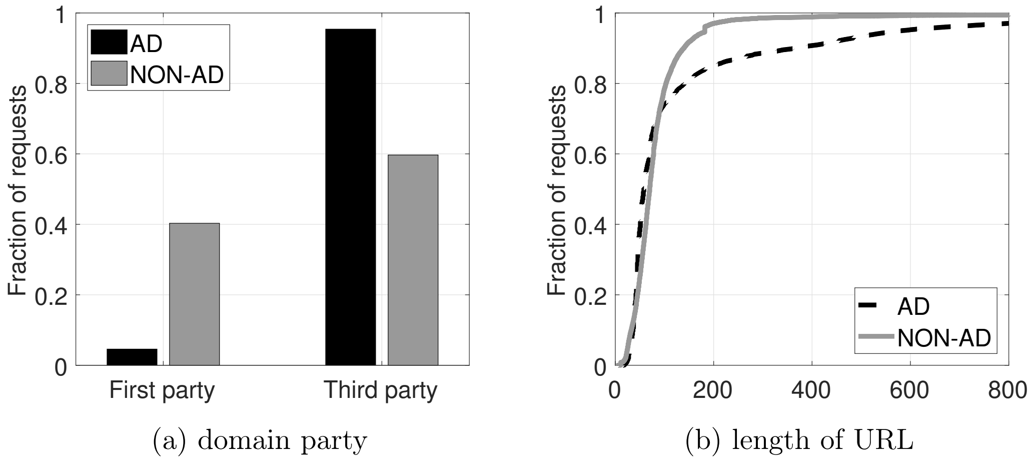


Figure 2.4. Conditional distributions for content features.

Structural features consider neighboring graph structure of a node while content features only consider a node in isolation. To manipulate the structural features, an adversary would need to change the target node, its neighbors, and subsequently their neighbors. Manipulating content features would only require changing the target node.

Thus, we conclude that the graph-based representation of AdGraph, as captured by structural features, contributes to its accuracy and robustness.

2.4.5 Tradeoffs in Browser Instrumentation

Recall from Section 2.3.2 that AdGraph modifies Chromium to attribute DOM modifications to JavaScript code units. This is different from most existing content blocking tools that operate at the extension layer. AdGraph's browser instrumentation is a trade off; it gains attribution accuracy at the cost of ease of distribution. This raises the question of whether AdGraph can instead be implemented as a browser extension on any web browser.

We investigate this question by implementing AdGraph as a browser extension, using the best possible attribution option available at the extension layer (JavaScript stack walking, discussed in Section 2.2.3). We test the accuracy of the best possible extension implementation of AdGraph by re-crawling the Alexa-10k with a modified version of AdGraph, using the same methodology described in Section 2.4.1. This modified version of AdGraph uses JavaScript stack walking to attribute DOM modifications to script units, instead of the Blink and V8 modifications. We then train and test ML classifier on the graphs constructed using JavaScript stack walking.

We compare the accuracy of this best-possible-extension implementation to our in-browser implementation of AdGraph. We find that implementing AdGraph as a browser extension significantly reduces classification accuracy. Implementing AdGraph as a browser extension degrades precision by 1.5%, recall by 16%, and accuracy by 2.3%. Thus, the mistakes JavaScript stack walking makes in attribution lead to more errors in classification. We conclude that costs of implementing AdGraph's as a set of browser modifications (i.e. difficulty in distribution) is more than offset by the benefits (i.e. increased classification accuracy), and that AdGraph is best implemented as Blink and V8 modifications.

2.4.6 Performance

We evaluate AdGraph's performance as compared to stock Chromium and Adblock Plus. AdGraph performs faster in most cases than the most popular block-

Resource Type	AdGraph faster	Chromium faster
Image	24.59%	14.92%
Script	20.82%	17.96%
CSS	6.47%	0.79%
AJAX	48.03%	36.14%
iFrame	37.66%	30.47%
Video	7.14%	6.20%

Table 2.8. Comparison of average percentage of resources AdGraph blocks on sites where AdGraph outperforms Chromium, and vise versa. For all resource types, AdGraph performs faster when more resources are blocked.

ing tool, Adblock Plus, and in many cases results in faster performance than stock Chromium. This is the result of both careful engineering in AdGraph's implementation, and AdGraph's instrumentation overhead (often) being more than offset by the network and rendering savings gained by having to fetch and render less page content (i.e. the content blocked by AdGraph).

To measure whether AdGraph is a practical blocking solution, we compare the performance of AdGraph, stock Chromium, and Chromium with Adblock Plus installed (using Adblock Plus's default configuration) on the Alexa 1K. Our simulated network uses a 10 Mbps downlink with a latency of 100ms. We visit the landing page of each website 10 times and record the average page load time (measured as the difference between the DOM's `navigationStart` and `loadEventEnd` events). Figure 2.5 presents AdGraph's page load time compared to stock Chromium, and Chromium with Adblock Plus.[6]

AdGraph performs faster than Chromium on 42% of websites. AdGraph is often faster than stock Chromium because it needs to fetch and render fewer resources than stock Chromium (i.e. the network requests blocked by AdGraph). Table 2.8 shows that AdGraph outperforms Chromium on sites where it blocks more ad/-tracking content, as compared to sites where it blocks less. Put differently, the more content AdGraph blocks, the more it is able to make up for the instrumentation and classification overhead with network and rendering savings.

AdGraph performs faster than Adblock Plus on 78% of websites. AdGraph is faster than Adblock Plus for two reasons. First, Adblock Plus implements element hiding rules (i.e. rules describing elements that are still fetched, but hidden when rendering), which carries with it an enforcement and display-reflow overhead AdGraph does not share. Second, AdGraph's blocking logic is implemented in-browser which leads to performance improvement over Adblock Plus's implementation at the extension layer.

Overall, we conclude that AdGraph is performant enough to be a practical online content blocking solution. Future implementation refinements, and the exploration of cheaper features, could further improve AdGraph's performance.

2.5 Discussions

2.5.1 Offline Application of AdGraph

AdGraph is designed and implemented to be used as an online, in-browser blocking tool. This is different than most blocking tools, which operate as extensions on mainstream browsers (e.g. Chrome, Firefox). Since AdGraph requires browser instrumentation, it cannot be directly used by extension-based blockers that rely on

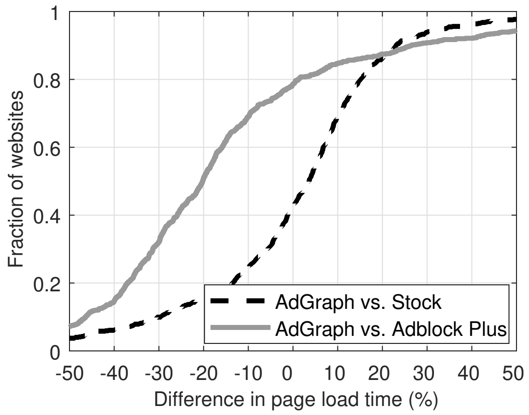


Figure 2.5. Overhead ratio in terms of page load time.

offline manually curated filter lists. AdGraph can benefit existing blocking tools through the creation and maintenance of filter lists in several ways.

First, the accuracy of filter lists suffers because of they are manual generated and rely on informal crowdsourced feedback. As discussed in Section 2.4.2, filter list maintainers can analyze disagreements between AdGraph and filter lists to identify and fix potential inaccuracies in filter lists

Second, AdGraph can support the generation of filter lists targeting underserved languages or region on the web. Filter lists are inherently skewed towards popular websites and languages because of their larger and more active blocking user base [186, 255]. Filter list maintainers receive much less feedback to fix inaccuracies on less popular websites. This makes the creation and maintenance of filter lists for underserved regions (geographically and linguistically) difficult, since these sites have less visitors. Language/region specific filter lists are updated much less frequently than general (and mostly English targeting) filter lists like EasyList. Many languages and regions (most notably Africa) do not have dedicated filter lists at all. AdGraph can assist in automatically generating filter lists for smaller or underserved regions.

Third, the manual nature of filter list maintenance has lead to increasing number of outdated and stale rules. Filter list rules can quickly get outdated because most websites frequently update and are highly dynamic. Prior research found that filter lists can take months to update in response to such changes [215]. Even when filter lists are updated, new rules are typically added (rather than editing old rules) which leads to accumulation of stale rules over time. Prior research reported that only 200 rules account for 90% blocking activity for EasyList [295]. In other words, the number of rare-to-never used rules in EasyList is increasing over time which has performance implications. AdGraph can by used by filter list maintainers to periodically audit filter lists for identifying outdated and stale rules.

2.5.2 AdGraph Limitations And Future Improvements

Ground Truth. AdGraph relies on filter lists as ground truth to train a ML classifier for detecting ads/trackers. As we showed in Sections 2.4.2, filter lists suffer from inaccuracies due to both false negatives and false positives. AdGraph can address these inaccuracies in ground truth by gathering valuable user feedback when it is deployed at scale. AdGraph can retrain its ML classifier periodically on improved ground truth as user feedback is received.

Features. The features used by AdGraph are manually designed, based on our domain knowledge and expert intuition, with the goal of achieving decent accuracy. Note that the feature set is by no means "complete" and there is room for additional feature engineering to further improve accuracy. New features can be systematically discovered by incorporating user feedback, which may reveal new characteristics of ads/trackers over time that are not currently covered by AdGraph. New features may require addition of new instrumentation points such as JavaScript APIs or new feature modalities altogether, such as image based perceptual information [85, 299, 300].

Classification Granularity. AdGraph is currently designed to make binary decisions to either block or allow network requests. However, as discussed in Section 2.4.2, AdGraph is also able to detect cases when a single JavaScript is used for both ad/tracking and functional content. The cases where JavaScript code serves dual-purpose are challenging because blocking the request may break page functionality, while allowing the request will allow ads/trackers on the page. AdGraph's context rich classification approach can be adapted to more than two labels for handling such dual-purpose scripts. Specifically, AdGraph can be trained at a more granular level to distinguish between ads/trackers, functional, and dual-purpose resources. AdGraph can respond to such dual-purpose resources with different remediations than outright allowing/blocking, such as giving those scripts a reduce set of DOM capabilities (e.g. reading/writing cookies [48, 310], access to certain APIs [17, 44]), or blocking network requests issued from such scripts.

2.6 Conclusion

In this chapter we proposed AdGraph, a graph-based ML approach to ad and tracker blocking. We designed AdGraph to leverage fine-grained interactions between network requests, DOM elements, and JavaScript code execution to construct a graph representation that is used to trace relationships between ads/trackers and the rest of the page content. To implement AdGraph, we instrumented Chromium's rendering engine (Blink) and JavaScript execution engine (V8) to efficiently gather complete HTML, HTTP, and JavaScript information during page load. We leveraged this rich context by extracting distinguishing features to train a ML classifier for in-browser ad and tracker blocking at runtime.

We showed that AdGraph not only blocks ads/trackers with 95.33% accuracy but uncovers many ad/tracker and functional resources that are missed and over-blocked by filter lists, respectively. We also showed that AdGraph's breakage is on par with filter lists. In addition to high accuracy and comparable breakage, we showed that AdGraph loads pages much faster as compared to existing content blocking tools.

We designed AdGraph to be used both online (for in-browser blocking) and offline (filter list curation). Since the vast majority of extension-based blocking tools currently rely on manually curated filer lists, AdGraph's offline use case will aid filter list monitoring and maintenance. Overall, we believe that AdGraph significantly advances the state-of-the-art in ad and tracker blocking.

CHAPTER 3

WebGraph: CAPTURING ADVERTISING AND TRACKING INFORMATION FLOWS FOR ROBUST BLOCKING

3.1 Introduction

Users rely on privacy-enhancing blocking tools to protect themselves from online advertising and tracking. Many of these tools—including uBlock Origin [94], Ghostery [51], Firefox [252, 253], Edge [256], and Brave [154]—rely on manually curated filter lists [31,39,114] to block advertising and tracking. The research community is actively developing machine learning (ML) approaches to automate the detection of advertising and tracking and make filter lists more comprehensive. The first generation of ML-based blocking approaches analyze network requests [149, 203, 287] or JavaScript code [209, 225, 316] to learn distinctive behaviors of advertising and tracking. However, these ML-based blocking approaches are highly susceptible to adversarial evasion techniques that are already found in the wild, including URL obfuscation [295] and code obfuscation [291]. To address this limitation, the next generation of ML-based blocking approaches leverage cross-layer graph information from multiple layers of the web stack [216, 290]. These approaches claim better robustness to evasion, as compared to single-layer approaches, due to their use of *structural* features of the graph (i.e., the hierarchy of resource inclusions) in addition to traditional *content* features (i.e., the resource's network location or response content).

In this chapter, we show that state-of-the-art ad and tracker detection approaches, such as AdGraph [216], are susceptible to adversarial evasion due to their disproportionate reliance on easy-to-manipulate content features. We show that a third-party adversary can achieve 8% evasion success by manipulating URLs of its resources. Worse yet, an adversary can achieve near-perfect evasion—as high as a 96% success rate—if they collude with the first party, e.g, by using the CNAME cloaking technique already deployed by some trackers [174, 179].

We introduce WebGraph, the first ML-based ad and tracker blocking approach that *does not rely on content features*. WebGraph improves the cross-layer graph representation by capturing a fundamental property of advertising and tracking services (ATS): the flow of information from one entity to the browser's storage, the network, and to other entities loaded on a page. The intuition behind adding these features is to focus on the *actions* of the advertising and tracking services, rather than the *contents* of their resources. We posit that actions are harder to obfuscate. Advertising and tracking scripts need to generate and store identifiers for users, and those identifiers must be shared with any other entity with which they wish to share data (e.g., via cookie syncing [270]). Ultimately, if a script wishes to store an identifier in the browser, it will need to call a browser API, and as such, we monitor the flow of information to and from browser APIs. We build a graph representation of the page load by monitoring network requests, JavaScript execution, HTML element creations, and browser storage access. From this graph we extract *flow* features, which explicitly capture distinctive information flows in advertising and tracking. Our evaluation shows that WebGraph's graph representation and flow features can entirely supplant content features, with comparable accuracy.

While high accuracy is necessary for deployment, it is not sufficient. We have repeatedly seen that advertisers and trackers will attempt to circumvent detection and evade blocking [174, 291, 295]. Therefore, in order for an advertising and tracking classifier to be useful in practice, it must be robust to adversarial manipulation. We show that WebGraph represents a significant step forward in robustness to adversarial evasion when compared to previous approaches. In particular, we find that WebGraph is robust to the types of URL, CNAME, and content manipulation evasion techniques that are in use on the web today. We also know that ad and tracking adversaries will attempt to deploy more sophisticated evasion techniques tailored to our classifier. To understand how robust WebGraph would be in the face of these new evasion techniques, we propose a novel realistic graph manipulation evasion technique. We show that this attack achieves only limited evasion success against WebGraph, while incurring a non-trivial usability loss in terms of mistakenly blocking its own advertising/tracking resources or other benign resources on the web page.

Overall, our findings suggest that the community should migrate away from unreliable content features for advertising and tracking blocking. We show that information flow features built upon the actions of advertisers and trackers provide a promising path forward.

In summary, our contributions are as follows:

- We show that existing ML-based ad and tracker detection approaches are susceptible to evasion due to their heavy reliance on content features. As a representative example, we show how an adversary can achieve near-perfect evasion of AdGraph using evasion techniques already in use on the web today.

- We introduce WebGraph, the first ML-based ad and tracker blocking approach that does not rely on content features and captures fundamentally distinctive information flows in advertising and tracking.
- Our in-depth evaluation shows that WebGraph achieves comparable accuracy to prior approaches and achieves significantly better robustness to adversarial manipulation of content features.
- We propose a novel graph manipulation evasion technique, and show that WebGraph (and the information flow features it relies on) remain robust under this sophisticated attack.

Chapter organization: The rest of this chapter is organized as follows: Section 3.2 provides an overview of recent advance in ML-based ad and tracker blocking. Section 3.3 evaluates robustness of existing graph-based approaches, using AdGraph as a representative example. Section 3.4 describes the design and evaluation of WebGraph. Section 3.5 further evaluates WebGraph's robustness to adversarial attacks. We discuss limitations of our work in Section 3.6 and conclude in Section 3.7.

3.2 Background & Related Work

Online behavioral advertising enables ad targeting based on users' interests and behaviors. To target ads, online advertising relies on the intertwined tracking ecosystem that uses cookies for cross-site tracking. For instance, the real-time bidding

(RTB) protocol that powers programmatic online advertising has built-in mechanisms for advertisers and trackers to share information [192, 270]. Thus, almost always, ads and trackers go together, often with intertwined execution flows and resource dependencies. Below, we revisit prior literature on ad and tracker blocking, and analyze its limitations.

Popular ad and tracker blocking tools such as Adblock Plus [5] rely on filter lists [39, 114]. These filter lists are manually curated based on user feedback. Prior work has shown that manually curated filter lists suffer from *scalability* and *robustness* issues. First, filter lists have trouble keeping up with the ever expanding advertising and tracking ecosystem. Filter lists have grown to include tens of thousands of rules that are often not updated in a timely fashion. For instance, prior work showed that filter lists may take as long as 3 months to add rules for newly discovered ads and trackers [215]. Once a filter rule is added to block an advertising and tracking service, it is rarely removed, even if it is no longer needed. In fact, prior work showed that almost 90% of the rules in filter lists are rarely or never used [295]. Second, filter lists are not robust to evasion attempts by advertisers and trackers. Filter lists are brittle in the face of domain rotation [168, 319] and manipulation of page structure [153, 292, 314]. For instance, prior work showed that filter lists are susceptible to evasion attacks such as randomization of URL path, hostname, or element attributes and IDs [140, 308].

Addressing scalability. To address the scalability issues that arise due to manual curation of filter lists, researchers have proposed to use machine learning (ML) for automated ad and tracker blocking. Prior ML-based approaches mainly detect ads and trackers at the network and JavaScript layers of the web stack. Specifically, these approaches detect ads and trackers by featurizing network requests [149, 203, 287] or JavaScript code [209, 225, 316].

Network layer approaches rely on content in URLs, HTTP headers, and request and response payloads (e.g., keywords, query strings, payload size) to extract features and train ML models to detect ads and trackers [149, 203]. While trying to mimic filter lists by detecting ad and tracker URLs, these approaches end up replicating some characteristics of filter lists and thus also naturally inherit their shortcomings. For example, presence of a certain keyword in the request URL could be a distinguishing feature. However, as discussed earlier, such keyword based features are brittle in the face of trivial evasions such as domain rotation [140, 308].

JavaScript layer approaches rely on static or dynamic analysis to extract features and train ML models to detect ads and trackers. Examples of features are n-grams of code statements obtained via static analysis [209] or JavaScript API invocations captured via dynamic analysis [316]. These approaches are susceptible to JavaScript obfuscation [173, 188, 204]. These approaches are also susceptible to evasion such as script amalgamation or dispersion. They implicitly assume that tracking code is bundled in a single script or that tracking scripts only contain tracking code. However, in practice, tracking code could be distributed across several chunks and packaged with functional code [216].

Addressing robustness. While network and JavaScript layer approaches consider information at each layer in isolation, ads and trackers rely on all three layers (i.e. network, JavaScript, and HTML) of the web stack for their execution. Therefore, it is natural that focusing on only one layer lacks robustness against the aforementioned evasion attempts. To address this limitation, graph-based approaches aim to capture the interactions among and across network, JavaScript, and HTML layers of the web stack.

Graph-based approaches extract features from the cross-layer graph representation to train ML models to detect ads and trackers [216, 290]. These approaches leverage rich cross-layer context and thus claim to be robust to evasion attempts. AdGraph was the first graph-based approach to ad and tracker classification [216]. It extracts structural features from the graph such as node connectivity and ancestry information as well as content features such as URL length and presence/absence of certain keywords. Sjösten et al. [290] introduced PageGraph, which extends AdGraph's graph representation by improving event attribution and capturing more behaviors. In addition to content and structural features, they also added perceptual features to train the classifier. Since *perceptual* features attempt to use the rendered resource content, they are also considered content features. Chen et al. [164] proposed an approach, using PageGraph, to detect trackers based on their execution signatures. In contrast to ML-based approaches, their signature-based approach would only be able to detect trackers that strictly match the signatures of tracking scripts, but miss trackers with even slight deviations in their behavior, such as changes in the execution order. Kargaran et al. [227] followed a different approach. Instead of building a graph representation per website, they combined graph representations across multiple websites to model relations between third parties on those sites. Just like AdGraph, they also extract structural and content features from the graph to train the classifier.

These graph-based systems use a combination of content and structural features for classification, which they claim increases the robustness to evasion attacks. While this combination should intuitively improve classifier robustness, we posit that it would be less robust than expected if the classifier relies heavily on content features. This is because content features pertain to a single node on the graph and are easy to manipulate for an adversary, e.g., using adversarial attacks on textual [322] and perceptual [301] content features, without causing undesired changes in other nodes. It is noteworthy that Zhu et al. [322], also manipulate structural features, however their manipulations are only limited to graph size. Further, they do not evaluate the impact of their mutations on overall graph.

In the next section, we analyze the robustness of graph-based ad and tracker detection systems. We focus on AdGraph as it is representative of other graph-based systems that use similar structural and content features.

3.3 AdGraph Robustness

In this section, we analyze AdGraph's robustness by evaluating its accuracy in the face of adversarial content manipulation.

AdGraph is a graph-based machine learning approach that detects ads and trackers based on their structural and content properties. AdGraph instruments the Chromium web browser to capture detailed execution of ads and trackers across the

HTML, JavaScript, and the network layer, and models the interaction among these layers in the form of a graph. Using this graph, AdGraph extracts two categories of features: *content* (information related to individual nodes in the graph, such as URL length and presence of ad/tracking keywords in the URL) and *structure* (information about relationships between nodes, such as connectivity and ancestry information). It uses the extracted features to train a machine learning classifier to detect advertising and tracking resources. The full list of AdGraph features are described in Table 3.4.

Since AdGraph relies on content properties, in addition to structural properties, it is subject to same evasion attacks that succeed against the filter lists-based ad and tracker detection approaches [140, 308].

3.3.1 Threat Model & Attack

Our threat model assumes an adversarial third-party advertiser or tracker embedded on a site, who aims to change the classification of its resources from advertising and tracking services (ATS) to benign resources (Non-ATS) in order to evade detection by ad and tracker blocking tools.

We assume that the adversarial third party has limited cooperation with the first-party publisher. We do not assume full cooperation because the parties are mutually distrusting. The third-party adversary generally does not trust the first-party publisher to serve its advertising and tracking resources via a reverse proxy [20, 218]. Likewise, the first-party publisher does not trust the third-party adversary to host functional resources via the adversary-controlled CDN [151]. Given existing practices, we assume that the adversary can serve its advertising and tracking resources from a first-party subdomain but not arbitrarily within the first-party domain space. For example, the adversary can masquerade its resources through CNAME cloaking [170], which only requires a minor change in DNS records by the first party. Recent measurement studies have reported an increase in the prevalence of CNAME cloaking over the last few years. Dao et al. [174] showed that the usage of CNAME cloaking-based tracking has steadily increased between 2016 and 2020, with 1,762 of Alexa's top-300K websites employing at least one CNAME-based tracker as of January 2020. Dimova et al. [179] also showed that the usage of CNAME cloaking has increased by 22% from 2018 to 2020, with 9.98% of Tranco's top-10K websites now employing at least one CNAME-based tracker as of October 2020.

We assume that the adversary is able to manipulate their own URLs by altering the domain name or query string. Naturally, the adversary can only manipulate URLs that are under their control, and only attempts to manipulate the subset of its URLs that were initially correctly classified as ATS (ad and tracker URLs initially classified as Non-ATS already benefit the adversary). The adversary cannot manipulate the data used to train the classifier. Therefore, we only implement mutations during inference.

We implement two types of URL manipulations. For domain names, we allow the adversary to randomly change the URL's domain, subdomain, or both. In practice, adversaries can rely on automated techniques to generate random domains and subdomains. For example, they can use malware-inspired domain generation algorithms (DGA) techniques to generate a large number of domains [168, 273]. For query strings, we randomly change the number of parameters, the parameter names, the parameter values in the URL, or a combination of the three.

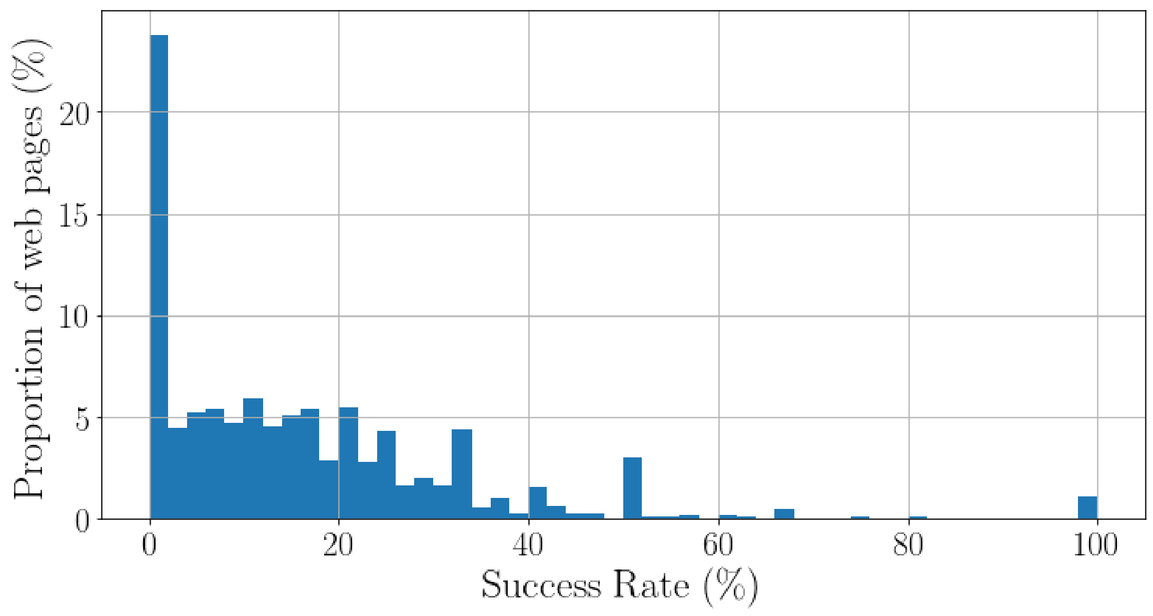


Figure 3.1. Classification switch success rate distribution by web page (over 10 folds) when the adversary does **not** collude with the first party. The average success rate per web page is 15.92 ± 0.03 %.

3.3.2 Results

Experimental setup. We extend OpenWPM [186] to automatically crawl websites with Firefox and build AdGraph's representation. We crawl 10K sites sampled from the Alexa's top-100K list, the top 1K sites and a random sample of 9K sites ranked between 1K-100K, and store their graph representations. Next, we implement a decision tree classifier that closely follows AdGraph's design [216], and extract features from the graphs for training and testing. For ground truth, we use the same set of filter lists for data labeling that were used by AdGraph [216]. A URL is labeled as ATS if it is present in one or more of the filter lists, and Non-ATS otherwise. We use 10-fold cross validation to obtain our results, where the folds are selected such that every fold uses a different set of web pages in the test set. Our classifier obtains comparable performance to the original results reported by [216]: 92.33% accuracy, 88.91% precision, and 92.14% recall. The minor differences are likely due to differences in crawled sites, updated filter lists, and a few subtle changes in our adaptation of AdGraph from online to offline. In AdGraph's online implementation, features are extracted from each node in the graph as they are created. Our offline adaptation, instead, extracts features after page load completion. There are also some minor differences due to JavaScript attribution, caused by the differences in instrumentation between Chromium-based AdGraph and Firefox-based OpenWPM.[1]

Adversarial success rate without collusion. In our first experiment, we assume that the adversary does not collude with the first party. The adversary can randomize their domain and subdomain, but cannot masquerade as the first party. Our content mutation procedure results in the mutation of 41.48 ± 1.47 % of all the test data URLs (averaged over 10 folds). The adversary's success rate in evading the classifier is 8.72 ± 0.42 % (over 10 folds). While this may seem like a low percentage, we note

[1]Due to these differences, our features are not exactly identical to the online implementation of AdGraph. For example, in AdGraph, a node can have a maximum of two parents, which need not be the case for our system. Therefore, we do not use AdGraph features specific to these two parents. The full feature list, showing these differences is provided in Appendix 3.8.1.

that every successful mutation is a win for the adversary since it means that one more of their ads or trackers is now unblocked. Over all 10 folds, the adversary mutated 691,602 URLs, out of which 60,270 had their classifications switched.

We also observe that the evasion success rate varies across sites, as shown in Figure 3.1. For ≈1% of the web pages in the test set (90 pages), the adversary achieves a perfect success rate, meaning that all third-party ads and trackers on the web page are now classified as benign content. It is noteworthy that 21.62% of the unblocked URLs belong to popular ad exchanges, which are responsible for further diffusion of user information due to the broadcast nature of real-time bidding (RTB) [146]. These unblocked ad exchanges can amplify the privacy harm because they often share information about page visits with multiple advertisers and trackers.

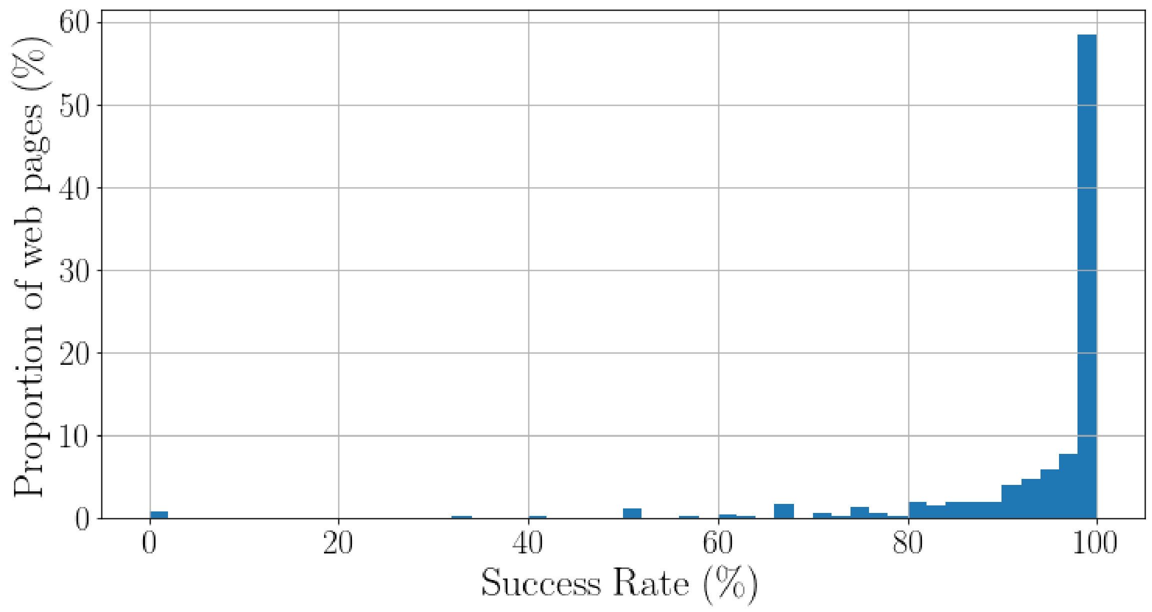


Figure 3.2. Classification switch success rate distribution by web page (over 10 folds) when adversary colludes with the first party. The average success rate per web page is 93.01 ± 0.01 %.

Adversarial success rate with collusion. In our second experiment, we assume that the adversary colludes with the first party. The adversary can perform domain mutation such that their URL is a subdomain of the first party. The adversary's success rate increases to 96.62 ± 0.37 % (over 10 folds). This means that being able to use a first-party subdomain provides almost perfect evasion capabilities. Figure 3.2 shows the evasion success rate variation across sites. For ≈50% of the web pages in the test set, the adversary achieves a perfect success rate. We also see a higher proportion (32.25%) of the unblocked URLs belonging to popular ad exchanges, as compared to the previous experiment.

To better understand why such URL manipulation is able to evade detection by AdGraph, we analyze feature importance using information gain (see Table 3.1). We see that content features are essential to the AdGraph classifier: not only are the top-3 most important features content features, their relative importance scores are also high compared to the other features. Two of the top-3 features depend on whether a URL is third-party, which explains why we obtain high success rates when the adversary has the capability to masquerade as the first party. These two features do not have an effect in the case where the adversary does not collude with the first party, since the adversary cannot change the fact that they are third party. However, the adversary's manipulations still influence the third top feature, length of the URL. Hence, we observe lower but non-trivial success rates even without collusion.

These results show that graph-based classifiers such as AdGraph are vulnerable because of their over-reliance on content features. In the next section, we propose

Feature	**Category**	**Information Gain (%)**
URL length	Content	14.87 ± 0.36
URL domain is a subdomain of the first party	Content	11.06 ± 1.24
URL is a third party	Content	10.67 ± 1.32
Degree of a node	Structure	7.56 ± 0.63
Number of edges divided by number of nodes	Structure	7.48 ± 0.41

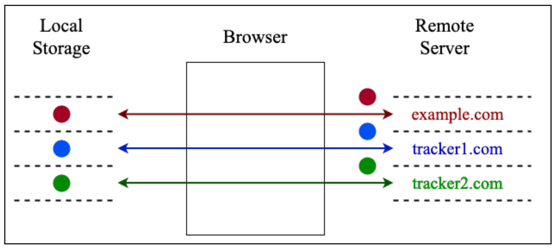
Table 3.1. Top 5 most important features for AdGraph's classification, their category, and information gain values (averaged over 10 folds).

an approach to improve the robustness of graph-based ad and tracker blocking tools.

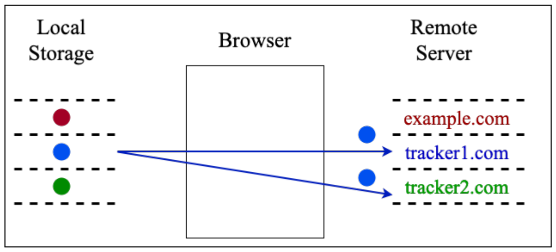
3.4 WebGraph

Online advertising and tracking fundamentally relies on information sharing. Trackers need to share information with each other to improve their coverage of users' browsing history [186, 270]. Trackers also need to share information with each other as part of built-in dependencies in programmatic advertising protocols [136, 268]. We contend that leveraging such fundamental information sharing patterns can help build accurate and robust classifiers for ad and tracker blocking. We introduce WebGraph, a classifier that explicitly captures these information sharing patterns as part of its cross-layer graph representation of the execution of a web page.

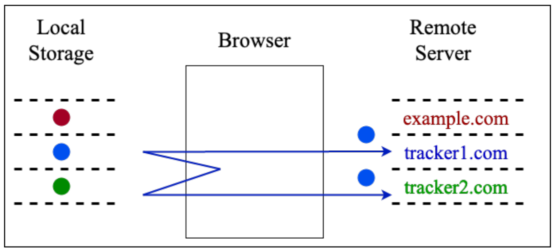
To illustrate the information sharing patterns that we want to capture in WebGraph, let us revisit how information sharing between different origins is mediated by the browser. We deliberately use a loose definition of origin. An origin can be, depending on the specific use case, a site, a domain, or an entity, among others. At a high-level, the web browser isolates different origins, based on various policies, so that their data is not leaked to each other. Figure 3.3a illustrates how the browser limits information sharing between different origins: `example.com`, `tracker1.com`, and `tracker2.com` each have access to their isolated local storage (e.g., cookies, IndexedDB) that may be used to store user identifiers. The browser isolates information flows between the local storage and remote servers of different origins: `tracker1.com` and `tracker2.com` cannot generally access each others' cookies.



(a)



(b)



(c)

Figure 3.3. Origin isolation vs. sharing. Circles represent information about a user gathered by a particular domain (`example.com`, ●; `tracker1.com`, ●; and `tracker2.com`, ●). The box represents the browser which acts as channel between the local storage on the user's device and the remote server of each domain. 3.3a Illustrates origin isolation in the browser: every domain can only access information in their own storage. 3.3b and 3.3c illustrate two information sharing patterns that trackers use to circumvent origin isolation: (b) cookie syncing, where users' identifiers are sent to more than one domain; and (c) sharing identifiers using JavaScript APIs.

Trackers typically circumvent these limitations in the browser in two main ways. First, Figure 3.3b illustrates how a tracker may share its identifier with another tracker through cookie syncing. This can be implemented in several ways. For example, let's say `example.com` loads a JavaScript from Tracker 1 that first uses `document.cookie` to retrieve Tracker 1's identifier cookie from its cookie storage and then initiates a GET request to Tracker 2. The script includes Tracker 1's identifier cookie in the request URL as a query string parameter. Note that the request automatically includes Tracker 2's identifier cookie in the Cookie header. Therefore, when Tracker 2's remote server receives the request, it would be able to sync Tracker 1's identifier with its own identifier. As another example, let's say `example.com` first loads an invisible pixel from Tracker 1, which responds back with a 3XX redirect status code along with the URL in the Location header that points to Tracker 2 and includes Tracker 1's identifier cookie. Upon receiving the response, the browser issues a GET request to Tracker 2 and includes Tracker 1's identifier cookie in the request URL and Tracker 2's identifier cookie in the Cookie header. Again, Tracker 2's remote server is able to sync Tracker 1's identifier with its own identifier.

Second, Figure 3.3c illustrates how a tracker may share its identifier with another tracker through various JavaScript APIs, in several ways. For example, let's say `example.com` loads scripts from Tracker 1 and Tracker 2 which then share their identifiers by reading/writing to the global variables of the `window` object. The script from Tracker 1 may assign its identifier to a new global variable `foo` that is then read by the script from Tracker 2. Therefore, Tracker 1 and Tracker 2's scripts would be able to sync identifiers with each other and also send them to their respective remote servers. As another example, let's say `example.com` loads iframes from Tracker 1 and Tracker 2 which then share their identifiers using `postMessage`. While these iframes have different origins, Tracker 1's iframe can use `window.parent` property to get a reference to the parent window and then use `window.frames` to get a reference to Tracker 2's iframe. Tracker 1's iframe can then use this reference to call `window.postMessage` and send its identifier to Tracker 2's iframe, which can use `window.addEventListener` to receive the identifier. Tracker 2's iframe can then send the shared identifier with its remote server to sync them.

Trackers use a wide variety of information sharing patterns, beyond the two aforementioned mechanisms. A sound and precise examination of all patterns warrants full-blown information flow tracking that adds significant implementation overheads and complexity [163,167,205]. As we discuss next, WebGraph approximately[2] captures these information sharing patterns by including additional nodes and edges in its graph representation that correspond to elements and actions associated with these information sharing patterns. It then extracts new features on this enriched graph representation to train a classifier for detecting ads and trackers.

3.4.1 Design & Implementation

3.4.1.1 Graph Construction

WebGraph captures the flow of information among and across the HTML, network, JavaScript, and storage layers of the web stack. At the HTML layer, WebGraph captures creation and modification of all HTML elements, e.g., `iframe`, that are initiated with scripts. At the JavaScript layer, WebGraph captures the

[2]See Section 3.6 for a discussion of completeness of WebGraph's implementation.

scripts' interaction with other layer, e.g., initiation of a network request. At the network layer, WebGraph captures all outgoing network requests and their responses. At the storage layer, WebGraph captures read/write in cookies and local storage through scripts and network requests, and also the exchange of values between network requests.

OpenWPM Instrumentation. We extend OpenWPM [186] to capture the execution and interaction of HTML, network, JavaScript, and storage layers. To capture HTML elements creation and modifications, we instrument `createElement` method and register a `MutationObserver` interface. To capture network requests, we parse OpenWPM's existing instrumentation, which uses a webRequests listener [3], to capture all of the network requests, their responses, and redirects. To capture JavaScript interaction, we parse OpenWPM's existing instrumentation, which relies on JavaScript's stack trace to log JavaScript execution. To capture read/write to storage, we instrument `document.cookie` and localStorage methods and also intercept cookie read/write HTTP headers.

Graph Composition. Elements at each of the layers are represented with nodes and the interaction between these nodes is represented with edges. Specifically, each HTML element, network request, script, and stored value, is represented as a node. Edges to HTML nodes from script nodes represent the creation and modification of elements. Edges from HTML nodes to network nodes represent initiation of network requests to load content, such as scripts and images. Edges from script nodes

[3]https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest

```
<html>
     <script src='tracker1.com/track.js'>
      ...
      var image =document.createElement('img');
      image.src = 'tracker2.com/sync';
      document.body.appendChild(image);
      ...
     </script>
     ...
     <iframe src='tracker2.com/track.html'>
      <script>
      ...
      idCookie = document.cookie;
      var newReq = new XMLHTTPRequest();
      newReq.open("GET", "tracker3.com?user_id=" + idCookie);
      ...
    </script>
  </iframe>
</html>
```

Code 3.1. An example web page sending requests to several trackers.

to network nodes represent the initiation of `XMLHTTPRequest` which will be parsed by the script. Edges between script and storage nodes and network and storage nodes, represent the read/write of values in the storage. Edges between network nodes either represent redirects or the presence of the same stored values.[4]

Graph Composition Example. To illustrate WEBGRAPH's graph representation, let us consider the example web page given by Code 3.1. The web page embeds a script from Tracker 1 and an iframe from Tracker 2. The tracking iframe from Tracker 2 reads its tracking cookies and sends them to Tracker 3 via an XHR. Both trackers trigger requests to share tracking identifiers. The HTTP requests and responses that result from loads in Code 3.1 are listed in Listing 3.1.

Tracker 1's script embeds an image element from Tracker 2, which causes the browser to send an HTTP request (Request 1 in Listing 3.1) that includes Tracker 2's

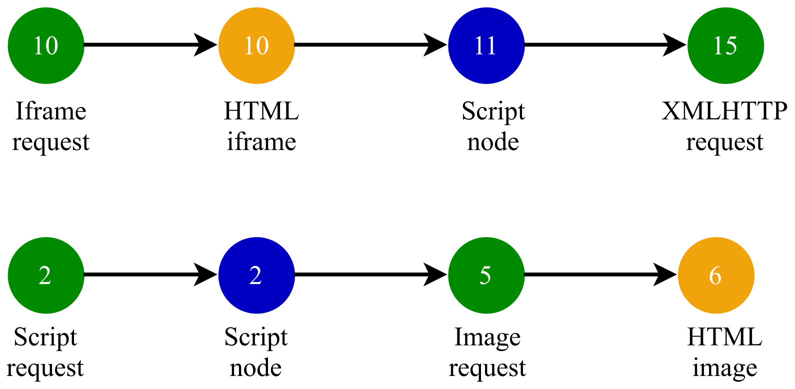
[4]We match stored values with their encoded and hashed counterparts. Specifically, we look for presence of base64 encoded and MD5 and SHA-1 hashed values [184, 192].

```
----------------------------------------------------
Request 1
URL: tracker2.com/sync
Cookie: user1
Response 1
Status: 302
Location: tracker1.com?tracker2_id=user1
----------------------------------------------------
Request 2
URL: tracker1.com?tracker2_id=user1
Response 2
Status: 200
Set-Cookie: userA
Content: pixel.png
----------------------------------------------------
Request 3
URL: tracker3.com?user_id=user1
Response 3
Status: 200
```

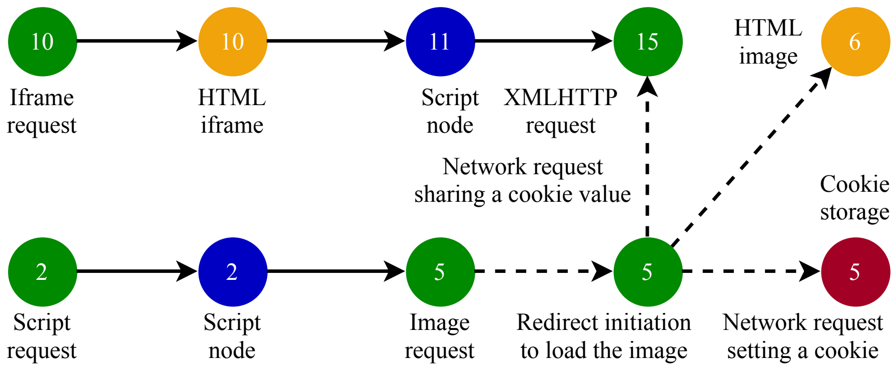
Listing 3.1. HTTP requests and responses initiated from Code 3.1.

cookie. Tracker 2 responds to this request with a redirect to Tracker 1 that embeds the user identifier Tracker 2 received via the initial request's `Cookie` header (i.e., `user1`). The browser makes a subsequent request (Request 2 in Listing 3.1) to Tracker 1. Tracker 1 responds with a tracking pixel image and a `Set-Cookie` header to set its own tracking cookie with the value `userA`. On the backend, Tracker 1 knows that `userA` is known as `user1` by Tracker 2. Tracker 2's embedded iframe further shares its identifier cookie with Tracker 3. It does so by accessing its cookies locally via `document.cookie` and embedding them in an XHR to Tracker 3 (Request 3 in Listing 3.1).

Differences as compared to AdGraph. WebGraph keeps AdGraph's HTML and JavaScript layers as they are, but extends the network layer and includes a new storage layer in the graph representation. WebGraph also introduces information flow edges, which are absent in AdGraph, to entwine the extended network layer and the storage layer. The extension of network and the addition of storage



(a) Graph representation of Code 3.1 in AdGraph



(b) Graph representation of Code 3.1 in WebGraph

Figure 3.4. Graph representation of Code 3.1 in AdGraph and WebGraph. 🟢 represents network nodes, 🔵 represents script nodes, 🟡 represents HTML nodes, and 🔴 represents storage nodes. Node numbers correspond to the lines in Code 3.1. In Figure 3.4b, dotted (- - -) lines represent the additional edges that are captured by WebGraph and missed by AdGraph.

layer allow WebGraph to explicitly capture information sharing patterns used in advertising and tracking.

We illustrate the differences in Figure 3.4 which shows the graph representation of the web page in Code 3.1 and request and response sequences in Listing 3.1 for both AdGraph (Figure 3.4a) and WebGraph (Figure 3.4b). AdGraph's representation of the example web page consists in two disjoint graphs which capture the individual actions of the two trackers: The first row of nodes (from 10 to 15) captures Tracker 2's tracking behavior: from the iframe loading to the initiation of an XHR request. The second row of nodes (from 2 to 6) captures Tracker 1's tracking behavior: from the script loading to the initiation of a network request for loading an image. In this figure, it becomes clear that AdGraph *does not* capture the information sharing pattern between the nodes, because of its inability to capture the redirect (Request 2) made by the image request (network node 5) and the cookie set (storage node 5; visible only in WebGraph's graph) by the redirect request. WebGraph, on the contrary, not only captures the flows appearing in AdGraph, but also captures the redirects (dotted edge between the two network nodes labeled 5) and cookies set by requests (the second network node 5 to storage node 5). This representation further enables WebGraph to link requests that share common identifiers (node 5 to 15).

3.4.1.2 Features

We take the AdGraph feature set and augment them with three categories of features. These additional features come from WebGraph's improved graph representation, i.e., extension of the network layer and a new storage layer. The features target storage, network, and information sharing behaviors that were absent in AdGraph. First, we extract features that measure the number of read/write cookie and localStorage accesses by a node. We obtain these features from the new storage layer. Second, we extract features that measure the number of requests and redirects to/from a node as well as the depth of a node in a redirect chain. These features come from our extension to the network layer. Third, we extract features that measure the number of different types of information sharing edges (e.g., nodes access the same storage node or share data of a storage node) to/from a node. We obtain these features using both the network and storage layers in WebGraph's graph representation. We also extract some standard graph features (e.g., in-degree, out-degree, eccentricity) for the information sharing edges. We jointly refer to these three newly added categories of features as *flow* features. Table 3.4 in Appendix 3.8.1 lists the full set of features in WebGraph, including these newly added flow features.

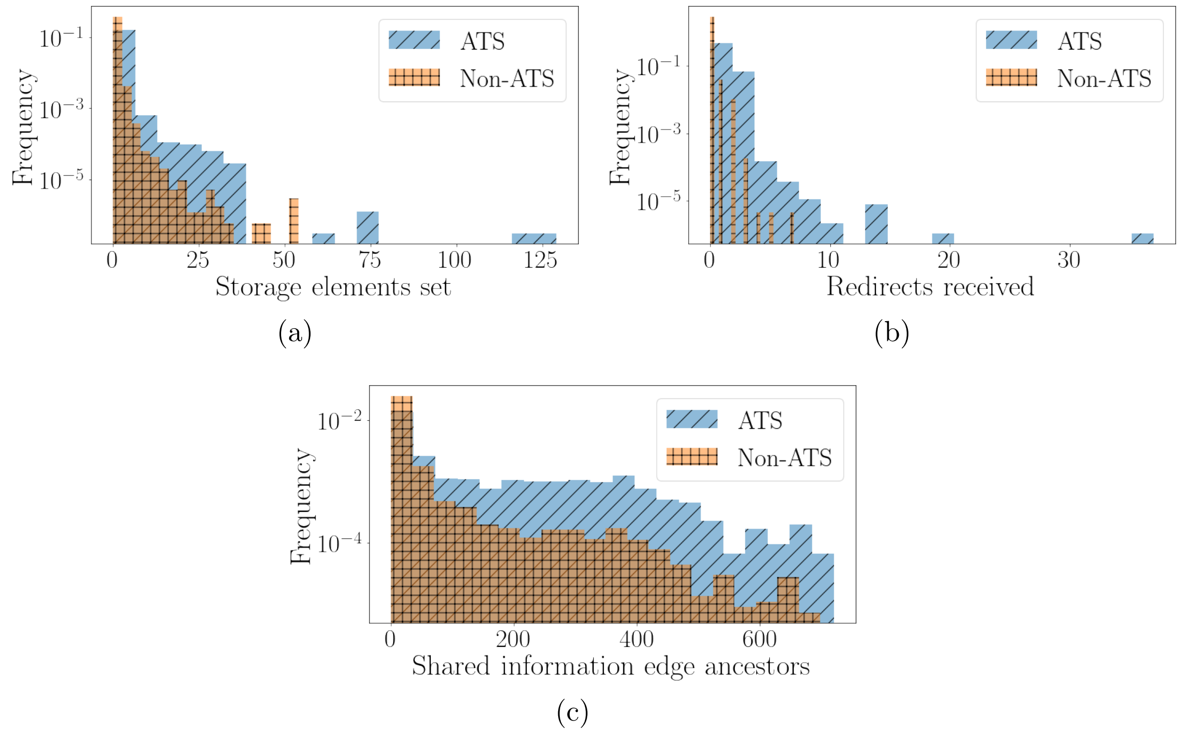


Figure 3.5. Histograms of three example flow features for ATS and Non-ATS resources (normalized, y-axis in log scale). (a) Number of storage elements set by a resource; (b) Number of network redirects received by a resource, and (c) Number of shared information ancestors of a resource. These features demonstrate different distributions for ATS and Non-ATS resources, and thus can help the classifier to distinguish between them.

To illustrate the potential of these features in distinguishing ATS and Non-

ATS resources, let us consider three flow features belonging to each of the categories described above:the number of storage elements set by a resource (Figure 3.5a), the number of requests that were redirected to a resource (Figure 3.6b), and the number of information sharing edge ancestors (Figure 3.6c). As explained in Section 3.4, ATS resources store user identifiers in storage elements and use redirects and sharing of identifiers in URLs to perform actions such as cookie syncing. Therefore, we expect ATS resources to set a larger number of storage elements, be at the receiving end of redirects, and be involved in a larger number of shared information edges than Non-ATS resources. We plot in Figure 3.5 the distributions of these features in our dataset. We see that, indeed, the distributions are different for benign and ATS resources, with ATS presenting higher values on average for the three features under study. The differences in distributions is especially apparent for Figure 3.6c, which shows the number of shared information edge ancestors. In our dataset, we observe 589,218 cases of ATS receiving a cookie value in a request URL, as compared to 89,564 cases for non-ATS. This sharing is detected as an information sharing edge, which in turn leads to ATS having larger values in shared information edge properties than Non-ATS. In the case of redirects (Figure 3.6b), the probability that a Non-ATS resource has more than 7 redirects tends to 0, which is not the case with ATS resources. The number of ATS resources with more than 7 redirects is very small in our dataset ($\approx$ 0.04%). Yet, it is a top-20 feature in our classifier, as observing more than 7 redirects directly identifies the resource as ATS. Storage element setting (Figure 3.5a) shows a similar behavior, with ATS resources sometimes having more than 54 elements set, while Non-ATS resources never have so many.

While individual contributions of some of these flow features might be small, they provide a strong signal in distinguishing ATS when combined, as we show in the next section.

3.4.2 Evaluation

To evaluate WebGraph, we use the same dataset of 10K web pages and method as in Section 3.3.2. To understand the marginal benefit of WebGraph over AdGraph, we systematically compare the performance of different feature sets and graph representations. Table 3.2 summarizes the results.

Graph	**Feature Set**	**Accuracy**	**Precision**	**Recall**
AdGraph	Structural + Content	92.33 ± 0.50	88.91 ± 1.14	92.14 ± 0.65
	Structural	80.22 ± 0.81	71.85 ± 1.53	82.44 ± 1.26
WebGraph	Structural+ Flow + Content	94.32 ± 0.27	92.24 ± 0.67	94.14 ± 0.30
	Structural + Flow	86.93 ± 0.64	80.57 ± 1.12	90.01 ± 0.50
	Structural	82.62 ± 0.47	75.67 ± 0.75	85.09 ± 1.41

Table 3.2. Evaluation of WebGraph and AdGraph with different feature set variations.

We observe that AdGraph's performance drops by at least 10% when content features are removed. Recall from Section 3.3.2 that if content features are present alongside structural features, AdGraph is particularly susceptible to evasion: trackers have an 8.72% evasion success rate on their own, and a 96.62% success rate if they collude with the first party. Thus, there is a trade-off in AdGraph between effectiveness (with content) and robustness to evasion (without content).

Second, Table 3.2 shows that WebGraph's performance is better than AdGraph due to its improved graph representation and new flow features. When using all feature sets, WebGraph outperforms AdGraph by about 2-4%. If we remove content features for robustness, we observe a drop in accuracy limited to just 4-9% across all measures. We conclude that WebGraph's improved graph representation and new flow features can compensate for the loss of content features to a large extent.

Finally, Table 3.2 shows that WebGraph's improved graph representation by itself (i.e., even without the new flow features) contributes to about half of the improvement over AdGraph. WebGraph with only structural features achieves 2-4% improvement across all measures as compared to AdGraph also with only structural features. We conclude that, while WebGraph's new flow features help improve its accuracy, the improved graph representation is an important contributor to performance.

Feature	**Category**	**Information gain (%)**
Shared information ancestors	Flow	6.48 ± 0.69
Number of requests sent by node	Flow	5.9 ± 0.69
Number of nodes in graph	Structure	5.46 ± 0.35
Average degree connectivity of node	Structure	5.18 ± 0.16
Number of edges in graph	Structure	4.19 ± 0.34

Table 3.3. Top-5 most important features for WebGraph's classification, their category, and information gain (averaged over 10 folds).

To provide insights into the relative importance of flow and structural features, we list top five most important features in terms of information gain in Table 3.3. The two most important features are flow features. As discussed in Section 3.4.1.2, the top feature distribution (Figure 3.6c) is very different for ATS and non-ATS, so it's not surprising that this feature contributes to the classification. Storage setting (Figure 3.5a) and received redirects (Figure 3.6b) contribute a smaller, but still useful, portion towards identification; they have information gains of 1.9% (± 0.37) and 2.5% (± 0.47) respectively (21st and 17th most important features). We also observe that structure features, enhanced by WebGraph's improved graph representation, also contribute towards the performance. We further analyze which features contribute most to each prediction of WebGraph using `treeinterpreter` [92]. For ≈ 32% of predicted ATS in the dataset, the flow features were the top contributors, indicating that they provide an important signal for the presence of trackers. In contrast, for ≈ 47% of predicted Non-ATS in the dataset, structure features were the top contributors. These results confirm our earlier intuition that capturing information sharing behaviors that are unique to advertising and tracking carries significant predictive power.

3.4.3 Efficiency

We envision WebGraph to be used for filter list curation and maintenance in an offline setting. WebGraph relies on large scale web crawls and notoriously expensive graph traversals for feature extraction. We now measure WebGraph's offline overhead to demonstrate its adequacy as a tool to periodically update filter lists.

Crawl time. Our implementation of WebGraph has an upper bound of 60 seconds, enforced with a timeout, to crawl a website. In the average case, crawls take only ∼26.46 seconds. Crawls can be parallelized over several instances to reduce the crawl time. For example, it took us around 10.5 hours to crawl 10K websites, parallelized over 7 instances. Without parallelization and if all websites would reach the timeout, the crawls would take ∼166 hours.

Processing websites. On average, WebGraph takes 0.72 seconds to build the graph, 15 seconds to extract features, and 0.25 seconds to train and test each website. For our crawl of 10K websites, it took us a total of ∼44.36 hours to create their graphs and extract features on a single instance. This time can be significantly reduced using parallelization.

Update frequency. These estimates suggest that for 10K websites containing ∼1.1 million requests WebGraph will require, at most, ∼166 (data crawling) and ∼44.36 (data processing) hours with a single instance. However, when averaged over 7 instances, the computation time significantly reduces to only 16.83 hours (10.5 for crawling and 6.33 for processing). We anticipate the computation time for periodic updates to reduce significantly because many websites have low update frequency. Specifically, monitoring the update frequency of websites will allow us to only crawl when changes are expected in websites. In cases, where we determine that the website did not change since the last crawl, we will not recompute their classifications. With this performance, WebGraph could be able to update filter lists on a daily basis, and certainly operate within the current the expiry period (mandated update frequency) of popular filter lists, e.g., 4 days for Easylist [114]. Frequent updates with WebGraph can help remove outdated rules and as well as add new rules to block newly discovered ads and trackers.

3.5 WebGraph Robustness

In this section, we evaluate WebGraph's robustness against content mutation attacks (described in Section 3.3) and structure mutation attacks.

3.5.1 Content mutation attacks

To evaluate WebGraph against content mutations, we strengthen the threat model described in Section 3.3 to enable the adversary to also masquerade their resources as first party, i.e., through first-party subdomains. Overall, our attacks involve random mutations to domain names, subdomains, and the query string in URLs (Section 3.3.2).

By relying on content mutations, the adversary is able to switch 96.62% of their `ATS` resources to `Non-ATS` against AdGraph. Against WebGraph, the adversary's success rate plummets to just 8.34 $\pm$ 0.66% (over 10 folds). For example, `mylives-ignature.com`, a tracking domain, was able to switch all of its 560 `ATS` resources to `Non-ATS` against AdGraph, but none against WebGraph.

Note that, even though WebGraph does not use content features, the evasion success rate against WebGraph does not drop to zero. This is because some of the WebGraph's features implicitly rely on URL properties. For example, shared information edges, that consider sharing of cookie values via query strings in the URL, are affected by URLs manipulations.

3.5.2 Structure mutation attacks

Next, we evaluate WebGraph's robustness against structure mutations. We assume that the adversarial third-party has unrestricted black box access to the WebGraph's classifier, i.e., the adversary can make unlimited queries and observe WebGraph's classification output. This access enables the adversary to validate the effect of their structure mutations.

Attack details. We assume that the adversary can mutate the structure of a web page through resource addition, re-routing, and obfuscation. Moreover, we assume that the adversary also performs content mutations, to maximize its chance of success Resource addition entails addition of new resources, such as images and scripts. Resource re-routing entails re-organization of existing redirect chains, i.e., dispersing a redirect chain in a sequence of `XMLHttpRequest's` through one or multiple scripts. Resource obfuscation entails obfuscation of cookie or query string parameter values of existing resources, i.e., encoding or encrypting cookie or query string parameter values in a format that is not detected by WebGraph's implementation, before sharing them in network requests. To remain stealthy, we assume that the adversary does not delete functional content from the web page that could damage usability.

It is important to note that even simple mutations, such as adding a single element to the web page, can significantly change graph properties and impact several features. For example, the addition of a child node causes a cascading effect. It increases the number of descendants of all the parent nodes in the branch, all the way up to the root node, and also impacts their centrality. Thus, the result of such simple mutations can become unpredictable and hard to control by the adversary: It can cause unintended classification changes for nodes under and outside the control of the adversary. Complex mutations, such as adding a combination of nodes at once, further complicate having control on the number of unintended classification changes. In our evaluation, we only consider atomic mutations, i.e., addition, re-routing, or obfuscation of individual resources.

Mutation algorithm. We capture the adversary's unrestricted black box access to classifier by implementing a greedy random algorithm to find suitable mutations. This kind of algorithm is extensively used in the literature due to its simplicity and practicality [207, 309, 323]. The algorithm (formally described in Appendix 3.8.4) iteratively mutates WebGraph's graph representation. At each step, it adds, re-routes, or obfuscates the resource that provides the best trade-off between *desired* (`ATS` to `Non-ATS`) and *undesired* (`NON-ATS` to `ATS`) classification switches. Resource addition is simulated by adding nodes to a randomly selected leaf nodes in the graph. Resource re-routing is simulated by adding each request, in a redirect chain, as an individual node to one or more randomly selected scripts. Resource obfuscation is simulated by replacing stored values in URLs with an encoding that is not detected by WebGraph.

3.5.3 Empirical evaluation

Experimental Setup. To evaluate WebGraph's robustness, we must rebuild the graph and recompute the features after each mutation. To keep the evaluation time reasonable, we sample 100 web pages from our dataset, and we limit the graph growth to 20%. To ensure that this sample is representative of our dataset, we divide graphs into 5 bins according to their size and sample 20 web pages from each bin. We only consider web pages that have 250 or fewer nodes (i.e., 80% of the dataset; see Appendix 3.8.2 for the full distribution). We exclude large web pages to avoid exceptionally long evaluation times. For each web page, we designate the adversary as the third party with the highest number of resources classified as `ATS`. It is noteworthy that the adversary with the highest number of `ATS` resources has an opportunity to do maximum damage.

In this dataset, the median evaluation time per web page was 29.08 minutes, with 39% of the pages taking more than an hour to run. Even though this is a simulation, the computational cost is directly proportional to the operational cost for the adversary. The adversary must consume additional CPU cycles and memory and in the case of node addition, send additional network requests, thereby increasing the cost of their attack.

Success metrics. To measure adversary's success, we define the following terms:

$\texttt{ATS}_{\texttt{Web}}$: Number of nodes classified as `ATS`.

$\texttt{ATS}_{\texttt{Adv}}$: Number of adversary nodes classified as `ATS`.

$\texttt{Non-ATS}_{\texttt{Web}}$: Number of nodes classified as `Non-ATS`.

$\texttt{Non-ATS}_{\texttt{Adv}}$: Number of adversary nodes classified as `Non-ATS`.

`desired`: Number of nodes switching from $\texttt{ATS}_{\texttt{Adv}}$ to $\texttt{Non-ATS}_{\texttt{Adv}}$.

`undesired`: Number of nodes switching from $\texttt{Non-ATS}_{\texttt{Web}}$ to $\texttt{ATS}_{\texttt{Adv}}$.

`neutral`: Number of nodes switching from `ATS` to `Non-ATS` for non-adversary nodes.

Success rate: Desired changes from the adversary's point of view. It is calculated as $\texttt{desired}/\texttt{ATS}_{\texttt{Adv}}$.

Collateral damage: Undesired changes from the adversary's point of view. It is calculated as $\texttt{undesired}/(\texttt{Non-ATS}_{\texttt{Adv}} + \texttt{Non-ATS}_{\texttt{Web}})$.

Other changes: Non-consequential changes from the adversary's point of view. It is calculated as $\texttt{neutral}/\texttt{ATS}_{\texttt{Web}}$.

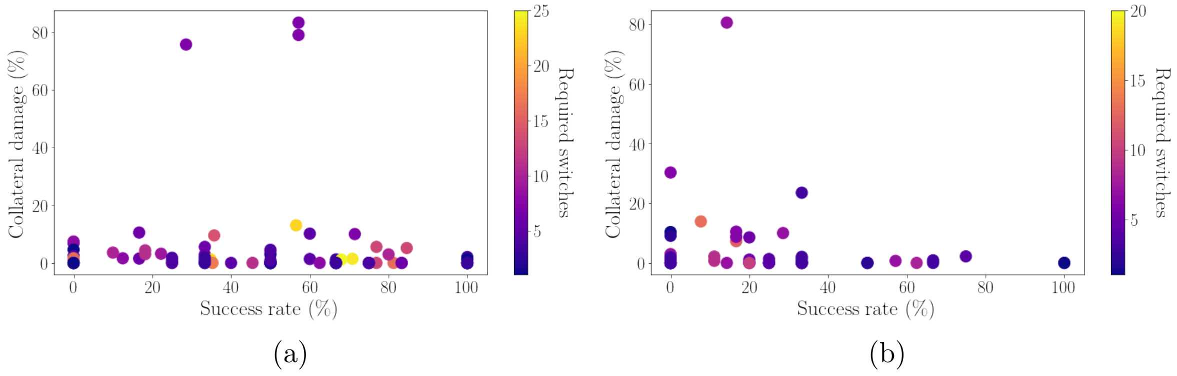
We illustrate the node switches, with the mutation algorithm, for an example graph in Appendix 3.8.5.

3.5.3.1 Adversary's success

We assume that the adversary neither colludes with other third parties nor with the first party and can only perform mutations on the nodes and edges it controls. We conduct the attack on 100 web pages. We note that increasing the number of graph mutations increases the adversary's mean success rate from 38.6 $\pm$ 33.01 (median: 33.33) at 5% graph growth to 52.48 $\pm$ 33.4 (median: 50.00) at 20% graph growth. The classification switches lead to a decrease in the overall classification accuracy by 1.5%, recall by 8.85%, and precision by 2.29%.

However, the adversary's success comes at a cost of collateral damage. The average collateral damage rises from 2.17 $\pm$ 11.19 to 3.88 $\pm$ 13.55 (median: 0). In

Figure 3.6, we illustrate the trade-off between success rate and collateral damage at 20% graph growth. The x-axis represents success rate, the y-axis represents collateral damage, and circles represent a trade-off between the two. The circles' color represents $\texttt{ATS}_{\texttt{Adv}}$ or the number of classifications the adversary has to switch for the particular web page. The lighter the color, the more switches are required, i.e, the cost of success increases. For the web pages in this dataset, the adversary has, on average, $\texttt{ATS}_{\texttt{Adv}} = 5.98 \pm 5.39$ nodes classified as $\texttt{ATS}$. For certain pages, the $\texttt{ATS}_{\texttt{Adv}}$ can be as high as 26.



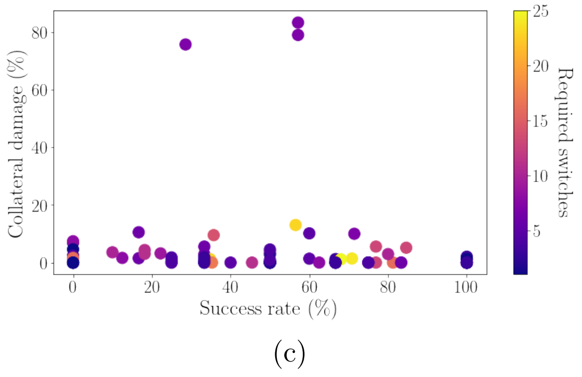


Figure 3.6. Adversary's success rate vs. collateral damage for each web page in the test data at 20% graph growth. Figure 3.6a represents all mutations, 3.6b represents only structure mutations, and 3.6c represents only resource re-routing and obfuscation mutations. Colored circles represent the number of required switches.

Ideally, the adversary wants to be at the bottom right of the graph, where it achieves 100% success rate with zero collateral damage. The adversary is able to reach its ideal target on only 13 web pages, which only required four switches. The adversary is able to achieve 50% or more success on 61 of the tested web pages. Together, they amount to 240 nodes switched, with 45 of these pages having non-zero collateral damage. On the other hand, we have 9 web pages that had a higher collateral damage than success rate: a net negative effect of the mutation. Out of these, 6 web pages had 0% success rate with non-zero collateral damage, and 3 web pages had a large collateral damage $> 75\%$ (with one web page hitting 83%).

Overall, our evaluation shows that even in the case of an unrealistic adversary that has the capabilities to manipulate structure features at will, and also the operational power to do so for a large number of iterations, there is no guarantee of perfect success.

Breakage. If undesired changes affect benign resources that are essential to the correct functioning of the web page, even a small collateral damage can break the page. This may have large impact on trackers. If users leave the broken web pages, the adversary cannot track them or show them ads.

We define website breakage as degradation in usability of the website. We say there is `major` breakage if the user is unable to complete the primary functionality of the web pages (e.g. login, search or page navigation). If the user is unable to complete a secondary functionality of the web pages (e.g. comment or review), we consider that there is `minor` breakage. Otherwise, we consider that the web page does not have any breakage.

We quantify breakage on all of the 21 web pages where the adversary experiences undesired classification switches.[5] We open these web pages side by side on stock Firefox and a Firefox configured with an extension that blocks the URLs that switched classification, and we compare them side by side to identify any visual signs of breakage.

We ask two reviewers to perform the analysis. Our reviewers attain an agreement of 90.46% in their evaluation. They find that the undesired classification switches cause `major` breakage on 3 and `minor` breakage on 2 web pages. This breakage mostly happens when the first-party resources are switched from `Non-ATS` → `ATS`.

Careless adversary. If the adversary is not concerned with changes to any non-adversarial nodes, their collateral damage decreases. The adversary still does not want their own content to be blocked, so it will optimize against their own nodes switching to `ATS`. This change in strategy updates the collateral damage calculation to: $\texttt{undesired}/\texttt{Non-ATS}_{\texttt{Adv}}$.

As per our modified definition, the web pages on which all of the adversary nodes are classified as `ATS`, there can be no collateral damage; we note 55 such web pages. For the remaining 45 web pages, where the adversary can experience collateral damage, the mean growth in success rate does not change much from the previous

[5]In total, the adversary experiences undesired classification switches on 45 web pages. However, 24 web pages no longer serve the switched `ATS` resources.

scenario, but naturally the trade-off is better. Further, out of 45 web pages, only 8 web pages have collateral damage, as compared to 27 web pages that had collateral damage as per our original definition. Out of these 8 web pages, 4 had a higher collateral damage than success rate (net negative effect), and 6 web pages have a large collateral damage $> 20\%$ (with 2 web pages hitting 100%). Thus, even when an adversary is not concerned about collateral damage to other parties they are not significantly more successful in subverting WebGraph.

Collusion with the first party. So far, we have assumed that the adversary is a single third party that does not collude with other third parties or the first party. If we assume the adversary colludes with both, the adversary can add child nodes to *any* node in the graph. This is a much stronger adversary than in Section 3.5.3.1, where in each iteration the adversary can only test a random subset of the options. Realistically, such a powerful collusion would be difficult to implement, as it would require coordination and cooperation among multiple parties to ensure that the mutation is feasible.

We repeat our experiment, but we now allow the adversary to consider all possible mutation options on any node, and pick the best one in each iteration. These experiments take longer to run (see Appendix 3.8.3), so we only analyze 100 web pages whose graphs have at most 50 nodes. We see that collusion enables the adversary to have a slightly higher success rate (63 pages with success rate $> 50\%$ as compared to 60 for the non-colluding adversary) and lower collateral damage (9 pages with damage $>0\%$ compared to 18 pages for a non-colluding adversary). These results are described in detail in Appendix 3.8.6).

3.5.3.2 Impact of mutation choice

Next, we evaluate the adversary's preference in selecting the most useful mutations. We notice that the adversary picks resource addition 81.70%, resource re-routing 17.26%, and resource obfuscation 0.04% of the time. Resource obfuscation is rarely chosen by the adversary because the graph already has content manipulations applied, and these manipulations have already severed many of the edges that would be severed by resource obfuscation. To separate out the impact of different mutations, we conduct two additional experiments: (1) where the adversary can only perform resource addition, and (2) where the adversary can only perform resource re-routing and obfuscation.

We exclude 33 of the web pages for experiment 2 because these web pages do not have re-route-able or obfuscate-able resources. For the remaining 67 pages, we see that the re-routing/obfuscation mutations (Figure 3.6c) are more effective than addition mutations (Figure 3.6b). Re-routing/obfuscation not only yields higher success rates for the adversary, but also results in lower collateral damage. This is unsurprising because these mutations target information sharing patterns which are distinctive of trackers; changing these patterns removes an important signal for the classifier (see Table 3.3).

However, in practice, resource re-routing and obfuscation would entail high costs for the adversary since they involve the manipulation of identifier sharing patterns. Specifically, the adversary would have to coordinate with other parties on changes to these patterns, and redesign how they perform tracking in order to perform these mutations. The success of these mutations also depends on the degree to which flows are captured by the instrumentation used to create the graph. WebGraph's instrumentation approximates information flows and will not capture all attempts by an adversary to use re-routing and obfuscation. We argue that this is not a fundamental flaw in WebGraph's architecture but a limitation in our implementation that approximates information flow (Section 3.4). A fully fledged instrumentation would make these manipulations much more difficult to deploy. See Section 3.6 for an extended discussion. Resource addition has fewer costs for the adversary since it does not involve coordination with additional parties. This manipulation is not affected by the type of implementation because it is not related to the flow of identifiers.

3.5.3.3 Comparison with AdGraph.

We also evaluate whether WebGraph, in addition to having superior classification performance, offers robustness benefits over AdGraph. For this comparison, we only use AdGraph's structural features, as we already demonstrated that content features are not robust. Because AdGraph does not have features based on flow information, we only perform resource addition. We find that the adversary has greater success against AdGraph than WebGraph, but also suffers from more collateral damage (Figures 3.13 and 3.14 in Appendix 3.8.6). This is because the structural effects of node additions are hard to control, as explained in Section 3.5.2. Since the former is beneficial to the adversary but the latter is not, it is not clear-cut as to whether one system provides more robustness than the other.

In summary, our results indicate that mutations to the structure of the graph are harder for an adversary to control than content mutations. It is not trivial for an adversary to produce the desired classification switches for their resources without producing any undesired changes. This makes WebGraph, without content features, more robust to adversarial evasion attacks than prior approaches. Within the structural mutations, re-routing/obfuscating resources target information flow and are a more effective strategy than adding resources. At the same time, performing these mutations is not trivial for the adversary since they involve coordination with multiple parties.

3.6 Limitations

In this section, we discuss limitations of WebGraph's design, implementation, and evaluation.

Completeness. For efficiency reasons, WebGraph focuses on a limited subset of the browser's API surface, such as HTTP cookie headers, `document.cookie`, and `window.localStorage`. WebGraph's implementation is also geared towards capturing client-side information that is pertinent to stateful tracking. However, techniques used by ATS need not to be limited to these APIs or to stateful tracking. Some ATS have started to use stateless tracking techniques, such as browser fingerprinting, which use APIs that are not currently covered by our instrumentation [175, 214, 249]. To account for these techniques, WebGraph's instrumentation must be extended to include the corresponding APIs.

WebGraph's manually designed graph representation and feature set capture the most well-known information sharing patterns. The limits of these approach are shown in Section 3.5.3.2, where we show that an adversary capable of hiding or obfuscating traditional sharing flows has a better chance to bypass WebGraph than doing structure modifications. This limitation is, however, linked to our implementation choices. To increase WebGraph's coverage of sharing behaviors, if suffices with increase the instrumentation to cover more information flows. Ideally, we would instrument full-blown information flow tracking. Such expansion would incur prohibitive runtime overheads (up to 100X-1000X [205]) and its complexity makes it hard to integrate in the browser [163,167,239,298]. Nevertheless, the design of WebGraph permits that the instrumentation to be upgraded gradually, as ATS evolve in response to our evasion protection techniques, increasing the cost of evasion without fundamentally changing the detection approach.

Robustness analysis. Inspired by previous work on graph-based detection evasion [207,309,323], we use a greedy algorithm to attack WebGraph. This algorithm only considers the best mutation in each iteration, and not the best overall mutation. Thus, it is not guaranteed to find the optimal mutation sequence that would lead to the best adversary performance. We note however that, as our experiments on small websites show, even exhaustive search does not lead to perfect success. We expect adversaries to try alternative algorithms to improve their success rates. However, any alternative that is close to exhaustive search will become prohibitively expensive for the adversary when the web page graph is large.

Another option for the adversary would be to perform more sophisticated graph mutations instead of the simple node additions that we perform. An adversary could tailor their mutations to the page's graph structure by studying how their node changes affect the graph properties of the web page. However, this requires that the rest of the graph (i.e., the portions outside of the adversary's control) remaining unchanged. Realistically, it would be difficult for an adversary to coordinate with other parties to generate these changes.

Finally, we note that the dynamism of modern websites [158] complicates the process for the adversary. Web pages change often, sometimes on every load. Even if the adversary manages to find an appropriate set of mutations, those mutations may be invalid the next time the page is reloaded.

3.7 Conclusion

In this chapter, we showed that state-of-the-art ad and tracker blocking approaches are susceptible to evasion due to their reliance on easy-to-manipulate content features. We then showed that information sharing patterns in online advertising and tracking can instead be leveraged for robust blocking. Specifically, our proposed WebGraph builds a cross-layer graph representation to capture such information flows and train a machine learning classifier for accurate and robust ad and tracker blocking. Our results showed that it is non-trivial to evade WebGraph's classifier without causing unavoidable collateral damage.

While it is not foolproof, we believe that WebGraph raises the bar for advertisers and trackers attempting to evade detection. We foresee that advertising and tracking services would need to significantly re-architecture their information sharing patterns to achieve long-lasting evasion against WebGraph. We note, however, that introducing new information flows may be quite complicated, as they may require collaboration among the first-party and numerous third-parties on a typical web page.

3.8 Appendix

3.8.1 Comparison between AdGraph and WebGraph features

Table 3.4 compares and contrasts the features used in WebGraph and AdGraph. WebGraph does not use content features. Graph size, Degree and Centrality features come under both structure and flow categories, since they include graph properties that are based on both normal (structure feature) and shared information edges (flow feature). WebGraph uses both types of edges, whereas AdGraph uses only normal edges. Some structural features used in AdGraph are not used in WebGraph due to WebGraph being adapted for offline use, whereas the features are useful in an online context.

3.8.2 Distribution of graph sizes

Figure 3.7 shows the distribution of number of nodes in the graph representations of the web pages in our dataset. Since 80% of web pages have 250 nodes or fewer, we sample from this subset in our structural mutation experiments in Section 3.5.

3.8.3 Experimental run times

Figures 3.8 and 3.9 describe the run times for the experiment described in Section 3.5.3.1 (adversary without collusion). Figure 3.8 shows the impact of graph size on each iteration of the experiment. As expected, smaller graphs have lower run

Table 3.4. WebGraph features comparison with AdGraph. ● indicates that a feature is present. WebGraph calculates Graph size, Degree and Centrality features using both normal and shared information edges. The former comes under structural features while the latter comes under flow features.

Feature	**Type**	**WebGraph**	**AdGraph**
Request type (e.g. iframe, image)	Content		●
Ad keywords in request (e.g. banner, sponsor)	Content		●
Ad or screen dimensions in URL	Content		●
Valid query string parameters	Content		●
Length of URL	Content		●
Domain party	Content		●
Sub-domain check	Content		●
Base domain in query string	Content		●
Semi-colon in query string	Content		●
Graph size (# of nodes, # of edges, and nodes/edge ratio)	Structure	●	●
Degree (in, out, in+out, and average degree connectivity)	Structure	●	●
Centrality (closeness centrality, eccentricity)	Structure	●	
Number of siblings (node and parents)	Structure		●
Modifications by scripts (node and parents)	Structure		●
Parent's attributes	Structure		●
Parent degree (in, out, in+out, and average degree connectivity)	Structure		●
Sibling's attributes	Structure		●
Ascendant's attributes	Structure	●	●
Descendant of a script	Structure	●	●
Ascendant's script properties	Structure	●	●
Parent is an eval script	Structure	●	●
Local storage access (# of sets, # of gets)	Flow (storage)	●	
Cookie access (# of sets, # of gets)	Flow (storage)	●	
Requests (sent, received)	Flow (network)	●	
Redirects (sent, received, depth in chain)	Flow (network)	●	
Common access to the same storage node	Flow (shared information)	●	
Sharing of a storage node's value in a URL	Flow (shared information)	●	
Graph size (# of nodes, # of edges, and nodes/edge ratio)	Flow (shared information)	●	
Degree (in, out, in+out, and average degree connectivity)	Flow (shared information)	●	
Centrality (closeness centrality, eccentricity)	Flow (shared information)	●	

times since features have to be calculated over a smaller number of nodes. Note that graph size is not the only contributing factor to run times. Other factors such as the complexity of the structure and flow behaviors would also contribute towards time spent in each iteration, which is why we observe variations in iteration time among graphs of the same size. We see that the mean time per iteration can be as high as $\approx$ 1200 seconds (median is $\approx$ 68 seconds). Figure 3.9 shows the total experiment time over all iterations for a graph. Since we increase the sizes of graphs by 20% of their

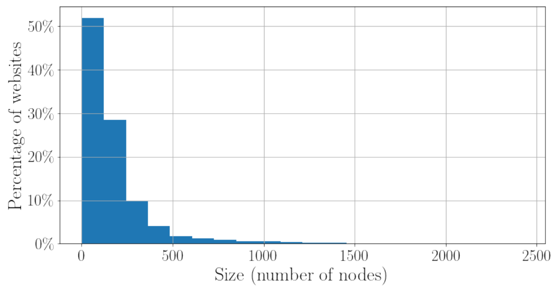


Figure 3.7. Distribution of number of nodes in the graph representations of the web pages in the dataset. 80% of the web pages have 250 nodes or fewer.

original size, bigger graphs will have a larger number of iterations. In our dataset, the maximum time taken for an experiment is 46654.19 seconds, the minimum is 15.67 seconds, and the median is 1745.11 seconds. 39% of the graphs in our dataset have a run time of more than an hour.

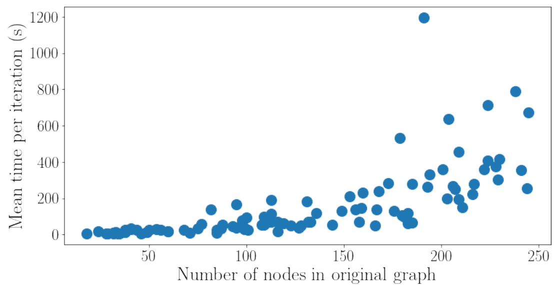


Figure 3.8. Mean time per iteration vs graph size for the experiment without collusion. Standard deviation over all the iterations for each graph was less than 2%. Larger graph sizes take longer time for each iteration.

Figure 3.10 shows the total experiment time over all iterations for the experiment described in Section 3.5.3.1 (collusion with first party). The median time is 265.03 seconds, with the maximum time going up to 992.67 seconds, despite the

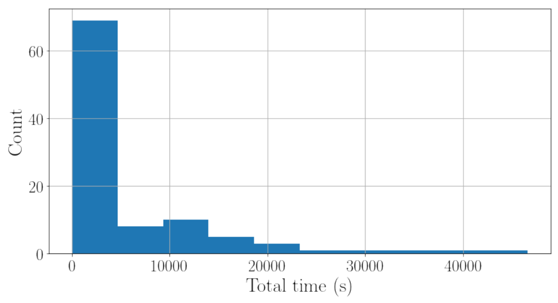


Figure 3.9. Distribution of total run time for the experiments in Section 3.5.3.1. 39% of the graphs in our dataset have a run time of more than an hour.

maximum graph size being only 50 nodes. In comparison, for the adversary without collusion, for graph sizes up to 50 nodes, the median is 21.46 seconds and the maximum is 221.51 seconds. Since the adversary considers all nodes in the graph as potential parents, each iteration takes a longer amount of time.

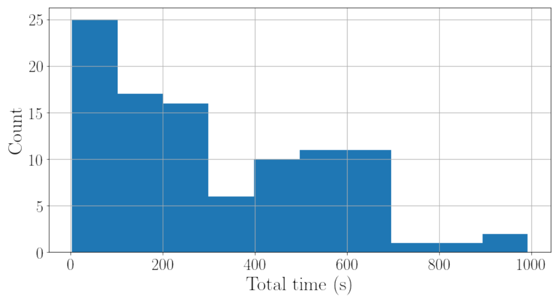


Figure 3.10. Distribution of total run time for the experiments in Section 3.5.3.1. The experiment time can be as high as 996.67 seconds for a graph size <= 50 nodes.

3.8.4 Graph Mutation algorithm

In each iteration, the algorithm mutates WebGraph's graph representation and probes the model for classification decisions. The algorithm takes the following inputs: a graph representation of a web page, G_0, consisting of all the nodes in the graph; a set of nodes and edges T of size l_T, representing the resources loaded by the adversary, hereafter referred to as the adversary resources; a trained classifier M that identifies ATS in WebGraph; and a maximum number of iterations that the algorithm can run, `max_iter`.

Algorithm 3.1 Greedy random graph mutation. G_0 is a web page representation, T is the set of l_T nodes and edges controlled by the adversary, M is a trained model, and `max_iter` is the maximum number of operations.

```
Input: $G_0, T, C, M, \texttt{max\_iter}$
 1: for v $\in G_0$ do
 2:     $\texttt{x}_{G_0} \leftarrow \texttt{ExtractFeatures}(v)\ \forall\ v \in G_0$
 3:     $\texttt{y}_{G_0} \leftarrow \texttt{Classify}(M, \texttt{x})\ \forall\ x\ in\ \texttt{x}_{G_0}$
 4: end for
 5: $G \leftarrow G_0$
 6: $i \leftarrow 0$
 7: $\texttt{graph-info} = []$
 8: while $i < \texttt{max\_iter}$ do
 9:     for $t \in T$ do
10:         $G_t \leftarrow \texttt{MutateGraph}(G, t)$
11:         $\texttt{x}_t \leftarrow \texttt{ExtractFeatures}(v)\ \forall\ v \in G_t$
12:         $\texttt{y}_t \leftarrow \texttt{Classify}(M, \texttt{x})\ \forall\ x\ in\ \texttt{x}_t$
13:         $\texttt{d}, \texttt{u} \leftarrow \texttt{GetDesiredAndUndesired}(\texttt{y}_t, \texttt{y}_{G_0})$
14:         $\Delta_t = \texttt{d} - \texttt{u}$
15:         $\texttt{graph-info}[t] \leftarrow (\Delta_t, t, G_t)$
16:     end for
17:     $G \leftarrow G_t$ in $\texttt{graph-info}$[t] with largest $\Delta_t$
18:     $T \leftarrow \texttt{UpdateAdv}(T, t \in \texttt{graph-info}[t])$
19:     $T \leftarrow \texttt{sample}(T, l_T)$
20:     $i \leftarrow i + 1$
21: end while
```

The algorithm processes the input as follows: It first uses the classifier M to obtain classifications of all nodes in the original graph G_0 (lines 1–4 in Algorithm 3.1). Second, it iterates over the steps from lines 9–20 `max_iter` times. In each iteration, every adversary node tries resource addition, resource re-routing, and obfuscation, and produces a new mutated graph, G_i (line 11). Third, it extracts features from the mutated graph G_i and uses them to classify all the nodes in this graph (lines 11–12). Fourth, it compares the predictions in the original and mutated graphs to obtain the number of desired and undesired switches (line 13). We assume an adversarial goal for which *desired* switches are all those in which an adversary node is switched from `ATS` to `Non-ATS`, whereas *undesired* switches are all those where any `Non-ATS` node is switched to `ATS` node. We call the total number of adversarial `ATS` nodes whose prediction the adversary wishes to change to `Non-ATS` the number of *required* switches. The switching of nodes not under the adversary's control from `ATS` to `Non-ATS` do not affect the adversary. These switches are, therefore, neither desired nor undesired. Finally, the adversary chooses the mutation that provides the best result, i.e., the one with the best trade-off between desired and undesired switches (lines 14–15). The adversary updates its T based on the chosen mutation (line 18). To keep memory and run time manageable, at the end of every iteration the algorithm randomly samples l_T adversarial nodes and edges from T (line 19) to be considered in the next iteration.

Mutation example. An example iteration of the greedy random algorithm, using resource addition as the mutation, is shown in Figure 3.11. The algorithm proposes two mutations (right) to the initial graph (left), and chooses the mutation that provides the best trade-off between the desired and undesired classification switches. Figure 3.11 also illustrates that even a simple node addition may lead to unintended changes in classification decisions: the adversary may change the classification decisions for some nodes from `ATS` to `Non-ATS`, but as a side effect nodes previously classified as `Non-ATS` are now classified as `ATS`.

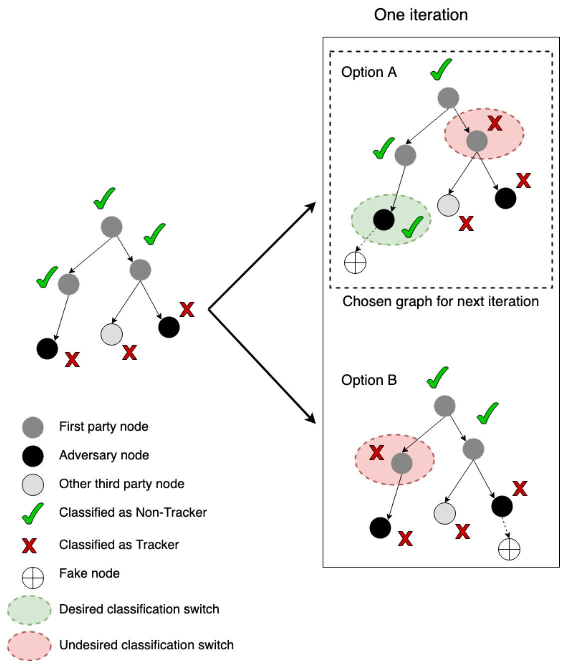

One iteration
Option A
Chosen graph for next iteration
Option B
First party node
Adversary node
Other third party node
Classified as Non-Tracker
Classified as Tracker
Fake node
Desired classification switch
Undesired classification switch

Figure 3.11. One iteration of the greedy random mutation algorithm. In this iteration the algorithm selected two adversarial nodes to add a child (Option A: bottom left node, top; Option B: bottom right node, top;). In option A, the change leads to one desired (green circle) and one undesired (red circle) modification. In option B, the change only causes one undesired modification. The algorithm would pick the graph with best trade-off between desired and undesired switches for the next iteration: i.e., Option A.

3.8.5 Mutations on a single web page.

To illustrate how mutations result in classification switches, we take as an example a web page in which the third party with the highest number of `ATS` resources is `assets.wogaa.sg`, which has 12 nodes in the graph. Figure 3.12 shows the breakdown of classification switches as the adversary mutates the graph using the greedy mutation algorithm. The $\texttt{ATS}_{\texttt{Adv}}$ or the number of classifications the adversary wants to switch is 5 (pink line —). From the adversary's point of view, adversarial nodes switching from `ATS` $\rightarrow$ `Non-ATS` are desired (blue line -●-), whereas adversarial nodes switching from `Non-ATS` $\rightarrow$ `ATS` are undesired (orange line -▲-). We consider `Non-ATS` $\rightarrow$ `ATS` changes on non-adversarial nodes to be undesired because they may have unintended impact on the web page (red line -◆- and brown line -■-). For instance, a first party `Non-ATS` $\rightarrow$ `ATS` switch may break the web page. We note that, if the adversary's goal is to just create a denial of service and force the user to disable ad and tracker blocking, the adversary might be unconcerned about breakage. In our experiments, switches that do not affect the adversary, such as `ATS` $\rightarrow$ `Non-ATS` for non-adversary nodes, are neither considered desired nor undesired (purple line -★- and green line -⬟-).

There are two points worth highlighting from Figure 3.12: (1) Even if an adversary achieves the maximum number of desired switches, the mutations may produce undesirable changes, to both the adversary's nodes and others. For instance, at 20% of growth, 3 of the adversary's `ATS` nodes are classified as `Non-ATS`, but also 7 `Non-ATS` nodes (3 adversary and 4 non-adversary) switch to the undesired `ATS` classification.

(2) The evolution of desired and undesired switches is not monotonic, i.e. the classification of nodes may change in both directions as the adversary mutates the graph, resulting in increasing or decreasing counts. This finding reinforces our argument that it can be cumbersome for an adversary to create targeted structural mutations without any unintended consequences. Not only it is hard to predict how mutations will affect adversary's own desired classification, but also how those mutations may result in undesirable changes to others.

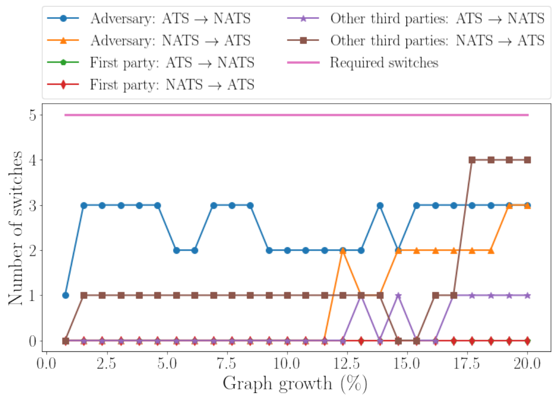


Figure 3.12. Example breakdown of classification switches for the adversary's and other nodes on the graph. `NATS` is shorthand for `Non-ATS`. $\texttt{ATS}_{\texttt{Adv}} = 5$ (pink line), $\texttt{Non-ATS}_{\texttt{Adv}} = 7$, $\texttt{ATS}_{\texttt{Web}} = 62$, $\texttt{ATS}_{\texttt{Web}} = 13$ (not shown in plot). At 20% growth, the adversary achieves 3 **`desired`** switches, 7 **`undesired`** switches and 1 **`neutral`** switch. This leads to a success rate of 60%, a collateral damage of 10.14% and other changes of 7.7%.

3.8.6 WebGraph robustness experiments

We show the success rate vs. collateral damage plots for the experiments described in Sections 3.5.3.1 and 3.5.3.3.

Figures 3.13 and 3.14 show the results for an adversary that performs only resource addition against AdGraph (with only structural features) and WebGraph respectively. AdGraph shows a higher number of successes for the adversary (44 pages with success rate $> 50\%$ as compared to 30 for WebGraph). At the same time, AdGraph also shows a higher amount of collateral damage (which is not beneficial for the adversary) – 66 pages with non-zero collateral damage, as compared to 47 for WebGraph. Hence, there is no clear-cut winner between the two classifiers in terms of robustness. However, we do see that AdGraph has lower successes and higher collateral damage than WebGraph against the powerful adversary that can do all mutations as shown in Figure 3.6a (note that this adversary cannot be used against AdGraph since AdGraph does not use information flow edges), since this adversary targets the effective, but costly, information sharing patterns.

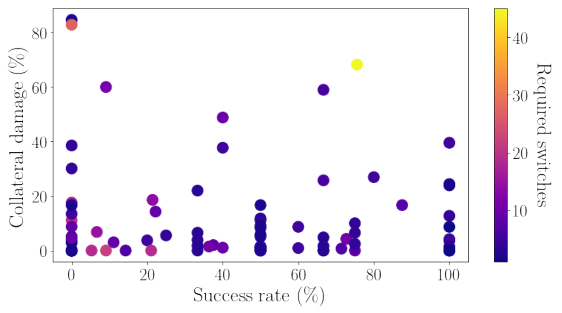


Figure 3.13. Adversary's success rate vs. collateral damage for each web page in the test data at 20% graph growth, for resource addition against WebGraph. Color denotes the number of required switches.

Figures 3.15 and 3.16 show the results for an adversary that colludes against an adversary with no collusion (Section 3.5.3.1). A colluding adversary shows a higher

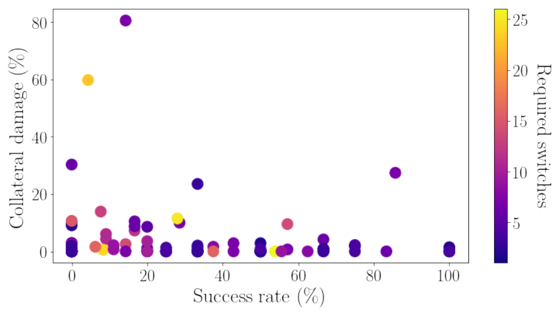


Figure 3.14. Adversary's success rate vs. collateral damage for each web page in the test data at up to 20% graph growth, for resource addition against WebGraph. Color denotes the number of required switches.

number of successes (63 pages with success rate > 50% as compared to 60 for the non-colluding adversary), and a lower collateral damage (9 pages with damage >0% compared to 18 pages for a non-colluding adversary).

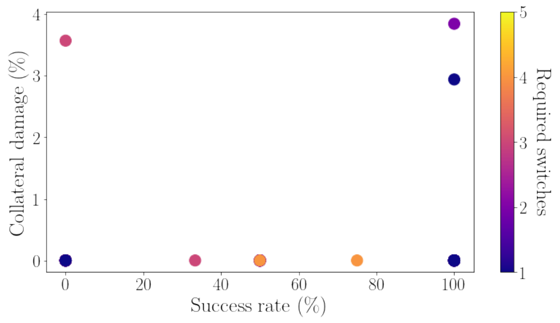


Figure 3.15. Adversary's success rate vs. collateral damage for each web page in the test data at 20% graph growth, for a colluding adversary. Color denotes the number of required switches.

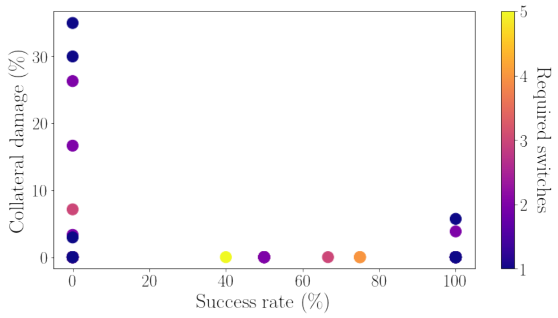


Figure 3.16. Adversary's success rate vs. collateral damage for each web page in the test data at up to 20% graph growth, a non-colluding adversary. Color denotes the number of required switches.

CHAPTER 4

Khaleesi: BREAKER OF ADVERTISING & TRACKING REQUEST CHAINS

4.1 Introduction

Request chains, most commonly implemented using HTTP redirects, enable several important web functionalities such as URL shortening [142], protocol upgrades [160], CDN request routing [148], etc. They have also been used by advertisers and trackers to implement cookie syncing as part of programmatic online advertising [135, 145, 186, 192, 268, 270]. As mainstream browsers have recently implemented countermeasures against third-party cookies [48, 50], advertisers and trackers have increasingly used request chains to circumvent these privacy protections. For example, request chains are being used to implement bounce tracking [86,220,222] — a tracking technique that advertisers and trackers use to circumvent third-party cookie blocking by forcing users to visit them in the first-party context. Request chains have also been used to generate HTTP Strict Transport Security (HSTS) super cookies [79, 221].

While malicious use of request chains for drive-by malware download, spam, and phishing has been extensively studied [156,165,196,210,238,246,254], the research community has only just started to look into their recent increased use by advertisers and trackers and comprehensive countermeasures are lacking [230]. The state-of-the-art approaches to detecting and blocking advertising and tracking resources are generally limited to analyzing requests individually. A slew of heuristic and machine learning (ML) approaches have been proposed to analyze information in HTTP request headers and payloads to detect advertising and tracking requests [149,203,287]. These approaches cannot effectively detect advertising and tracking request chains since they lack the necessary context to do so. Several approaches target cookie syncing enabled by request chains using information in HTTP request and response headers and payloads [145, 186, 192, 270]. These approaches narrowly target request chains implementing a specific behavior such as cookie syncing and cannot detect other behaviors such as bounce tracking.

Browsers have deployed defenses against specific abuses of request chains [195, 253, 313]. Brave's AdGraph/PageGraph [216, 296] and Safari's ITP [310] aim to capture some context but they are not geared towards detecting all potential abuses of request chains. While AdGraph/PageGraph analyzes fine-grained JavaScript execution behavior during the page load, it does not specifically capture the sequential linkability of requests in a chain. Furthermore, capturing such fine-grained JavaScript execution behavior incurs significant performance overhead that limits its suitability for online deployment. ITP considers the appearance of a request in a chain but not what it actually does. Thus, since ITP does not capture the available sequential context, it is prone to misclassify benign request chains limiting its utility to cookie blocking rather than outright request blocking.

We propose KHALEESI, a machine learning approach that focuses on sequential context to detect advertising and tracking request chains. Specifically, we design a light-weight representation of request chains to capture the sequence of requests and responses. This representation allows us to effectively capture the interactions between chains of interrelated requests triggered by HTTP redirects or embedded scripts. We leverage this purpose-built representation to extract features that capture this context and train our classifier. Khaleesi's classifier makes a new classification decision in an online fashion as new request in a chain is loaded. This capitalization of sequential context enables Khaleesi to more accurately and efficiently detect advertising and tracking request chains than the state-of-the-art ad and tracker blocking approaches.

We evaluate Khaleesi across various browser configurations on top-10K websites. The results show that Khaleesi classifies request chains with an accuracy ranging from 98.63–99.95%. Khaleesi outperforms prior approaches by 3.51–40.07% in terms of accuracy. Khaleesi is also generally more robust against evasion attempts such as domain rotation, URL randomization, and CNAME cloaking. Khaleesi's accuracy holds well over time, degrading less than 5% after 8 months. Moreover, Khaleesi loads webpages faster on 59.82% of the websites as compared to Adblock Plus and it is 2.2× faster than AdGraph.

Our analysis of Khaleesi's findings sheds light on the information sharing ecosystem enabled by request chains. We build a request chain graph to understand bilateral information sharing relationships between different entities and find that despite individual request blocking, through ad and tracker blocking extensions, trackers can still share information with each other via request chains. Our findings show that Khaleesi's improved accuracy helps significantly reduce the proliferation of tracking in the request chain graph. We also find emerging use cases of request chains to implement bounce tracking for circumventing recently introduced restrictions on third-party cookies.

We summarize our key contributions as follows:

1. An **ML approach that capitalizes on sequential context** to detect advertising and tracking request chains.

2. A **rigorous evaluation of accuracy and robustness** in detecting advertising and tracking request chains.

3. A **performant implementation** that is feasible for online deployment.

4. An **analysis of information sharing and circumvention** by advertisers and trackers through request chains.

Chapter Organization This rest of the chapter is organized as follows. Section 4.2 presents an overview of request chains and describes limitations of prior work. Section 4.3 describes the design and implementation of Khaleesi. Section 4.4 presents the evaluation of Khaleesi's accuracy. Section 4.5 discusses Khaleesi's runtime performance. Section 4.6 analyzes Khaleesi's findings to shed light into information sharing and circumvention use cases of request chains. Section 4.7 concludes the chapter.

4.2 Background & Related Work

4.2.1 Background

A URL redirect (short for redirection) is a standard technique to forward a user's request for a resource to another address. Redirects form a *chain* of interrelated requests that are often used to share data between trackers. In a typical HTTP request, the browser will send a request to a server for a resource at a specific location. The server evaluates the request and responds with the requested resource if it is available. However, if the server cannot fulfill the request on its own then it may forward the browser to another server. To this end, the server responds with an HTTP 3XX response code and includes a new URL for the resource in the `Location` response header. The browser then issues a subsequent HTTP request to the new URL and the same process ensues. Subsequent request-response sequences form a *request chain.*

Request chains can be implemented at multiple layers in the web stack: via HTTP redirects, HTML redirects, and in JavaScript.[1] *HTTP-layer request chains* are implemented as part of the HTTP protocol using 3XX response codes. In this type of request chain, a browser will be bounced between locations on one or more servers to retrieve content for a single resource. *HTML-layer request chains* are implemented through *meta refreshes.* This type of request chain occurs when the HTML of a page includes a `meta` tag that specifies `http-equiv="Refresh"` with a new URL in the `url` attribute. Because they rely on HTML, meta refresh redirection can only

[1]DNS CNAME records also provide a form of redirection at the DNS level, but we do not study these.

occur at the frame level. *JavaScript-layer request chains* can be implemented in a number of ways. Request chains can be built in the top-level frame when a script updates `window.location` to a new URL. Request chains can be built from embedded resources when a script loads a chain of resources with intertwined dependencies [210]. A recent measurement study sampled sites across the Alexa top 1M and found that almost 80% of them use HTTP redirects and 35% use JavaScript-based top-level redirects [281].

Request chains are most often built from redirects. Redirects serve many necessary functions on the web: they allow websites to seamlessly migrate content between hostnames, upgrade connections to more secure protocols (i.e., HTTP to HTTPS), and implement URL shortening services. However, redirects are also often used for abuse. Most notably, redirects are used to obscure URLs that distribute spam or other types of malware [254]. Request chains also play a core role in online advertising: they allow advertisers and trackers to share tracking information across origins [135, 145, 268, 271].

Cookie Syncing. Advertisers most often track users across the web using client-side identifiers stored in cookies. When advertisers wish to collaborate, they first need to "sync" these identifiers with each other. Advertisers cannot directly share these identifiers with each other due to the same-origin policy. To bypass this restriction, advertisers embed their identifiers in requests to other advertisers. Such information sharing between different origins is called *cookie syncing* [135, 192, 268]. Prior work has shown that some tracking origins sync cookies with 100+ other origins

[186], and that syncing enhances their coverage by as much as a $7\times$ [271].

Bounce Tracking. To combat cookie based cross-site tracking, several browsers such as Safari and Firefox now block third-party cookies [30, 310].[2] Advertisers have been found to bypass third-party cookie blocking by "bouncing" the browser through a tracking website during an otherwise unrelated top-level navigation (i.e., by using a top-level redirect) [222, 253]. When embedded as a third-party, the bounce tracker has no access to cookies. However, once a tracker is visited directly during a "bounce" it can read and write cookies as a first-party. This allows it to associate tracking data with identifiers stored in the users cookies. A recent study showed that as many as 87% of popular websites might be bypassing third-party cookie restrictions using such techniques [281].

HSTS Tracking. Request chains have also been used to exploit HTTP Strict Transport Security (HSTS) to create user identifiers [79]. HSTS provides a way for servers to direct the browser to always load resources from a specific hostname over HTTPS, even when a website specifies HTTP. A request to a hostname that has HSTS set will automatically upgrade HTTPS before the request is ever sent to the network. A tracker can abuse this functionality by setting HSTS on a random subset of hosts it controls—each hostname allows the tracker to store one bit of information. Thus, a unique N-bit value can be created by redirecting the user to N hostnames who specify to be loaded over HTTPS or HTTP. To identify users, the trackers can

[2]Safari blocks all third-party cookies and Firefox blocks third-party cookies from known trackers.

regenerate the N bit value by requesting N tracking pixels, where secure requests load different tracking pixels as compared to the insecure requests (e.g. 1x1 vs 1x2). A tracking script can recognize the different images and can regenerate the N bit value by assigning 1 to HTTPS and 0 to HTTP.

4.2.2 Related Work

There are no purpose-built countermeasures against advertising and tracking request chains. Existing countermeasures generally operate at the level of individual requests even if they are occurring as part of a request chain. These countermeasures do not fully take into account the available context of request chains. Based on the context they leverage, existing countermeasures can be divided into three categories: (1) approaches that only use request information, (2) approaches that use both request and response information, and (3) approaches that use both request and response information in a sequential manner.

Request based detection. Content blockers, such as Adblock Plus [5], are the most commonly used defense against advertising and tracking. Content blockers rely on filter lists (e.g., EasyList [114]). Prior research has tried to use machine learning (ML) on request information to enhance filter lists. For example, Bhagavatula et al. [149] extracted features from the URL (e.g., query string parameters) to train different ML classifiers for detecting advertising requests. However, filter lists ultimately rely on a subset of information available in HTTP requests: the request URL, the content type of the requested resource, and the top level domain of page. Thus, filter lists cannot detect if trackers engage in cookie syncing because they do not look at the HTTP `Cookie` header. Filter lists do not use sequential context that reflect the *intent* of the request occurring in a chain. The lack of consideration of sequential context also contributes to their susceptibility to adversarial evasion [140, 168, 295].

Request and response based detection. Some approaches [76, 203, 318] have tried to utilize both request and response information to detect ads and trackers using ML classifiers. Yu et al. [318] proposed to detect trackers by observing the unique values shared by a significant number of third parties. Privacy Badger [76] labels third-party domains as trackers if they set cookies on three or more websites. Gugelmann et al. [203] proposed to use HTTP request and response meta data, such as size of requests and whether cookies are set, to train their classifier that detects ads and trackers. Other approaches [186, 192, 271] have been purpose-built to detect cookie syncing in request chains. Papadopoulos et al. [271] proposed to detect cookie syncing events using an ML classifier by relying on keywords in HTTP requests, such as domain name and URL parameters, as features. Both Fouad et al. [192] and Englehardt and Narayanan [186] proposed to detect cookie syncing by observing identifiers across consecutive requests and responses. Though these approaches perform better than the request based detection approaches, they are far from ideal because they are only able to observe a subset of the advertising and tracking behaviors that occur in a request chain. For example, these approaches cannot observe bounce tracking because it requires an analysis of multiple request and response pairs in a sequential manner. Similar to request based approaches, request and response based approaches are also susceptible to adversarial evasion [140, 168, 295] because of their reliance on domain and parameter names (as we demonstrate later in § 4.4.2.2).

Request and response sequence based detection. Recent ML-based approaches have proposed using sequential context to detect ads and trackers, including Safari's Intelligent Tracking Prevention (ITP) [310] and Brave's AdGraph/-PageGraph [216, 290]. ITP uses an ML classifier that detects third-party trackers by monitoring the redirects from third-parties to other domains, the presence of third parties on other websites as a resource, and the presence of third parties on other websites as `iframes` [224]. Though ITP utilizes partial sequential information, by monitoring the number of redirects, it does not capitalize on tracking information revealed in the requests and responses. Furthermore, ITP is a "post-hoc" approach to tracker detection, i.e,. ITP is only able to detect tracking after it has observed the tracking behavior. More recently, AdGraph/PageGraph [216, 290] use a graph-based ML approach to detect ads and trackers by monitoring their interactions across HTML, HTTP, and JavaScript layers. While they implicitly utilize sequential context that is obtained through structural properties of graphs, they do not explicitly capture advertising and tracking behavior via request chains. Because of their partial reliance on sequential context, both ITP and AdGraph/PageGraph cannot precisely detect cookie syncing as it requires to analyze the flow of cookies in a sequence of request and response pairs.

In conclusion, existing approaches do not fully consider the necessary sequential context and thus cannot effectively detect advertising and tracking request chains. To bridge that gap, we next propose KHALEESI, an ML approach that captures the sequential context in request chains to effectively detect advertising and tracking request chains.

4.3 Khaleesi

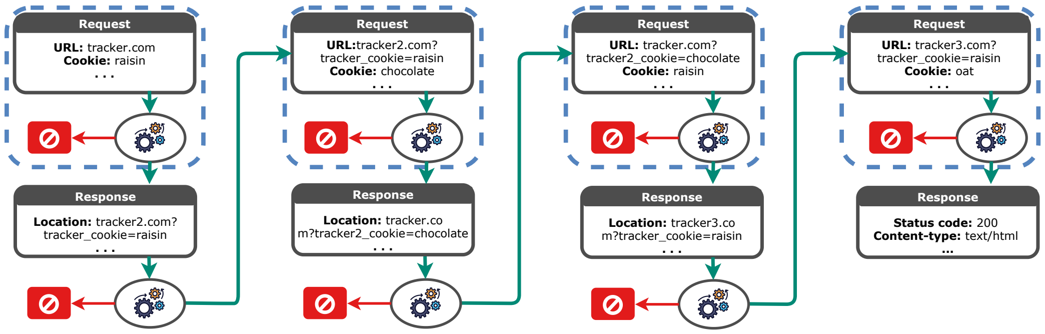

Request
URL: tracker.com
Cookie: raisin
...
Response
Location: tracker2.com?tracker_cookie=raisin
...
Request
URL:tracker2.com?tracker_cookie=raisin
Cookie: chocolate
...
Response
Location: tracker.com?tracker2_cookie=chocolate
...
Request
URL: tracker.com?tracker2_cookie=chocolate
Cookie: raisin
...
Response
Location: tracker3.com?tracker_cookie=raisin
...
Request
URL: tracker3.com?tracker_cookie=raisin
Cookie: oat
...
Response
Status code: 200
Content-type: text/html
...

Figure 4.1. An example of a cookie syncing request chain implemented with HTTP redirects. The dotted blue blocks represent the context utilized by existing approaches in isolation. Whereas the green arrows represent the increasing sequential context utilized by Khaleesi to detect ads and trackers.

In this section, we present the design and implementation of Khaleesi, a machine learning based approach that uses sequential context for early detection of advertising and tracking request chains. At a high level, Khaleesi organizes requests into a chain-like structure which captures the sequential context of interrelated requests. It then leverages this sequential context to train a machine learning classifier to effectively detect advertising and tracking resources.

4.3.1 Motivation & Key Idea

Existing approaches detect individual advertising and tracking requests in isolation, even when they are part of a request chain. They utilize only partial request information, which might indicate what the resource is (i.e., a *tracker*) but not what the resource does (i.e., *tracking*). Such partial consideration of request information not only makes existing approaches less accurate, but is also susceptible to evasion attempts by advertisers and trackers. Prior research has shown that the URL-based ad and tracker detection approaches are vulnerable to hostname and URL path randomization [140, 308]. There have been several instances in the wild, where determined advertisers and trackers have used domain generation algorithms (DGA), to rotate domains, to evade ad and tracker blocking [168, 319]. Since redirects provide an apparatus to load resources from different endpoints at the runtime, evasions attempts (e.g., domain rotation) are even more applicable.

However, the sequentially chained nature of many advertising and tracking requests puts us in an advantageous position to detect them. For example, we can easily observe cookie syncing—a fundamental component of advertisement-related tracking—by analyzing the sequential chain of requests. KHALEESI is the first privacy-enhancing blocking approach that leverages the sequential context available in request chains for early detection. To demonstrate the benefit of the sequential context leveraged by KHALEESI, we show an example of a cookie syncing request chain implemented using HTTP redirects in Figure 4.1. The dotted blue blocks show the visibility of existing request-based detection approaches and the green arrows show KHALEESI's visibility of sequential context. Existing approaches will use incomplete request and response information and will fail to link the redirects in a sequence. Thus, these approaches will miss the fact that the resources are cookie syncing. In contrast, Khaleesi operates with a much richer context: it includes information from requests, responses, and their sequential connectivity. This allows Khaleesi to make a classification decision that incorporates aspects of cookie syncing visible only when multiple request-response pairs are examined.

4.3.2 Request Chain Construction

Khaleesi is able to represent both network-layer and JavaScript-layer request sequences.

Network-layer request chains are constructed by linking HTTP requests that instruct the browser to initiate a redirect. Redirects are continuously linked until the response specifies that the request is complete. We also include Javascript-initiated top-level frame redirections in this category. Top-level redirects can be triggered in a number of ways, including by an HTTP 3XX response status or by HTTP response that includes embedded JavaScript to automatically navigate the frame to a new URL (e.g., via `window.location`). Regardless of how the redirects are triggered, the end result is the same: the server sending the HTTP response forwards the browser to another server.

Network-layer request chains provide decentralized control because each server in the chain may choose to redirect the browser to a new location. Decentralized control is preferred in use cases where each entity wants to control the navigation flow. For example, cookie syncing is a use case where decentralized control is preferred because each entity decides who they want to sync cookies with.

JavaScript-layer request chains are constructed by linking together JavaScript-initiated HTTP requests initiated by the same script. Specifically, we intercept JavaScript stack trace each time a request is initiated and associate the request to the script at the top of the stack. However, not all HTTP requests that originate from a script are interrelated. For example, tag management scripts will initiate a bunch of unrelated HTTP requests. We filter out unrelated requests by only linking requests that share identifiers with each other. Below we describe our requests linking method:

1. Tokenize the values stored in Cookie and Set-Cookie headers, query string parameters from requests and referrers, and the values of non-standard HTTP headers. Values are split on any character other than `a-zA-Z0-9_=-`.

2. Filter out the tokens that have fewer than 8 characters to prevent false matches with common identifiers, such as `en-US`.

3. Compute the Base64 encoded, MD5 hashed, and SHA-1 hashed versions of the tokens.

4. Consider plain, Base64 encoded, MD5 hashed, and SHA-1 hashed versions of the tokens as identifiers.

5. Match the identifiers from the preceding request with the cookies, query string parameters, and non-standard HTTP request headers of all future requests.

6. If there is a match, we consider the request a part of the chain.

In principle, subsequent JavaScript-layer requests achieve the same objective as HTTP redirects and they have been linked into chains in prior work [210]. In contrast with network-layer request chains, JavaScript request chains provide centralized control because the requests are solely determined by the originating script without any intervention from external web servers. Centralized control is preferred in use cases where only a single central entity wants to control the navigation flow. For example, header bidding is a use case where centralized control is preferred because a single script at the client side conducts the bidding process.

4.3.3 Feature Extraction

Next, we utilize sequential context to extract features that distinguish advertising and tracking request chains from functional request chains. These features are designed based on our domain knowledge and expert intuition. We utilize sequential context to extract three types of features: (1) sequential, (2) response, and (3) request. Sequential features capture chain-level properties whereas request and response capture individual HTTP request-level properties, as a whole providing complete sequential context. In total we extract 29 features from the request chains. Table 4.1 lists the features grouped by each category. Below we describe some of the key features in each category. We define remaining features in Appendix 4.9.

Sequential features capture communications that reveal the collaboration between domains. For example, a chain in which several domains redirect to each other may indicate that it is an advertising and tracking redirect chain where domains are trying to share information and identifiers. KHALEESI captures these properties

Sequential Features
Consecutive requests to the same domain
Number of unique domains in the chain
Length of the chain
Probability of the previous prediction
Average probability of the previous predictions
Response Features
Status code
ETag in response header
P3P in response header
Whether the response sets a cookie
Content type (e.g. image)
Content sub-type (e.g. png)
Content length
Number of response headers
Request Features
Length of the URL
Subdomain check
Subdomain of the top-level domain check
Accept type (e.g. image, script)
UUID in URL
Ad or screen dimensions in URL
Third-party
Number of special characters in query string
Top-level domain in query string
Number of cookies in request
Semicolon in query string
Length of the query string
Ad/tracking keywords in URL (e.g. pixel, track)
Ad/tracking keywords in URL surrounded by special char.
Number of request headers
Request method (GET or POST)

Table 4.1. Features extracted from the request chains.

by considering the *number of unique domains* and the *length of the chain* as features. The *number of unique domains* represents the unique number of domains contacted and the *length of the chain* capture the total number of requests in the chain.

Relying on sequence further allows Khaleesi to capitalize on the growing chains. Specifically Khaleesi uses the *probability of the previous prediction* and the *average probability of the previous predictions* features to summarize properties of the prior request and response pairs. The *probability of the previous prediction* feature represents the classification probability in the previous classification and the the *average probability of the previous predictions* feature represents the average classification probability in prior classifications, indicating the likelihood of a request chain being advertising and tracking. To compute the *probability of the previous prediction* and *average probability of the previous predictions* features during the training phase, we mimic how these features would be computed during the real-time classification. Specifically, for each position in the chain, we train separate surrogate classifiers and classify (test) the redirects to compute the classification probability. To ensure that we do not test and train on the same data, we create 10 folds of data and test each fold by training on the remaining 9 folds. We limit the training of surrogate classifiers to 21 times because it is the maximum number of HTTP redirects allowed in Firefox and only 3.29% of the JavaScript request chains exceed that length. It is important to note that sequential features are updated after receiving each response and can only be computed after receiving the first response.

Response features capture properties that indicate the actions of the loaded content. For example, a response that loads a 1x1 image and sets a cookie is likely a tracking pixel. KHALEESI captures these properties by considering *content type*, *content length*, and *whether the response sets a cookie* as features. We also use *P3P* and *ETag* presence in response header as features, because P3P is a W3C standard used to specify cookie access policies [73] and prior research has shown that the ETag is exploited for tracking [143]. Similar to sequential features, response features are also computed after a response is received for a request.

Request features capture properties that can reveal the intent of the requester. For example, a request to a third-party domain that contains a large number of parameters in the query string may indicate that the third party is a tracker. KHALEESI captures these properties by considering the *length of the URL* and whether the request is from a *third-party*. The *length of the URL* feature captures the total number of characters in a URL and the *third-party* feature represents whether a request is first-party or third-party. We also try to capture the semantics of the content shared in the query string which may indicate that the request is advertising and tracking. Specifically, we use regular expressions to look for *UUID in URL* and *Ad or screen dimensions in URL*. The presence of *UUID in URL* indicates the leakage of a unique identifier and *Ad or screen dimensions in URL* indicates a request for an ad with certain dimensions. In contrast to the sequential and response features, the request features capture information that is available before a request is sent.

Network-layer vs. JavaScript-layer chains. For most of the features, the network-layer and JavaScript-layer chains have similar patterns. This allows us to use a single classifier to classify both types of chains. However, certain features are drastically different between network-layer and JavaScript-layer request chains, and that level of variance can confuse the classifier. For example, the *length of the chain* feature has smaller values for network-layer chains than JavaScript-layer chains. To mitigate these differences we include *chain type* as a meta-feature, which allows the classifier to avoid confusion when feature values vary significantly.

4.3.4 Classification

Khaleesi uses the random forest [155] machine learning algorithm to classify request chains. Random forest is an ensemble learning method that combines multiple decision trees and makes a prediction by taking the majority decision among trees. Each decision tree in random forest is trained on a random subset of the data and a random selection of the features (selected with replacement). Branches in a decision tree are constructed by splitting on features that provide the best separation for positive and negative samples. The random selection of data and features helps random forest avoid overfitting. We configure our random forest model to have 100 decision trees with randomly selected $int(\sqrt{N})$ features for each decision tree, where N is the total number of features.

Khaleesi's random forest model classifies individual requests in a request chain based on that request's sequential context. Since each request may potentially leak data to an ad and tracking server, Khaleesi tries to detect the ad and tracking request chains as early as possible. Note that in the case of network-layer request chains, no further requests are made from a chain once a request in the chain is detected and blocked as advertising and tracking. This is because each new request in the chain is initiated directly by the previous request. For JavaScript-layer request chains, all requests that are not classified as advertising and tracking are still sent by the script. This is because each request in the chain is initiated by an external script; blocking one request doesn't prevent the script from making another request to a non-blocked domain.

4.4 Evaluation

We rigorously evaluate KHALEESI along several dimensions.

4.4.1 Accuracy

Data Collection. We evaluate KHALEESI on crawl data collected using version 0.10.0 of OpenWPM [186] in August 2020 in the US. In each crawl, we visit the Alexa top-10K homepages. Interactive crawls additionally navigate to random sub-pages by clicking on `iframes` and anchor tags.

We test the following configurations:

1. A non-interactive crawl using Firefox configured to *allow* all third-party cookies.
2. An interactive crawl using Firefox configured to *allow* all third-party cookies.
3. An interactive crawl using Firefox configured to *block* all third-party cookies.
4. An interactive crawl using Firefox configured to spoof some basic properties of Safari. Specifically, we configure Firefox to block all third-party cookies, override the `User-Agent` HTTP header, and override the JavaScript-accessible `useragent`, `vendor`, `appVersion`, and `platform` properties.
5. A dataset of emails collected by Englehardt et al. [185], which used an older OpenWPM configured to emulate an email client by disabling JavaScript and stripping the `Referer` header.

These configurations represent a sample of browsing conditions that users may experience and choose while browsing the web. For example, spoofed Safari with third-party cookie blocking is an emulation of default settings in the actual Safari web browser. Table 4.2 summarizes the chains extracted from all of the crawled dataset configurations.

Ground truth labeling. We compare KHALEESI's classifications against a ground truth of advertising and tracking request chains provided by filter lists. In line with prior literature [149,203,216], we compile the set of advertising and tracking request chains by using filter lists as ground truth. Specifically, we use EasyList [114] and EasyPrivacy [39], two of the most widely used filter lists, to label advertising and tracking request chains. We label a request chain as advertising and tracking if at least one of the requests in the chain matches the filter lists. Further, we address the imperfect nature of the ground truth provided by filter lists [140,215,295] by doing a post-hoc validation of KHALEESI's disagreements with the filter lists.

Classifier training and testing. We train separate random forest classifiers for each of the crawled dataset configurations and use 10-fold cross validation to test the datasets. Specifically, we divide request chains into 10 folds, where we use 9 folds for training and 1 fold for testing and repeat the process for 10 times, ensuring that we do not train and test on the same data.

Results. Table 4.3 presents KHALEESI's evaluation across all of the datasets. When trained on the same configuration where testing occurs, KHALEESI's machine learning models perform well. However, users won't always have access to a classifier trained on their exact browser configuration. For example, KHALEESI may be used alongside other privacy extensions, or in a browser that blocks all third-party cook-

	Network			**JavaScript**		
Configuration	**# Requests**	**# Chains**	**%AT**	**# Requests**	**# Chains**	**%AT**
Cookies allowed (homepage)	192,038	76,816	78.7%	253,582	44,747	65.6%
Cookies allowed (interactive)	575,550	229,151	82.3%	1,320,733	145,135	61.1%
Cookies blocked (interactive)	432,935	183,230	78.6%	1,195,188	120,879	61.8%
Spoofed Safari (interactive)	384,404	166,018	70.5%	1,328,986	130,187	63.0%
Webmail (no JavaScript)	217,102	76,938	78.1%	–	–	–

Table 4.2. Total number of requests and request chains crawled for each configuration. AT refers to Advertising and Tracking.

Configuration	**Recall**	**Precision**	**Accuracy**
Cookies allowed (homepage)	98.87%	98.76%	98.63%
Cookies allowed (interactive)	99.24%	99.13%	99.06%
Cookies blocked (interactive)	99.10%	99.05%	98.97%
Spoofed Safari (interactive)	99.06%	99.02%	98.99%
Webmail (no JavaScript)	99.97%	99.97%	99.95%

Table 4.3. KHALEESI's accuracy in detection advertising and tracking request chains with a separate classifier for each of the crawled datasets.

ies. Thus, we evaluate KHALEESI's performance in these unanticipated use cases. Specifically, we train a random forest classifier on the homepage dataset and test this classifier on the other datasets. We find that KHALEESI is less accurate when trained on one dataset and tested on another (Table 4.4). Specifically, the accuracy drops by around 4% to 6% for each of our web crawl configurations, and by 19.38% for our webmail configuration. This decline in accuracy is mainly due to differences in the feature distributions. We highlight a few of these differences below.

There are several key differences between the homepage configuration and the others. First, 302 redirects are over-represented in the homepage crawl. 42.91% of the advertising and tracking redirect requests in the homepage configuration use 302 redirects as compared to the interactive crawls where cookies were allowed (28.56%), where cookies were blocked (22.91%), and where we spoofed Safari (18.81%). Second, navigations to new domains are more common in the homepage crawl. Specifically, 51.32% of the advertising and tracking requests navigate to new domains in the homepage crawl as compared to the interactive crawls where cookies were allowed (47.49%), where cookies were blocked (35.59%), and where we spoofed Safari (35.41%). Third, non advertising and non tracking URLs are shorter in the homepage crawl. Specifically, the lengths of non advertising and non tracking URL in the interactive crawls are between around 16 to 25 characters longer than the homepage crawl. Fourth, cookie-related features are also different between the stock and cookie blocking configurations. Specifically, there are an average of 3.1 cookies per advertising and tracking request in the homepage crawl (which does not block third-party cookies) compared to 0.37 cookies for the configurations that block third-party cookies. In the latter case, cookies can only be set and retrieved in the first-party context.

The webmail configuration has several important differences compared to the web crawls. First, the chains in the webmail configuration are 50.90% smaller than the homepage crawl. Second, the content embedded in webmail is almost exclusively images (i.e., 99.99% of requests), whereas only 59.03% of requests in the homepage crawl are for images. Third, 301 redirects are much more common in the webmail dataset. Specifically, 301 redirects are initiated by 25.59% requests in the webmail configuration, and are almost nonexistent in other configurations (e.g., only 0.48% of requests in the homepage configuration initiate 301 redirects).

Configuration	**Recall**	**Precision**	**Accuracy**
Cookies allowed (interactive)	95.59%	94.78%	94.44%
Cookies blocked (interactive)	94.90%	92.00%	92.60%
Spoofed Safari (interactive)	94.26%	92.16%	92.77%
Webmail (no JavaScript)	81.11%	97.33%	80.57%

Table 4.4. KHALEESI's accuracy in detection advertising and tracking request chains when trained on the homepage crawl configuration and tested on a different configuration.

Disagreements between KHALEESI and filter lists. Since our ground truth is imperfect, we analyze disagreements between KHALEESI and filter lists using the following heuristics inspired from prior work [216]. We check whether the URL contains any of the usual advertising and tracking keywords, such as *rtb*, *tracking*, and *adsbygoogle*. We also check whether a small tracking pixel (i.e., less than 5x5 pixels) is loaded. We apply this heuristic to the disagreements that occurred while testing with the homepage configuration classifier (Table 4.4). We find that many of the KHALEESI's "mistakes" are in fact mistakes in the ground truth. Overall, KHALEESI's accuracy improves by 0.78% for the non-interactive homepage crawl, 1.26% for the interactive crawl that doesn't block cookies, 1.75% for the interactive crawl that blocks cookies, 1.53% for the interactive crawl where Safari is spoofed, and 17.08% for webmail crawl as compared to the original accuracy, computed with filter lists as reference in Table 4.4. In fact, we note that EasyList and EasyPrivacy recently added several of the domains earlier detected as advertising/tracking by KHALEESI. Some examples include `sync.taboola.com`, `siteintercept.qualtrics.com`, `s0.2mdn.net`, and `log.popin.cc`. To our surprise, we notice that some of these domains are popular advertising services which should have been blocked by EasyList and EasyPrivacy. We further notice that the parent domains of some of these advertising services are already blocked by filter lists on some websites. For example, all resources from `taboola.com` are blocked on `independent.co.uk`, `scoopwhoop.com` and `technobuffalo.com`. We surmise that the filter lists take a conservative approach and avoid creating generic rules to block top level domains to mitigate breakage. Moreover, we find that the several of the KHALEESI's detected domains are still currently unblocked by EasyList and EasyPrivacy. Notable examples include `mediaiqdigital.com`, `quantumdex.io`, and `intentiq.com`.

4.4.2 Robustness Analysis

Next, we evaluate KHALEESI's classification accuracy over time and its robustness against evasion.

4.4.2.1 Classification accuracy over time

Machine learning models are prone to lose their effectiveness over time. Specifically, the underlying data distributions on which the model is trained change over time, in unforeseen ways, making predictions less accurate as the time passes [302]. KHALEESI tries to slow down the degradation by learning on fundamental properties of request chains, i.e., their sequential context, and by avoiding frequently changing variables such as domain names. Learning on sequential context makes KHALEESI's machine learning model generalizable, which can adapt to new, previously unseen, request chains because they will have the same distribution as the request chains that were used to train the initial model. To evaluate KHALEESI's accuracy over time, we test it on a new data set crawled in April 2021, i.e., 8 months after the initial crawl on which the model was trained. The results show that KHALEESI achieves an accuracy of 94.07% with a recall of 97.26% and a precision of 93.11% in detecting advertising and tracking request chains. We note that the accuracy only drops by 4.56%, recall by 1.61%, and precision by 5.65% due to the drift in the request chain distributions over time. We conclude that, to keep the accuracy consistent over time, the model must be fine-tuned continuously, which might include adding features to capture new patterns or removing features that capture inconsistent patterns.

4.4.2.2 Robustness against evasions

Due to the countermeasures against cross-site tracking by main-stream browsers, such as Safari [91] and Firefox [30], trackers are increasingly relying on evasive tactics to circumvent the deployed countermeasures. Prior research has shown that the URL-based ad and tracker detection approaches are vulnerable to domain rotation and URL path randomization [140, 308]. Since Khaleesi also relies on request properties, in addition to response and sequential properties, we evaluate its robustness against real world domain rotation and URL randomization attacks and as well as hypothetical response and sequence manipulation attacks. Specifically, we evaluate Khaleesi against real-world domain rotation [168, 319] and CNAME cloaking [278] attacks and hypothetical query string randomization, response header removal, and chain shortening attacks. We describe these evasion attacks below:

1. **Domain rotation** involves randomizing the label that is followed by the effective Top Level Domain (eTLD), e.g., example.com, example is the label and com is the eTLD.

2. **CNAME cloaking** involves switching the third party's eTLD+1 to the first party's eTLD+1, along with the addition of the third party's label as a sub domain.

3. **Query string randomization** involves randomization, removal, and MD5 hash-

ing of parameter name-value pairs.

4. **Response header removal** involves removal of ETag and P3P header from the response.

5. **Chain shortening** involves random truncation of request chains by 50–70%.

To launch these attacks, we randomly select 100 adversaries from the top 20% of the most common advertising and tracking third parties, which account for 16.35% of all the chains in our data set, i.e., cookies allowed homepage configuration. Our selection includes prominent advertisers and trackers, such as Criteo and PubMatic. Further, to provide a relative perspective, we draw a comparison between KHALEESI and CONRAD [271] (best approach from § 4.4). [3] Table 4.5 presents the adversaries' success rate against KHALEESI and CONRAD with different evasion attacks. Success rate is defined as the percentage of classification switches from advertising and tracking to non advertising and non tracking. In case of KHALEESI, we note that the adversaries are able to evade KHALEESI for 1.31–13.85% of the times. Whereas in case of CONRAD, the adversaries are able to evade the classifier for 8.57–9.71% of the times. It is noteworthy that the KHALEESI is much more robust as compared to CONRAD for all of the evasion attacks, except for CNAME cloaking, where the adversaries are able to switch 4.35% more instances from advertising and tracking to non advertising and non tracking. We attribute KHALEESI's robustness against

[3]Response header removal and chain shortening attacks are not applicable to CONRAD because it does not use response headers, other than the status code, and sequential context of request chains.

Evasion attack	**Khaleesi**	**CONRAD [271]**
Domain rotation	1.75%	8.57%
CNAME cloaking	13.85%	9.50%
QS randomization	3.94%	9.71%
Response removal	1.67%	–
Chains shortening	1.31%	–

Table 4.5. Evasion rate against Khaleesi and CONRAD.

adversarial evasion to complementary accuracy gained from request, response, and sequential features. Specifically, we note that request, response, and sequential features alone provide an accuracy of 98.69%, 91.94%, and 93.11%, respectively. Overall, we conclude that Khaleesi is fairly robust against adversarial evasions.

4.5 Performance

We evaluate Khaleesi's performance in real-time blocking of advertising and tracking request chains. We plan to address two questions: (1) Is Khaleesi suitable for online deployment? and (2) How does Khaleesi compare to prior real-time ML based ad and tracker blocking approaches?

Ad and tracker blocking tools generally improve the page load time by blocking content; however, at the same time, they also incur performance overhead. In traditional filter list based ad and tracker blocking tools, overheads are incurred in matching requests against filter lists and removing content from DOM. In ML based ad and tracker blockers, there are overheads to extract features and use the model to classify requests. As we show next, Khaleesi incurs overheads but improves the performance, significantly more, as compared to other ad and tracker blocking tools.

Khaleesi's performant implementation. We implement Khaleesi as a browser extension by extending the open source implementation of Adblock Plus [7]. Since Khaleesi is designed to only block advertising and tracking request chains, we complement Khaleesi with EasyList [114] and EasyPrivacy [39] for blocking non-request-chain based ads and trackers. Specifically, request chains are routed through Khaleesi and non-request-chain based requests are routed through filter lists. We create a lightweight in-memory representation of request chains and extract features from requests in a streaming fashion, before they leave the browser, and response headers, before even the whole response is received and parsed by the browser. Extracting features from response headers, before they are parsed, allows us to save time by avoiding the need to wait for their parsing and rendering. Moreover, we flush the in-memory representation when the browser navigates to another page. For network layer request chains, however, we flush the in-memory representation as soon as we block them or they return a non-redirect response.

Khaleesi's performance comparison. We quantify Khaleesi's overheads and performance improvements by comparing it against: (1) stock Firefox, (2) Adblock Plus configured with Firefox[4], and (3) AdGraph [216], a prior ML based ad and tracker blocking approach. We compare against stock Firefox and Adblock Plus by measuring the page load time[5], averaged over five runs for each website, on

[4]We disable Firefox's Enhanced Tracking Protection (ETP) and configure Adblock Plus with EasyList [114] and EasyPrivacy [39].

[5]Page load time is measured as a difference between `navigationStart` and `loadEventEnd` events of the `Performance` API.

Alexa top-500 websites. We simulate consistent network conditions by setting the bandwidth to 15 Mbps with a latency of 100 ms. For AdGraph comparison, we resort to the performance results quoted in the paper (see § IV.F [216] for more details) because a head-to-head comparison is not feasible due to several inherent differences in their implementation — the primary one being that AdGraph is implemented in Chromium whereas KHALEESI is implemented in Firefox. We also compare Adblock Plus's performance against stock Firefox as a baseline.

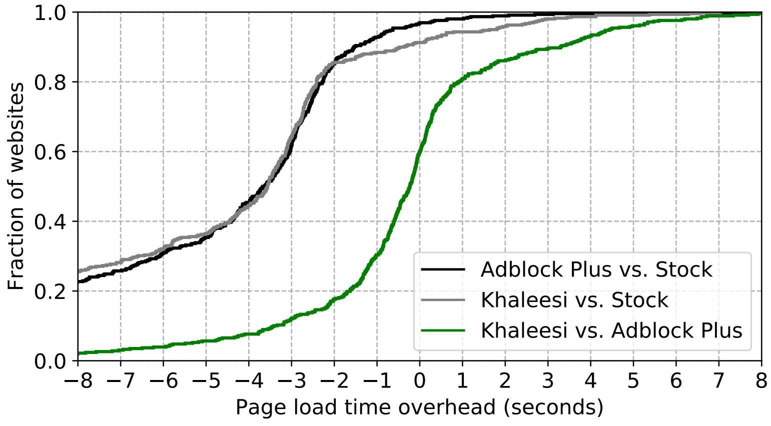


Figure 4.2. KHALEESI's overhead in terms of page load time.

Results. Figure 4.2 shows KHALEESI's page load time as compared to stock Firefox and Adblock Plus and as well as the Adblock Plus's performance against stock Firefox configuration. As compared to stock Firefox, we find that KHALEESI improves the page load time on 91.26% of the websites. KHALEESI's improvements over stock Firefox are due to blocking advertising and tracking content, which results in less network requests and less rendering. As compared to Adblock Plus, we find that the KHALEESI improves the page load time on 59.82% of the websites. The performance gains over Adblock Plus highlight KHALEESI's added benefit of blocking advertising and tracking request chains. As compared to AdGraph, KHALEESI has a net performance gain of about 2.2×. Specifically, AdGraph was faster than the stock Chromium on only 42% of the websites [216] whereas KHALEESI is faster than stock Firefox on 91.26% of the websites. KHALEESI's significant performance improvement over AdGraph can be attributed to KHALEESI's lightweight implementation. Specifically, AdGraph needs to construct and traverse its fine-grained graph representation to extract features for each request, whereas KHALEESI only needs to iteratively traverse request and response pairs.

Overall, we conclude that KHALEESI has significantly lower overheads as compared to both filter lists based and real-time ML based ad and tracker blockers. We believe that KHALEESI's improved accuracy outweighs its minor feature extraction and classification overheads, making it suitable for a performant online deployment.

4.5.1 Breakage Analysis

Content blocking tools are prone to website breakage because of incorrect blockage of functional resources or their dependence on advertising and tracking resources. To be usable in the real-world, KHALEESI' breakage must be on par with existing content blocking tools such as Adblock Plus [5]. Therefore, we evaluate and compare KHALEESI's breakage with Adblock Plus[6] on popular websites. We manually quantify website breakage on a random sample of 100 websites from top-1K list by conducting common browsing actions such as reading articles, adding products to carts, and opening sub pages of a website. We open each website using stock Firefox as a control and with Adblock Plus and KHALEESI as treatments. We open three browser instances side by side, analyze the website functionality on each of them, and assign one of the following labels:

1. **Major:** The core functionality of the website is broken and the user cannot fulfill their objective. For example, the user is unable to submit a form or book a flight.

2. **Minor:** The non-core functionality of the website is broken but the user is able to fulfill their objective. For example, some icons or images are missing on the webpage.

3. **None:** The experience is similar across control and treatments.

Tool	**None**		**Minor**		**Major**	
	#	%	#	%	#	%
KHALEESI	90.0	93.8	3.5	3.6	2.5	2.6
Adblock Plus	90.5	94.3	3.0	3.1	2.5	2.6

Table 4.6. Average breakage assessment of 2 reviewers.

[6]We configure Adblock plus with the same filter lists, i.e., EasyList [114] and EasyPrivacy [39], on which KHALEESI was trained.

To mitigate potential bias and subjectivity in manual breakage analysis and avoid any inconsistencies due to dynamic content, we asked two reviewers to independently analyze the test websites at the same time. The two reviewers achieved a high 94.79% agreement in their breakage determinations. Table 4.6 presents the averaged breakage results[7]. We note that, KHALEESI and Adblock Plus cause no breakage on 93.8% and 94.3% of the tested sites, respectively. Both cause major breakage on the same 2.6% of the sites. Only in one instance, KHALEESI misclassified and blocked a request to `intercomcdn.com`, leading to a minor breakage (missing chat button) on the website. We conclude that KHALEESI's breakage is on par with Adblock Plus and is thus feasible for deployment in a real world setting.

4.5.2 Performance

We evaluate KHALEESI's performance in real-time blocking of advertising and tracking request chains. We compare its performance against: (1) stock Firefox, (2) Adblock Plus configured with Firefox, and (3) AdGraph [216], a prior ML based ad and tracker blocking approach. As compared to stock Firefox, we find that KHALEESI improves the page load time on 91.26% of the websites. As compared to Adblock Plus, we find that the KHALEESI improves the page load time on 59.82% of the websites. As compared to AdGraph, KHALEESI has a net performance gain of about 2.2×. The detailed performance results are reported in Appendix 4.5.

[7]Out of the 100 websites, 5 websites failed to load in both control and treatments, so we exclude them from our analysis.

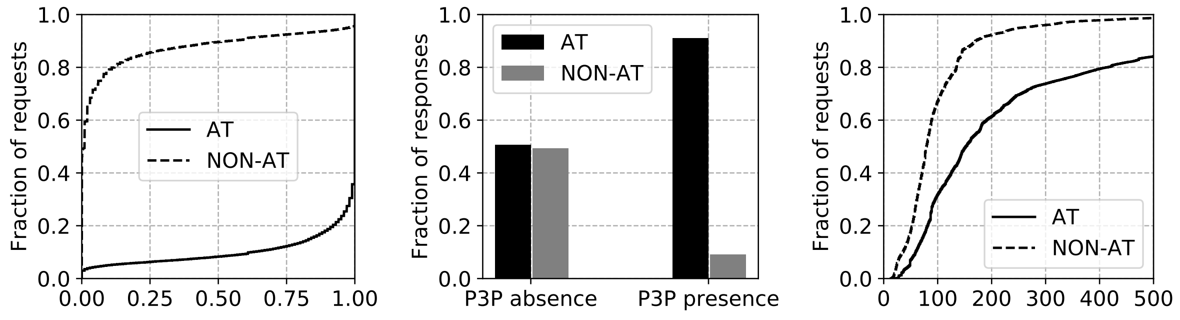
4.6 Discussion

In this section, we first discuss KHALEESI's inner workings by analyzing its features. We also analyze KHALEESI's findings to shed light into the online information sharing ecosystem through graph analysis and case studies emerging use cases of request chains.

4.6.1 Feature Analysis

To shed light into KHALEESI's inner workings, we analyze a few of the most important features, with the highest information gain [219] across sequential, response, and request feature categories. These features are the most helpful for KHALEESI in distinguishing advertising and tracking requests from non-advertising and non-tracking requests in the request chains.

Probability of previous prediction provides the highest information gain amongst the sequential features. The probability that the previous prediction was advertising and tracking captures the information capitalized by KHALEESI as the chains grow longer in length. Intuitively, using the prior probability helps KHALEESI make a better determination. For example, if KHALEESI is moderately confident that the request will load an ad or a tracker, it can reaffirm that by leveraging the confidence from its prior prediction. Figure 4.3a plots the distribution of the probability of previous prediction. We note that 85.47% of the non-advertising and non-tracking requests have less than 25% probability of being an ad or a tracker. Whereas, 93.74% of the advertising and tracking requests have greater than 25% probability in being an ad or a tracker.



(a) Probability of previous prediction. (b) P3P is response header. (c) Length of URL.

Figure 4.3. Distribution of top sequential, response, and request features. AT refers to advertising and tracking requests and NON-AT refers to non advertising and non tracking requests.

The presence of a P3P response header provides the highest information gain amongst the response features. P3P (Platform for Privacy Preferences) is a W3C standard that allows sites to convey their privacy policies in a standardized format [73]. Internet Explorer by default rejected third-party cookies in first party context unless the cookie usage was specified by a P3P header. P3P was implemented by Internet Explorer, Edge, and Mozilla [1, 240] as well as supported by thousands of domains at one point [171], but it is not used now except for by older versions of Internet Explorer. Despite this, we find that many third parties still use P3P as the HTTP response header to specify their cookie usage policy. Figure 4.3b shows the prevalence of P3P headers in request chains. It can be seen from the figure that 90.99% of advertising and tracking responses have a P3P header as compared to only 9.01% of the non-advertising and non-tracking responses.

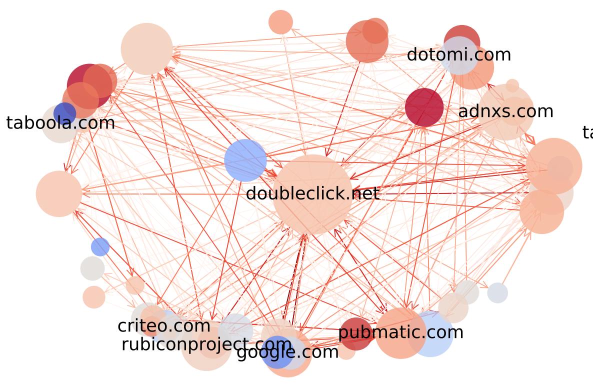
URL length provides the highest information gain amongst the request features. Advertisers and trackers collect sensitive information from users' devices and share it with others as URL query strings. This so-called *link decoration* is increasingly being used to bypass privacy protections against third party cookies [222, 223, 311]. As can be seen in Figure 4.3c, this information sharing leads to longer URLs for advertising and tracking requests. 38.78% of advertising and tracking URLs are longer than 200 characters, while only 7.79% of the non-advertising and non-tracking URLs are longer than 200 characters.

4.6.2 Request Chain Graph

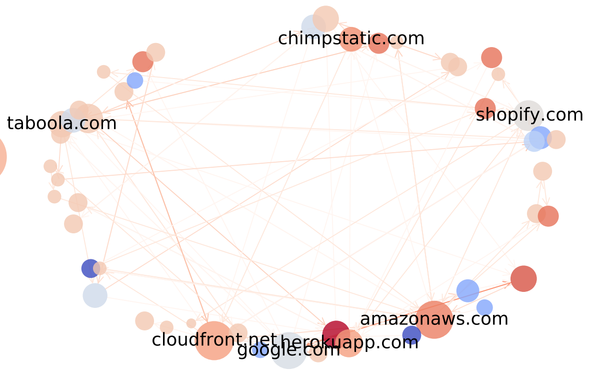
Next, we build the request chain graph to understand bilateral information sharing relationships between different entities. To this end, we construct a graph such that the nodes represent domains and the edges represent consecutive domains in request chains. Note that edges are directed and weighted: the source/destination of the directed edge represents the source/destination in the redirect and the edge weight represents the frequency of their co-occurrence.[8] Also, the size of a node is proportional to its degree and the color represents degree ratio—ratio of in-degree and out-degree—red/blue represents higher/lower in-degree as compared to out-degree. Note that for the node colors, darker shades of either blue or red mean higher asymmetry and lighter shades mean higher symmetry.

Figure 4.4(a) plots the largest strongly connected component (LSCC) of the

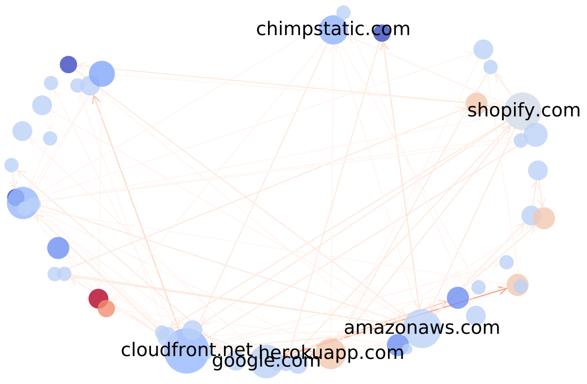
[8]For example, a request chain example1.com → sub.example2.com → example3.com would be represented in a graph with 3 nodes (example1.com, example2.com, example3.com) and 2 edges (example1.com → example2.com, example2.com → example3.com).



(a) No Blocking



(b) Filter Lists



(c) Khaleesi

Figure 4.4. Request chain graph of redirects between top-50 most popular domains.

request chain graph constructed using the first crawl configuration (cookies allowed (homepage)). The nodes with higher/lower degree ratios (in shades of red/blue) are destinations/sources of redirects. We note that the destinations of most of the redirects are typically well-known advertising and tracking domains such as doubleclick.com and pubmatic.com. In contrast, their sources typically include general-purposed cloud providers such as cloudfront.net as well as tag management services such as googletagmanager.com. This indicates that advertising and tracking domains are generally the recipients of tracking information from other domains.

Other SCCs (not shown in Figure 4.4) are much smaller in size (most of them with around 10 nodes or less). We note that most of the nodes in these smaller SCCs are mostly cliques of related domains (e.g., amazon.fr $\rightarrow$ amazon.com, towardsdatascience.com $\rightarrow$ medium.com) and sometimes country-specific domains (e.g., admaster.com.cn, reachmax.cn, yoyi.com.cn). Also not shown in Figure 4.4 are self-loops, which mostly represent sub-domains trying to sync cookies with their parent domains (e.g., aliexpress.com, admicro.vn).

Figures 4.4(b) and (c) plot the LSCC of the request chain graph after blocking using filter lists and KHALEESI, respectively. It is evident that the request chain graph becomes significantly more sparse when advertising and tracking requests are blocked by either filter lists or KHALEESI. More specifically, we note that both node sizes and number of edges significantly reduce for filter lists or KHALEESI in Figures 4.4(b) and (c) as compared to no blocking in Figure 4.4(a). In fact, some of the largest nodes in Figure 4.4(a) (e.g., rubiconproject.com) completely disappear in Figures 4.4(b) and (c). Comparing Figures 4.4(b) and (c), we note that even more nodes (e.g., taboola.com) disappear and edges' weights decrease going from filter lists to KHALEESI. In Figure 4.4(c), we observe mostly smaller sized nodes in blue colors represent popular cloud providers such as amazonaws.com and cloudfront.net.

To further analyze these differences quantitatively, Table 4.7 lists a few key graph connectivity metrics based on clustering and connected components. These metrics quantitatively demonstrate that redirect graphs progressively become more fragmented as we go from no blocking to blocking with filter lists and KHALEESI.

	No Blocking	**Filter-list**	**Khaleesi**
Average Clustering	0.03283	0.00214	0.00155
# of SCC	10221	9852	9732
# of Nodes in LSCC	865	83	52
Top Degree in LSCC	603	88	77

Table 4.7. Statistics of graphs in three configurations.

For example, the size of the LSCC decreases by 10× due to filter lists, and further decreases by 37% due to Khaleesi. We note a similar trend for other reported metrics in Table 4.7.

In sum, our analysis shows that request chains enable widespread information sharing between third-party advertising and tracking domains. Both filter lists and Khaleesi help mitigate this by degrading the graph connectivity. We also conclude that Khaleesi is more effective than just filter lists in mitigating the information sharing by advertising and tracking request chains.

4.7 Concluding Remarks

In this chapter we proposed Khaleesi, a machine learning based approach that capitalizes on sequential context to detect advertising and tracking request chains. We conclude the chapter by discussing some limitations and future work.

Data collection: Our data collection has some limitations that pose internal and external threats to the validity of our findings. For crawling, we used OpenWPM instrumented browser, which is more complete than primitive crawlers such as PhantomJS [144] but is still detectable [201]. Our web crawler used a vantage point at an academic institute in the United States and the results may vary across different networks (e.g., residential) and geographic locations (e.g., Europe).

Feature robustness: KHALEESI's robustness against feature manipulation is not foolproof. A motivated adversary can, in theory, dial-up these feature manipulations (§4.4.2.2) to evade detection by KHALEESI. However, that would require significant changes to the infrastructure, techniques, and working model of the current advertising and tracking ecosystem. For example, changing domain names and query string parameters requires non-trivial coordination between front-end and back-end across several advertisers and trackers [133, 134].

Adversarial attacks: Similar to adversarial attacks on neural networks (e.g., FGSM [200]), KHALEESI's decision tree ensemble classifier may also be susceptible. This is an active area of research and future work can look into available countermeasures to harden decision tree ensemble classifiers [161, 162, 226].

Model accuracy over time: The web is continuously changing and so is the online advertising and tracking ecosystem. As the advertising and tracking techniques evolve, this may change the underlying data distributions on which KHALEESI is trained. While we showed that KHALEESI's accuracy degraded only minimally (§4.4.2.1), it would require periodic re-training to ensure high accuracy. Future work can look into automatically updating KHALEESI's ML classifier using online learning techniques based on user feedback.

Integration with other tools: KHALEESI focuses only on request chains and other standalone network requests are considered outside the scope. It is meant to be complemented with filter lists to handle standalone network requests. Future work can look into integrating Khaleesi into existing ML-based ad and tracker blocking tools (e.g., [216, 290]).

4.8 Appendix

4.9 Features definitions

Below we describe all features from Table 4.1.

4.9.1 Sequential features

1. *Consecutive requests to the same domain:* Count of the number of requests in the chain that follow each other to the same domain.

2. *Number of unique domains in the chain:* Count of the number of unique domains that appear in a request chain.

3. *Length of the chain:* Count of the number of requests that appear in a request chain.

4. *Probability of the previous prediction:* Classification probability in the previous classification.

5. *Average probability of the previous predictions:* Average classification probability in prior classifications.

4.9.2 Response features

1. *Status code:* HTTP Status code returned by the response (e.g. 200, 302, 404).

2. *ETag in response header:* Checks for the presence of ETag property in the response

headers.

3. *P3P in response header :* Checks for the presence of P3P property in the response headers.

4. *Whether the response sets a cookie:* Checks whether the response returns a cookie to be set.

5. *Content type:* Captures the type of Content-Type (Content-Type=type/subtype) property of the response header (e.g. image).

6. *Content sub-type:* Captures the subtype of Content-Type (Content-Type=type/subtype) property of the response header (e.g. png).

7. *Content length:* Count of characters returned in content.

8. *Number of response headers:* Count of properties returned in the response header.

4.9.3 Request features

1. *Length of the URL:* Count of characters in the URL.

2. *Subdomain check:* Whether the hostname has a subdomain.

3. *Subdomain of the top-level domain check:* Whether the hostname is a subdomain of the top-level document's domain.

4. *Accept type:* Captures the MIME type, determined by the Content-Type header (e.g. image, script).

5. *UUID in URL:* Presence of a UUID pattern (........-....-....-....-............) in a URL.

6. *Ad or screen dimensions in URL:* Presence of ad/screen dimensions (x|X followed by 2-4 digits on each side) in a URL.

7. *Third-party:* Whether the request is directed to a third party.

8. *Number of special characters in query string:* Count of alphanumeric characters in the query string.

9. *Top-level domain in query string:* Whether the top level domain appears in the query string.

10. *Number of cookies in request:* Count of cookies sent in a request.

11. *Semicolon in query string:* Whether the query string contains semicolon.

12. *Length of the query string:* Count of characters in the query string.

13. *Ad/tracking keywords in URL:* Whether the URL contains common advertising and tracking keywords (e.g. pixel, track).

14. *Ad/tracking keywords in URL surrounded by special char:* Whether the URL contains common advertising and tracking keywords that are surrounded by alphanumeric characters.

15. *Number of request headers:* Count of properties sent in the request header.

16. *Request method:* Whether the request is sent through GET or POST method.

CHAPTER 5

FINGERPRINTING THE FINGERPRINTERS: LEARNING TO DETECT BROWSER FINGERPRINTING BEHAVIORS

5.1 Introduction

Mainstream browsers have started to provide built-in protection against cross-site tracking. For example, Safari [50] now blocks all third-party cookies and Firefox [315] blocks third-party cookies from known trackers by default. As mainstream browsers implement countermeasures against stateful tracking, there are concerns that it will encourage trackers to migrate to more opaque, stateless tracking techniques such as browser fingerprinting [283]. Thus, mainstream browsers have started to explore mitigations for browser fingerprinting.

Some browsers and privacy tools have tried to mitigate browser fingerprinting by changing the JavaScript API surface exposed by browsers to the web. For example, privacy-oriented browsers such as the Tor Browser [90, 232] have restricted access to APIs such as Canvas and WebRTC, that are known to be abused for browser fingerprinting. However, such blanket API restriction has the side effect of breaking websites that use these APIs to implement benign functionality.

Mainstream browsers have so far avoided deployment of comprehensive API restrictions due to website breakage concerns. As an alternative, some browsers—Firefox in particular [182]—have tried to mitigate browser fingerprinting by blocking network requests to browser fingerprinting services [31]. However, this approach relies heavily on manual analysis and struggles to restrict fingerprinting scripts that are served from first-party domains or dual-purpose third parties, such as CDNs. Englehardt and Narayanan [186] manually designed heuristics to detect fingerprinting scripts based on their execution behavior. However, this approach relies on hard-coded heuristics that are narrowly defined to avoid false positives and must be continually updated to capture evolving fingerprinting and non-fingerprinting behaviors.

We propose FP-Inspector, a machine learning based approach to detect browser fingerprinting. FP-Inspector trains classifiers to learn fingerprinting behaviors by extracting syntactic and semantic features through a combination of static and dynamic analysis that complement each others' limitations. More specifically, static analysis helps FP-Inspector overcome the *coverage* issues of dynamic analysis, while dynamic analysis overcomes the inability of static analysis to handle *obfuscation*.

Our evaluation shows that FP-Inspector detects fingerprinting scripts with 99.9% accuracy. We find that FP-Inspector detects 26% more fingerprinting scripts than manually designed heuristics [186]. Our evaluation shows that FP-Inspector helps significantly reduce website breakage. We find that targeted countermeasures that leverage FP-Inspector's detection reduce breakage by a factor 2 on websites that are particularly prone to breakage.

We deploy FP-Inspector to analyze the state of browser fingerprinting on the web. We find that fingerprinting prevalence has increased over the years [135,186], and is now present on 10.18% of the Alexa top-100K websites. We detect fingerprinting scripts served from more than two thousand domains, which include both anti-ad fraud vendors as well as cross-site trackers. FP-Inspector also helps us uncover several new APIs that were previously not known to be used for fingerprinting. We discover that fingerprinting scripts disproportionately use APIs such as the `Permissions` and `Performance` APIs.

We summarize our key contributions as follows:

1. An **ML-based syntactic-semantic approach** to detect browser fingerprinting behaviors by incorporating both static and dynamic analysis.

2. An **evaluation of website breakage** caused by different mitigation strategies that block network requests or restrict APIs.

3. A **measurement study** of browser fingerprinting scripts on the Alexa top-100K websites.

4. A **clustering analysis of JavaScript APIs** to uncover new browser fingerprinting vectors.

Chapter Organization: The rest of the chapter proceeds as follows. Section 5.2 presents an overview of browser fingerprinting and limitations of existing countermeasures. Section 5.3 describes the design and implementation of FP-Inspector. Section 5.4 presents the evaluation of FP-Inspector's accuracy and website breakage. Section 5.5 describes FP-Inspector's deployment on Alexa top-100K websites. Section 5.6 presents the analysis of JavaScript APIs used by fingerprinting scripts. Section 5.7 describes FP-Inspector's limitations. Section 5.8 concludes the chapter.

5.2 Background & Related Work

Browser fingerprinting for online tracking. Browser fingerprinting is a stateless tracking technique that uses device configuration information exposed by the browser through JavaScript APIs (e.g., `Canvas`) and HTTP headers (e.g., `User-Agent`). In contrast to traditional stateful tracking, browser fingerprinting is stateless—the tracker does not need to store any client-side information (e.g., unique identifiers in cookies or local storage). Browser fingerprinting is widely recognized by browser vendors [13, 28, 57] and standards bodies [95, 265] as an abusive practice. Browser fingerprinting is more intrusive than cookie-based tracking for two reasons: (1) while cookies are observable in the browser, browser fingerprints are opaque to users; (2) while users can control cookies (e.g., disable third-party cookies or delete cookies altogether), they have no such control over browser fingerprinting.

Browser fingerprinting is widely known to be used for bot detection purposes [64, 177, 245, 259], including by Google's reCAPTCHA [157, 289] and during general web authentication [139,233]. However, there are concerns that browser fingerprinting may be used for cross-site tracking especially as mainstream browsers such as Safari [311] and Firefox [315] adopt aggressive policies against third-party cookies [283]. For example, browser fingerprints (by themselves or when combined with IP address) [234] can be used to regenerate or de-duplicate cookies [88, 285]. In fact, as we show later, browser fingerprinting is used for both anti-fraud and potential cross-site tracking.

Origins of browser fingerprinting. Mayer [250] first showed that "quirkiness" can be exploited using JavaScript APIs (e.g., navigator, screen, Plugin, and

MimeType objects) to identify users. Later, Eckersley [181] conducted the Panopticlick experiment to analyze browser fingerprints using information from various HTTP headers and JavaScript APIs. As modern web browsers have continued to add functionality through new JavaScript APIs [293], the browser's fingerprinting surface has continued to expand. For example, researchers have shown that `Canvas` [257], `WebGL` [159,257], fonts [190], extensions [297], the `Audio` API [186], the `Battery Status` API [266], and even mobile sensors [175] can expose identifying device information that can be used to build a browser fingerprint. In fact, many of these APIs have already been found to be abused in the wild [135,137,175,186,263,267]. Due to these concerns, standards bodies such as the W3C [96] have provided guidance to take into account the fingerprinting potential of newly proposed JavaScript APIs. One such example is the Battery Status API, which was deprecated by Firefox due to privacy concerns [267].

Does browser fingerprinting provide unique and persistent identifiers? A browser fingerprint is a "statistical" identifier, meaning that it does not deterministically identify a device. Instead, the identifiability of a device depends on the number of devices that share the same configuration. Past research has reported widely varying statistics on the uniqueness of browser fingerprints. Early research by Laperdrix et al. [235] and Eckersley [181] found that 83% to 90% of devices have a unique fingerprint. In particular, Laperdrix et al. found that desktop browser fingerprints are more unique (90% of devices) than mobile (81% of devices) due to the presence of plugins and extensions. However, both Eckersley's and Laperdrix's studies are based on data collected from self-selected audiences—visitors to Panopticlick and AmIUnique, respectively—which may bias their findings. In a more recent study, Boix et al. [199] deployed browser fingerprinting code on a major French publisher's website. They found that only 33.6% of the devices in that sample have unique fingerprints. However, they argued that adding other properties, such as the IP address, *Content language* or *Timezone*, may make the fingerprint unique.

To be used as a tracking identifier, a browser fingerprint must either remain stable over time or be linkable with relatively high confidence. Eckersley measured repeat visits to the Panopticlick test page and found that 37% of repeat visitors had more than one fingerprint [181]. However, about 65% of devices could be re-identified by linking fingerprints using a simple heuristic. Similarly, Vastel et al. [305] found that half of the repeat visits to the AmIUnique test page change their fingerprints in less than 5 days. They improve on Eckersley's linking heuristic and show that their linking technique can track repeat AmIUnique visitors for an average of 74 days.

Prevalence of browser fingerprinting. A 2013 study of browser fingerprinting in the wild [263] examined three fingerprinting companies and found only 40 of the Alexa top-10K websites deploying fingerprinting techniques. That same year, a large-scale study by Acar et al. [137] found just 404 of the Alexa top 1-million websites deploying fingerprinting techniques. Following that, a number of studies have measured the deployment of fingerprinting across the web [135, 175, 186, 267]. Although these studies use different methods to fingerprinting, their results suggest an overall trend of increased fingerprinting deployment. Most recently, an October

2019 study by The Washington Post [193] found fingerprinting on about 37% of the Alexa top-500 US websites. This roughly aligns with our findings in Section 5.5, where we discover fingerprinting scripts on 30.60% of the Alexa top-1K websites. Despite increased scrutiny by browser vendors and the public in general, fingerprinting continues to be prevalent.

Browser fingerprinting countermeasures. Existing tools for fingerprinting protection broadly use three different approaches.[1] One approach randomizes return values of the JavaScript APIs that can be fingerprinted, the second normalizes the return values of the JavaScript APIs that can be fingerprinted, and the third uses heuristics to detect and block fingerprinting scripts. All of these approaches have different strengths and weaknesses. Some approaches protect against *active* fingerprinting, i.e. scripts that probe for device properties such as the installed fonts, and others protect against *passive* fingerprinting, i.e. servers that collect information that's readily included in web requests, such as the `User-Agent` request header. Randomization and normalization approaches can defend against all forms of active fingerprinting and some forms of passive (e.g., by randomizing `User-Agent` request header). Heuristic-based approaches can defend against both active and passive fingerprinting, e.g., by completely blocking the network request to resource that fingerprints. We further discuss these approaches and list their limitations.

[1]Google has recently proposed a new approach to fingerprinting protection that doesn't fall into the categories discussed above. They propose assigning a "privacy cost" based on the entropy exposed by each API access and enforcing a "privacy budget" across all API accesses from a given origin [28]. Since this proposal is only at the ideation stage and does not have any implementations, we do not discuss it further.

1. The *randomization* approaches, such as Canvas Defender [22], randomize the return values of the APIs such as `Canvas` by adding noise to them. These approaches not only impact the functional use case of APIs but are also ineffective at restricting fingerprinting as they are reversible [304]. Additionally, the noised outputs themselves can sometimes serve as a fingerprint, allowing websites to identify the set of users that have the protection enabled [176, 304].

2. The *JavaScript API normalization* approaches, such as those used by the Tor Browser [44] and the Brave browser [17], attempt to make all users return the same fingerprint. This is achieved by limiting or spoofing the return values of some APIs (e.g., `Canvas`), and entirely removing access to other APIs (e.g., `Battery Status`). These approaches limit website functionality and can cause websites to break, even when those websites are using the APIs for benign purposes.

3. The *heuristic* approaches, such as Privacy Badger [76] and Disconnect [31], detect fingerprinting scripts with pre-defined heuristics. Such heuristics, which must narrowly target fingerprinters to avoid over-blocking, have two limitations. First, they may miss fingerprinting scripts that do not match their narrowly defined detection criteria. Second, the detection criteria must be constantly maintained to detect new or evolving fingerprinting scripts.

Learning based solutions to detect fingerprinting. The ineffectiveness of randomization, normalization, and heuristic-based approaches motivate the need of a learning-based solution. Browser fingerprinting falls into the broader class of *stateless* tracking, i.e., tracking without storing on data on the user's machine. Stateless tracking is in contrast to *stateful* tracking, which uses APIs provided by the browser to store an identifier on the user's device. Prior research has extensively explored learning-based solutions for detecting stateful trackers. Such approaches try to learn tracking behavior of scripts based on their structure and execution. One such method by Ikram et al. [209] used features extracted through static code analysis. They extracted n-grams of code statements as features and trained a one-class machine learning classifier to detect tracking scripts. In another work, Wu et al. [316] used features extracted through dynamic analysis. They extracted one-grams of web API method calls from execution traces of scripts as features and trained a machine learning classifier to detect tracking scripts.

Unfortunately, prior learning-based solutions generally lump together stateless and stateful tracking. However, both of these tracking techniques fundamentally differ from each other and a solution that tries to detect both stateful and stateless techniques will have mixed success. For example, a recent graph-based machine learning approach to detect ads *and* trackers proposed by Iqbal et al. [216] at times successfully identified fingerprinting and at times failed.

Fingerprinting detection has not received as much attention as stateful tracking detection. Al-Fannah et. al. [138] proposed to detect fingerprinting vendors by matching 17 manually identified attributes (e.g., `User-Agent`), that have fingerprinting potential, with the request URL. The request is labeled as fingerprinting if at least one of the attributes is present in the URL. However, this simple approach would incorrectly detect the functional use of such attributes as fingerprinting. Moreover, this approach fails when the attribute values in the URL are hashed or encrypted. Rizzo [303], in their thesis, explored the detection of fingerprinting scripts using machine learning. Specifically, they trained a machine learning classifier with features extracted through static code analysis. However, only relying on static code analysis might not be sufficient for an effective solution. Static code analysis has inherent limitations to interpret obfuscated code and provide clarity in enumerations. These limitations may hinder the ability of a classifier, trained on features extracted through static analysis, to correctly detect fingerprinting scripts as both obfuscation [291] and enumerations (canvas font fingerprinting) are common in fingerprinting scripts. Dynamic analysis of fingerprinting scripts could solve that problem but it requires scripts to execute and scripts may require user input or browser events to trigger.

A complementary approach that uses both static and dynamic analysis could work—indeed this is the approach we take next in Section 5.3. Dynamic analysis can provide interpretability for obfuscated scripts and scripts that involve enumerations and static analysis could provide interpretability for scripts that require user input or browser triggers.

5.3 FP-Inspector

In this section we present the design and implementation of FP-Inspector, a machine learning approach that combines static and dynamic JavaScript analysis to counter browser fingerprinting. FP-Inspector has two major components: the *detection component*, which extracts syntactic and semantic features from scripts and trains a machine learning classifier to detect fingerprinting scripts; and the *mitigation component*, which applies a layered set of restrictions to the detected fingerprinting scripts to counter passive and/or active fingerprinting in the browser. Figure 5.1 summarizes the architecture of FP-Inspector.

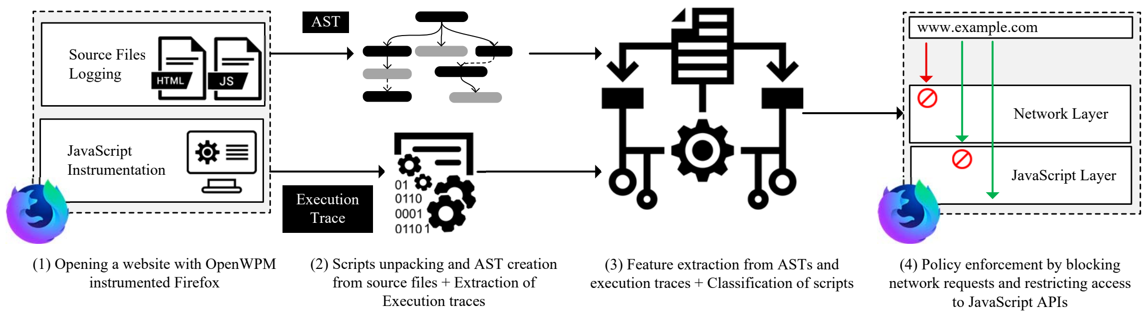

Source Files
Logging
HTML
JS
JavaScript
Instrumentation
AST
Execution
Trace
01
0110
0001
01101
www.example.com
Network Layer
JavaScript Layer
(1) Opening a website with OpenWPM instrumented Firefox
(2) Scripts unpacking and AST creation from source files + Extraction of Execution traces
(3) Feature extraction from ASTs and execution traces + Classification of scripts
(4) Policy enforcement by blocking network requests and restricting access to JavaScript APIs

Figure 5.1. FP-Inspector: (1) We crawl the web with an extended version of OpenWPM that extracts JavaScript source files and their execution traces. (2) We extract Abstract Syntax Trees (ASTs) and execution traces for all scripts. (3) We use those representations to extract features and train a machine learning model to detect fingerprinting scripts. (4) We use a layered approach to counter fingerprinting scripts.

5.3.1 Detecting fingerprinting scripts

A fingerprinting script has a limited number of APIs it can use to extract a specific piece of information from a device. For example, a script that tries to inspect the graphics stack must use the `Canvas` and `WebGL` APIs; if a script wants to collect 2D renderings (i.e., for canvas fingerprinting), it must call `toDataURL()` or `getImageData()` functions of the Canvas API to access the rendered canvas images. Past research has used these patterns to manually curate heuristics for detecting fingerprinting scripts with fairly high precision [175, 186]. Our work builds on them and significantly extends prior work in two main ways.

First, FP-Inspector *automatically* learns emergent properties of fingerprinting scripts instead of relying on hand-coded heuristics. Specifically, we extract a large number of low-level heuristics for capturing syntactic and semantic properties of fingerprinting scripts to train a machine learning classifier. FP-Inspector's classifier trained on limited ground truth of fingerprinting scripts from prior research is able to generalize to detect new fingerprinting scripts as well as previously unknown fingerprinting methods.

Second, unlike prior work, we leverage *both* static features (i.e., script syntax) and dynamic features (i.e., script execution). The static representation allows us to capture fingerprinting scripts or routines that may not execute during our page visit (e.g., because they require user interaction that is hard to simulate during automated crawls). The dynamic representation allows us to capture fingerprinting scripts that are obfuscated or minified. FP-Inspector trains separate supervised machine learning models for static and dynamic representations and combines their output to accurately classify a script as fingerprinting or non-fingerprinting.

Script monitoring. We gather script contents and their execution traces by automatically loading webpages in an extended version of OpenWPM [186]. By collecting both the raw content and dynamic execution traces of scripts, we are able to use both static and dynamic analysis to extract features related to fingerprinting.

Collecting script contents: We collect script contents by extending OpenWPM's network monitoring instrumentation. By default, this instrumentation saves the contents of all HTTP responses that are loaded into script tags. We extend OpenWPM to also capture the response content for all HTML documents loaded by the browser. This allows us to capture both external and inline JavaScript. We further parse the HTML documents to extract inline scripts. This detail is crucial because a vast majority of webpages use inline scripts [236, 261].

Collecting script execution traces: We collect script execution traces by extending OpenWPM's script execution instrumentation. OpenWPM records the name of the Javascript API being accessed by a script, the method name or property name of the access, any arguments passed to the method or values set or returned by the property, and the stack trace at the time of the call. By default, OpenWPM only instruments a limited number of the JavaScript APIs that are known to be used by fingerprinting scripts. We extend OpenWPM script execution instrumentation to cover additional APIs and script interactions that we expect to provide useful information for differentiating fingerprinting activity from non-fingerprinting activity. There is no canonical list of fingerprintable APIs, and it is not performant to instrument the browser's entire API surface within OpenWPM. In light of these constraints, we extended the set of APIs instrumented by OpenWPM to cover several additional APIs used by popular fingerprinting libraries (i.e., fingerprintjs2 [45]) and scripts (i.e., MediaMath's fingerprinting script [68]).[2] These include the Web Graphics Library (`WebGL`) and `performance.now`, both of which were previously not monitored by OpenWPM. We also instrument a number of APIs used for Document Object

[2]The full set of APIs monitored by our extended version of OpenWPM in Appendix 5.9.1.

Model (DOM) interactions, including the `createElement` method and the `document` and `node` objects. Monitoring access to these APIs allows us to differentiate between scripts that interact with the DOM and those that do not.

Static analysis. Static analysis allows us to capture information from the contents and structure of JavaScript files—including those which did not execute during our measurements or those which were not covered by our extended instrumentation.

AST representation: First, we represent scripts as Abstract Syntax Trees (ASTs). This allows us to ignore coding style differences between scripts and ever changing JavaScript syntax. ASTs encode scripts as a tree of syntax primitives (e.g., `VariableDeclaration` and `ForStatement`), where edges represent syntactic relationship between code statements. If we were to build features directly from the raw contents of scripts, we would encode extraneous information that may make it more difficult to determine whether a script is fingerprinting. As an example, one script author may choose to loop through an array of device properties by index, while another may choose to use that same array's `forEach` method. Both scripts are accessing the same device information in a loop, and both scripts will have a similar representation when encoded as ASTs.

Figure 5.2b provides an example AST built from a simple script. Nodes in an AST represent keywords, identifiers, and literals in the script, while edges represent the relation between them. *Keywords* are reserved words that have a special meaning for the interpreter (e.g. `for`, `eval`), *identifiers* are function names or variable names (e.g. `CanvasElem`, `FPDict`), and *literals* are constant values, such as a string assigned to an identifier (e.g. "`example`"). Note that whitespace, comments, and coding style are abstracted away by the AST.

Script unpacking: The process of representing scripts as ASTs is complicated by the fact that JavaScript is an interpreted language and compiled at run time. This allows portions of the script to arrive as plain text which is later compiled and executed with `eval` or `Function`. Prior work has shown that the fingerprinting scripts often include code that has been "packed" with `eval` or `Function` [291]. To unpack scripts containing `eval` or `Function`, we embed them in empty HTML webpages and open them in an instrumented browser [216] which allows us to extract scripts as they are parsed by the JavaScript engine. We capture the parsed scripts and use them in place of the packed versions when building ASTs. We also follow this same procedure to extract in-line scripts, which are scripts included directly in the HTML document.

Script 5.1 shows an example canvas font fingerprinting script that has been packed with `eval`. This script loops through a list of known fonts and measures the rendered width to determine whether the font is installed (see [186] for a thorough description of canvas font fingerprinting). Script 5.2 shows the unpacked version of the script. As can be seen from the two snippets, the script is significantly more interpretable after unpacking. Figure 5.2 shows the importance of unpacking to AST generation. The packed version of the script (i.e., Script 5.1) creates a generic stub AST (i.e., Figure 5.2a) which would match the AST of any script that uses `eval`. Figure 5.2b shows the full AST that has been generated from the unpacked version of the script (i.e., Script 5.2). This AST captures the actual structure and content of the fingerprinting code that was passed to `eval`, and will allow us to extract meaningful features from the script's contents.

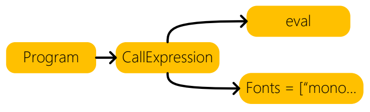
```
eval("Fonts =[\"monospace\",..,\"sans-serif\"];CanvasElem = document.
createElement(\"canvas\");CanvasElem.width = \"100\";CanvasElem.height =
\"100\";context = CanvasElem.getContext('2d');FPDict= {};
for(i=0;i<Fonts.length;i++){CanvasElem.font = Fonts[i];FPDict[Fonts[i]] =
CanvasElem.measureText(\"example\").width;}")
```

Script 5.1. A canvas font fingerprinting script packed with eval.

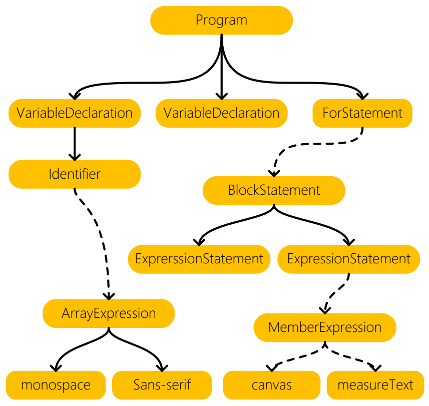
```
// Canvas font fingerprinting script.
Fonts = ["monospace" , ... , "sans-serif"];

CanvasElem = document.createElement("canvas");
CanvasElem.width = "100";
CanvasElem.height = "100";
context = CanvasElem.getContext('2d');
FPDict= {};
for (i = 0; i < Fonts.length; i++)
{
  CanvasElem.font = Fonts[i];
  FPDict[Fonts[i]] = context.measureText("example").width;
}
```

Script 5.2. An unpacked version of the script in Script 5.1.

Static feature extraction: Next, we generate static features from ASTs. ASTs have been extensively used in prior research to detect malicious JavaScript [172, 189, 215]. To build our features, we first hierarchically traverse the ASTs and divide them into pairs of parent and child nodes. Parents represents the context (e.g., `for` loops, `try` statements, or `if` conditions), and children represent the function inside that



(a) AST for packed Script 5.1



(b) AST for unpacked script 5.2

Figure 5.2. A truncated AST representation of Scripts 5.1 and 5.2. The edges represent the syntactic relationship between nodes. Dotted lines indicate an indirect connection through truncated nodes.

context (e.g., `createElement`, `toDataURL`, and `measureText`). Naively parsing `parent:child` pairs for the entire AST of every script would result in a prohibitively large number of features across all scripts (i.e., millions). To avoid this we only consider `parent:child` pairs that contain at least one keyword that matches a name, method, or property from one of the JavaScript APIs [67]. We assemble these `parent:child` combinations as feature vectors for all scripts. Each `parent:child` combination is treated as a binary feature, where 1 indicates the presence of a feature and 0 indicates its absence. Since we do not execute scripts in static analysis, fingerprinting-specific JavaScript API methods usually have only one occurrence in the script. Thus, we found the binary representation to sufficiently capture this information from the script.

As an example, feature extracted from AST in Figure 5.2b have `ForState-`

`ment:var` and `MemberExpression:measureText` as features which indicate the presence of a loop and access to `measureText` method. These methods are frequently used in canvas font fingerprinting scripts. Intuitively, fingerprinting script vectors have combinations of `parent:child` pairs that are specific to an API access pattern indicative of fingerprinting (e.g., setting a new font and measuring its width within a loop) that are unlikely to occur in non-fingerprinting scripts. A more comprehensive list of features extracted from the AST in Figure 5.2b are listed in Appendix 5.9.2 (Table 5.7).

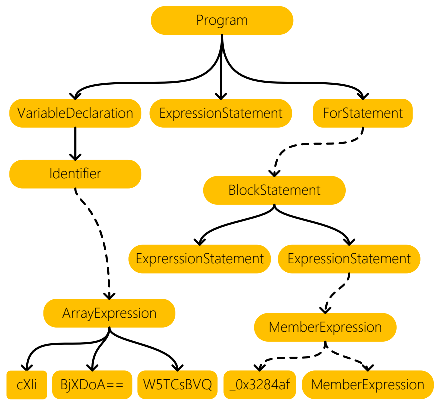
To avoid over-fitting, we apply unsupervised and supervised feature selection methods to reduce the number of features. Specifically, we first prune features that do not vary much (i.e., variance < 0.01) and also use information gain [219] to short list top-1K features. This allows us to keep the features that represent the most commonly used APIs for fingerprinting. For example, two of the features with the highest information gain represent the usage of `getSupportedExtensions` and `toDataURL` APIs. `getSupportedExtensions` is used to get the list of supported WebGL extensions, which vary depending on browser's implementation. `toDataURL` is used to get the base64 representation of the drawn canvas image, which depending on underlying hardware and OS configurations differs for the same canvas image. We then use these top-1K features as input to train a supervised machine learning model.

Dynamic analysis. Dynamic analysis complements some weaknesses of static analysis. While static analysis allows us to capture the syntactic structure of scripts, it fails when the scripts are obfuscated or minified. This is crucial because prior research has shown that fingerprinting scripts often use obfuscation to hide their functionality [291]. For example, Figure 5.3 shows an AST constructed from an obfuscated version of Script 5.2. The static features extracted from this AST would miss important `parent:child` pairs that are essential to capturing the script's functionality. Furthermore, some of the important `parent:child` pairs may be filtered during feature selection. Thus, in addition to extracting static features from script contents, we extract dynamic features by monitoring the execution of scripts. Execution traces capture the semantic relationship within scripts and thus provide additional context regarding a script's functionality, even when that script is obfuscated.

Dynamic feature extraction: We use two approaches to extract features from execution traces. First, we keep presence and count of the number of times a script accesses each individual API method or property and use that as a feature. Next, we build features from APIs that are passed arguments or return values. Rather than using the arguments or return values directly, we use derived values to capture a higher-level semantic that is likely to better generalize during classification. For example, we will compute the length of a string rather than including the exact text, or will compute the area of a element rather than including the height and width. This allows us to avoid training our classifier with overly specific features—i.e., we do not care whether the text "CanvasFingerprint" or "C4NV45F1NG3RPR1NT" is used during a canvas fingerprinting attempt, and instead only care about the text length and complexity. For concrete example, we calculate the area of canvas element, its text size, and whether its is present on screen when processing execution logs related

```
var _0x2c4a=['\x63\x58\x49\x69','\x42\x6a\x58\x44\x6f\x41\x3d\x3d','\x55\x54\x72
\x43\x69\x73\x4f\x77\x4f\x38\x4f\x6c\x50\x45\x6e\x43\x6d\x77\x30\x3d','\x49\x38
\x4f\x38\x49\x4d\x4f\x42\x77\x70\x72\x44\x6e\x41\x3d\x3d','\x77\x35\x54\x43\x73
\x42\x56\x51','\x77\x37\x62\x43\x69\x4d\x4f\x38\x77\x78..........................
..................................x3284af={};for(i=0x0;i<_0x1b2b65[_0x5d52
('0x7','\x28\x6d\x68\x26')];i++){_0x1d1d56[_0x5d52('0x8','\x67\x33\x48\x21')]
=_0x1b2b65[i];_0x3284af[_0x1b2b65[i]]=_0x4d24cc[_0x5d52('0x9','\x35\x70\x64\x4
c')](_0x5d52('0xa','\x28\x6d\x68\x26'))['\x77\x69\x64\x74\x68'];}
```

(a) Obfuscated canvas font fingerprinting script from Script 5.2.



(b) AST of the obfuscated script shown in (a).

Figure 5.3. A truncated example showing the AST representation of an obfuscated version of the canvas font fingerprinting script in Script 5.2. The edges represent the syntactic relationship between nodes. Dotted lines indicate an indirect connection through truncated nodes.

to `CanvasRenderingContext2D.fillText()`.

As an example, the features extracted from the execution trace of Script 5.3a includes (`HTMLCanvasElement.getContext, True`) and (`CanvasRenderingContext2D.measureTex` 7) as features, where `True` indicates the usage of `HTMLCanvasElement.getContext` and `7` indicates the size of text in `CanvasRenderingContext2D.measureText`. A more comprehensive list of features extracted from the execution trace of Script 5.3a can be found in Appendix 5.9.2 (Table 5.8).

To avoid over-fitting, we again apply unsupervised and supervised feature selection methods to limit the number of features. Similar to feature reduction for static analysis, this allows us to keep the features that represent the most commonly used APIs for fingerprinting. For example, two of the features with the highest information gain represent the usage of `CanvasRenderingContext2D.fillStyle` and `navigator.platform` APIs. `CanvasRenderingContext2D.fillStyle` is used to specify the color, gradient, or pattern inside a canvas shape, which can make a shape render differently across browsers and devices. `navigator.platform` reveals the platform (e.g. MacIntel and Win32) on which the browser is running. We then use these top-1K features as input to train a supervised machine learning model.

Classifying fingerprinting scripts. FP-Inspector uses a decision tree [275] classifier for training a machine learning model. The decision tree is passed feature vectors of scripts for classification. While constructing the tree, at each node, the decision tree chooses the feature that most effectively splits the data. Specifically, the attribute with highest information gain is chosen to split the data by enriching one class. The decision tree then follows the same methodology to recursively partition the subsets unless the subset belongs to one class or it can no longer be partitioned.

Note that we train two separate models and take the union of their classification results instead of combining features from both the static and dynamic representations of scripts to train a single model. That is, a script is considered to be a fingerprinting script if it is classified as fingerprinting by either the model that uses static features as input or the model that uses dynamic features as input. We use union of the two models because we only have the decision from one of the two models for some scripts (e.g., scripts that do not execute). Furthermore, the two models are already trained on high-precision ground truth [186] and taking the union would allow us to push for better recall. Using this approach, we classify all scripts loaded during a page visit—i.e., we include both external scripts loaded from separate URLs and inline scripts contained in any HTML document.

5.3.2 Mitigating fingerprinting scripts

Existing browser fingerprinting countermeasures can be classified into two categories: content blocking and API restriction. Content blocking, as the name implies, blocks the requests to download fingerprinting scripts based on their network location (e.g., domain or URL). API restriction, on the other hand, does not block fingerprinting scripts from loading but rather limits access to certain JavaScript APIs that are known to be used for browser fingerprinting.

Privacy-focused browsers such as the Tor Browser [44] prefer blanket API restriction over content blocking mainly because it side steps the challenging problem of detecting fingerprinting scripts. While API restriction provides reliable protection against active fingerprinting, it can break the functionality of websites that use the restricted APIs for benign purposes. Browsers that deploy API restriction also require additional protections against passive fingerprinting (e.g., routing traffic over the Tor network). Content blocking protects against both active and passive fingerprinting, but it is also prone to breakage when the detected script is dual-purpose (i.e., implements both fingerprinting and legitimate functionality) or a false positive.

Website breakage is an important consideration for fingerprinting countermeasures. For instance, a recent user trial by Mozilla showed that privacy countermeasures in Firefox can negatively impact user engagement due to website breakage [59]. In fact, website breakage can be the deciding factor in real-world deployment of any privacy-enhancing countermeasure [29, 70]. We are interested in studying the impact of different fingerprinting countermeasures based on FP-Inspector on website breakage. We implement the following countermeasures:

1. **Blanket API Restriction.** We restrict access for all scripts to the JavaScript APIs known to be used by fingerprinting scripts, hereafter referred to as "fingerprinting APIs". Fingerprinting APIs include functions and properties that are used in fingerprintjs2 and those discovered by FP-Inspector in Section 5.6. Note that this countermeasure does not at all rely on FP-Inspector's detection of fingerprinting scripts.

2. **Targeted API Restriction.** We restrict access to fingerprinting APIs only for the scripts served from domains that are detected by FP-Inspector to deploy fingerprinting scripts.

3. **Request Blocking.** We block the requests to download the scripts served from domains that are detected by FP-Inspector to deploy fingerprinting scripts.

4. **Hybrid.** We block the requests to download the scripts served from domains that are detected by FP-Inspector to deploy fingerprinting scripts, except for first-party and inline scripts. Additionally, we restrict access to fingerprinting APIs for first-party and inline scripts on detected domains. This protects against active fingerprinting by

first parties and both active and passive fingerprinting by third parties.

5.4 Evaluation

We evaluate FP-Inspector's performance in terms of its accuracy in detecting fingerprinting scripts and its impact on website breakage when mitigating fingerprinting.

5.4.1 Accuracy

We require samples of fingerprinting and non-fingerprinting scripts to train our supervised machine learning models. Up-to-date ground truth for fingerprinting is not readily available. Academic researchers have released lists of scripts [175, 186], however these only show a snapshot at the time of the work's publication and are not kept up-to-date. While many anti-tracking lists (e.g., EasyPrivacy) do include some fingerprinting domains, Disconnect's tracking protection list [31] is the only publicly available list that does not lump together different types of tracking and separately identifies fingerprinting domains. However, Disconnect's list is insufficient for our purposes. First, Disconnect's list only includes the domain names of companies that deploy fingerprinting scripts, rather than the actual URLs of the fingerprinting scripts. This prevents us from using the list to differentiate between fingerprinting and non-fingerprinting resources served from those domains. Second, the list appears to be focused on fingerprinting deployed by popular third-party vendors. Since first-party fingerprinting is also prevalent [175], we would like to train our classifier to detect both first- and third-party fingerprinting scripts. Given the limitations of these options, we choose to detect fingerprinting scripts using a slightly modified version of the heuristics implemented in [186].

5.4.1.1 Fingerprinting Definition

The research community is not aligned on a single definition to label fingerprinting scripts. It is often difficult to determine the *intent* behind any individual API access, and classifying all instances of device information collection as fingerprinting will result in a large number of false positives. For example, an advertisement script may collect a device's screen size to determine whether an ad was viewable and may never use that information as part of a fingerprint to identify the device. With that in mind, we take a conservative approach: we consider a script as fingerprinting if it uses `Canvas`, `WebRTC`, `Canvas Font`, or `AudioContext` as defined in [186]. Specifically, if the heuristics trigger for any of the above mentioned behaviors, we label the script as fingerprinting and otherwise label it as non-fingerprinting. We do not consider the collection of attributes from navigator or screen APIs from a script as fingerprinting, as these APIs are frequently used in non-distinct ways by scripts that do not fingerprint users. We decide to initially use this definition of fingerprinting because it is precise, i.e., it has a low false positive rate. A low false positive rate is crucial for a reliable ground truth as the classifiers effectiveness will depend on the soundness of ground truth. The exact details of heuristics are listed in Appendix 5.9.3.

5.4.1.2 Data Collection

We use our extended version of OpenWPM to crawl the homepages of twenty thousand websites sampled from the Alexa top-100K websites. To build this sample, we take the top-10K sites from the list and augment it with a random sample of 10K sites with Alexa ranks from 10K to 100K. This allows us to cover both the most popular websites as well as websites further down the long tail. During the crawl we allow each site 120 seconds to fully load before timing out the page visit. We store the HTTP response body content from all documents and scripts loaded on the page as well as the execution traces of all scripts.

Our crawled dataset consists of 17,629 websites with 153,354 distinct executing scripts. Since we generate our ground truth by analyzing script execution traces, we are only able to collect ground truth from scripts that actually execute during our crawl. Although we are not able train our classifier on scripts that do not execute during our crawl, we are still able to classify them. Their classification result will depend entirely on the static features extracted from the script contents. For static features, we successfully create ASTs for 143,526 scripts—9,828 scripts (6.4%) fail because of invalid syntax. Out of valid scripts, we extract a total of 47,717 `parent:child` combinations and do feature selection as described in Section 5.3. Specifically, we first filter by a variance threshold of 0.01 to reduce the set to 8,597 `parent:child` combinations. We then select top 1K features when sorted by information gain. For dynamic features, we extract a total of 2,628 features from 153,354 scripts. Similar to static analysis, we do feature selection as described in Section 5.3 and reduce the feature set to top 1K when sorted by information gain.

5.4.1.3 Enhancing Ground Truth

As discussed in Section 5.2, heuristics suffer from two inherent problems. First, heuristics are narrowly defined which can cause them to miss some fingerprinting scripts. Second, heuristics are predefined and are thus unable to keep up with evolving fingerprinting scripts. Due to these problems, we know that our heuristics-based ground truth is imperfect and a machine learning model trained on such a ground truth may perform poorly. We address these problems by enhancing the ground truth through iterative re-training. We first train a base model with incomplete ground truth, and then manually analyze the disagreements between the classifier's output and the ground truth. We update the ground truth whenever we find that our classifier makes a correct decision that was not reflected in the ground truth (i.e., discovers a fingerprinting script that was missed by the ground truth heuristics). We perform three iterations of this process.

Manual labeling. The manual process of analyzing scripts during iterative re-training works as follows. We automatically create a report for every script that requires manual analysis. Each report contains: (1) all of the API method calls and property accesses monitored by our instrumentation, including the arguments and return values, (2) snippets from the script that capture the surrounding context of calls to the APIs used for canvas, WebRTC, canvas font, and AudioContext fingerprinting, (3) a fingerprintjs2 similarity score,[3] and (4) the formatted contents of the

[3]We compute Jaccard similarity between the script, by first beautifying it and then

complete script. We then manually review the reports based on our domain expertise to determine whether the analyzed script is fingerprinting. Specifically, we look for heuristic-like behaviors in the scripts. The heuristic-like behavior means that the fingerprinting code in the script:

1. Is similar to known fingerprinting code in terms of its functionality and structure,
2. It is accompanied with other fingerprinting code (i.e. most fingerprinting scripts use multiple fingerprinting techniques), and
3. It does not interact with the functional code in the script.

For example, common patterns include sequentially reading values from multiple APIs, storing them in arrays or dictionaries, hashing them, and sending them in a network request without interacting with other parts of the script or page.

Findings. We found the majority of reviews to be straightforward—the scripts in question were often similar to known fingerprinting libraries and they frequently use APIs that are used by other fingerprinting scripts. If we find any fingerprinting functionality within the script we label the whole script as fingerprinting, otherwise we label it is non-fingerprinting. To be on the safe side, scripts for which we were unable to make a manual determination (e.g., due to obfuscation) were considered non-fingerprinting.

tokenizing it based on white spaces, and all releases of fingerprintjs2. The release with the highest similarity is reported along with the similarity score.

Itr.	Initial		New Detections		Correct Detections		Enhanced	
	FP	NON-FP	FP	NON-FP	FP	NON-FP	FP	NON-FP
S1	884	142,642	150	232	103	10	977	142,549
S2	977	142,549	109	182	84	5	1,056	142,470
S3	1,056	142,470	76	158	53	1	1,108	142,418
D1	928	152,426	11	52	4	9	923	152,431
D2	923	152,431	8	35	4	1	926	152,428
D3	926	152,428	13	36	5	2	929	152,425

Table 5.1. Enhancing ground truth with multiple iterations of retaining. Itr. represents the iteration number of training with static (S) and dynamic (D) models. New Detections (FP) represent the additional fingerprinting scripts detected by the classifier and New Detections (NON-FP) represent the new non-fingerprinting scripts detected by the classifier as compared to heuristics. Whereas Correct Detections (FP) represent the manually verified correct determination of the classifier for fingerprinting scripts and Correct Detections (NON-FP) represent the manually verified correct determination of the classifier for non-fingerprinting scripts.

Overall, perhaps expected, we find that our ground truth based on heuristics is high precision but low recall within the disagreements we analyzed. Most of the scripts that heuristics detect as fingerprinting do include fingerprinting code, but we also find that the heuristics miss some fingerprinting scripts. There are two major reasons scripts are missed. First, the fingerprinting portion of the script resides in a dormant part of the script, waiting to be called by other events or functions in a webpage. For example, the snippet in Script 5.3 (Appendix 5.9.4) defines fingerprinting-specific prototypes and assign them to a `window` object which can be called at a later point in time. Second, the fingerprinting functionality of the script deviates from the predefined heuristics. For example, the snippet in Script 5.4 (Appendix 5.9.4) calls `save` and `restore` methods on `CanvasRenderingContext2D` element, which are two method calls used by the heuristics to filter out non-fingerprinting scripts [186].

However, for a small number of scripts, the heuristics outperform the classifier. Scripts which make heavy use of an API used that is used for fingerprinting, and which have limited interaction with the webpage, are sometimes classified incorrectly. For example, we find cases where the classifier mislabels non-fingerprinting scripts that use the Canvas API to create animations and charts, and which only interact with a few HTML elements in the process. Since heuristics cannot generalize over fingerprinting behaviors, they do not classify partial API usage and limited interaction as fingerprinting. In other cases, the classifier labels fingerprinting scripts as non-fingerprinting because they include a single fingerprinting technique along with functional code. For example, we find cases where classifier mislabels fingerprinting scripts embedded on login pages that only include canvas font fingerprinting alongside functional code. Since heuristics are precise, they do not consider functional aspects of the scripts and do not classify limited usage of fingerprinting as non-fingerprinting.

Improvements. Table 5.1 presents the results of our manual evaluation for ground truth improvement for both static and dynamic analysis. It can be seen from the table that our classifier is usually correct when it classifies a script as fingerprinting in disagreement with the ground truth. We discover new fingerprinting scripts in each iteration. In addition, it is also evident from the table that our models are able to correct its mistakes with each iteration (i.e., correct previously incorrect non-fingerprinting classifications). This demonstrates the ability of classifier in iteratively detecting new fingerprinting scripts and correct mistakes as ground truth is improved. We further argue that this iterative improvement with re-training is essential for an operational deployment of a machine learning classifier and we empirically demonstrate that for FP-Inspector. Overall, we enhance our ground truth by labeling an additional 240 scripts as fingerprinting and 16 scripts as non-fingerprinting for static analysis, as well as 13 scripts as fingerprinting and 12 scripts as non-fingerprinting for dynamic analysis. In total, we detect 1,108 fingerprinting scripts and 142,418 non-fingerprinting scripts with static analysis and 929 fingerprinting scripts and 152,425 non-fingerprinting scripts using dynamic analysis.

5.4.1.4 Classification Accuracy

We use the decision tree models described in Section 5.3 to classify the crawled scripts. To establish confidence in our models against unseen scripts, we perform standard 10-fold cross validation. We determine the accuracy of our models by comparing the predicted label of scripts with the enhanced ground truth described in Section 5.4.1.3. For the model trained on static features, we achieve an accuracy of 99.8%, with 85.5% recall, and 92.7% precision. For the model trained on dynamic features, we achieve an accuracy of 99.9%, with 96.7% recall, and 99.1% precision.

Combining static and dynamic models. In FP-Inspector, we train two separate machine learning models—one using features extracted from the static representation of the scripts, and one using features extracted from the dynamic representation of the scripts. Both of the models provide complementary information for detecting fingerprinting scripts. Specifically, the model trained on static features identifies dormant scripts that are not captured by the dynamic representation, whereas the model trained on dynamic features identifies obfuscated scripts that are missed

Classifier	**Heuristics (Scripts/Websites)**	**Classifiers (Scripts/Websites)**	**FPR**	**FNR**	**Recall**	**Precision**	**Accuracy**
Static	884 / 2,225	1,022 / 3,289	0.05%	15.7%	85.5%	92.7%	99.8%
Dynamic	928 / 2,272	907 / 3,278	0.005%	5.3%	96.7%	99.1%	99.9%
Combined	935 / 2,272	1,178 / 3,653	0.05%	6.1%	93.8%	93.1%	99.9%

Table 5.2. FP-Inspector's classification results in terms of recall, precision, and accuracy in detecting fingerprinting scripts. "Heuristics (Scripts/Websites)" represents the number of scripts and websites detected by heuristics and "Classifiers (Scripts/Websites)" represents the number of scripts and websites detected by the classifiers. FPR represents false positive rate and FNR represent false negative rate.

by the static representation. We achieve the best of both worlds by combining the classification results of these models. We combine the models by doing an `OR` operation on the results of each model. Specifically, if either of the model detects a script as fingerprinting, we consider it a fingerprinting script. If neither of the model detects a script as fingerprinting, then we consider it a non-fingerprinting script. We manually analyze the differences in detection of static and dynamic models and find that the 94.46% of scripts identified only by the static model are partially or completely dormant and 92.30% of the scripts identified only by the dynamic model are obfuscated or excessively minified.

Table 5.2 presents the combined and individual results of static and dynamic models. It can be seen from the table that FP-Inspector's classifier detects 26% more scripts than the heuristics with a negligible false positive rate (FPR) of 0.05% and a false negative rate (FNR) of 6.1%. Overall, we find that by combining the models, FP-Inspector increases its detection rate by almost 10% and achieves an overall accuracy of 99.9% with 93.8% recall and 93.1% precision.[4]

[4]Is the complexity of a machine learning model really necessary? Would a simpler approach work as well? While our machine learning model performs well, we seek to answer this question in Appendix 5.9.5 by comparing our performance to a more straightforward similarity approach to detect fingerprinting. We compute the similarity between scripts and the popular fingerprinting library fingerprintjs2. Overall, we find that script similarity not only detects a partial number of fingerprinting scripts detected by our machine learning model but also incurs an unacceptably high number of false positives.

5.4.2 Breakage

We implement the countermeasures listed in Section 5.3.2 in a browser extension to evaluate their breakage. The browser extension contains the countermeasures as options that can be selected one at a time. For API restriction, we override functions and properties of fingerprinting APIs and return an error message when they are accessed on any webpage. For targeted API restriction, we extract a script's domain by traversing the stack each time the script makes a call to one of the fingerprinting APIs. We use FP-Inspector's classifier determinations to create a domain-level (eTLD+1, which matches Disconnect's fingerprinting list used by Firefox) filter list. For request blocking, we use the `webRequest` API [102] to intercept and block outgoing web requests that match our filter list [26].

Next, we analyze the breakage caused by these enforcements on a random sample of 50 websites that load fingerprinting scripts along with 11 websites that are reported as broken in Firefox due to fingerprinting countermeasures [47]. Prior research [216,294] has mostly relied on manual analysis to analyze website breakage due the challenges in automating breakage detection. We follow the same principles and manually analyze website breakage under the four fingerprinting countermeasures. To systemize manual breakage analysis, we create a taxonomy of common fingerprinting breakage patterns by going through the breakage-related bug reports on Mozilla's bug tracker [47]. We open each test website on vanilla Firefox (i.e., without our extension installed) as control and also with our extension installed as treatment. It is noteworthy that we disable Firefox's default privacy protections in both the control and treatment branches of our study to isolate the impact of our protections. We test each of the countermeasures one by one by trying to interact with the website for few minutes by scrolling through the page and using the obvious website functionality. If we discover missing content or broken website features only in the treatment group, we assign a breakage label using the following taxonomy:

1. **Major:** The core functionality of the website is broken. Examples include: login or registration flow, search bar, menu, and page navigation.
2. **Minor:** The secondary functionality of the website is broken. Examples include: comment sections, reviews, social media widgets, and icons.
3. **None:** The core and secondary functionalities of the website are the same in treatment and control. We consider missing ads as no breakage.

Policy	**Major (%)**	**Minor (%)**	**Total (%)**
Blanket API restriction	48.36%	19.67%	68.03%
Targeted API restriction	24.59%	5.73%	30.32%
Request blocking	44.26%	5.73%	50%
Hybrid	38.52%	8.19%	46.72%

Table 5.3. Breakdown of breakage caused by different countermeasures. The results present the average assessment of two reviewers.

To reduce coder bias and subjectivity, we asked two reviewers to code the breakage on the full set of 61 test websites using the aforementioned guidelines. The inter-coder reliability between our two reviewers is 87.70% for a total of 244 instances

(4 countermeasures × 61 websites). Table 5.3 summarizes the averaged breakage results. Overall, we note that targeted countermeasures that use FP-Inspector's detection reduce breakage by a factor of 2 on the tested websites that are particularly prone to breakage.[5] More specifically, blanket API restriction suffers the most (breaking more than two-thirds of the tested websites) while the targeted API restriction causes the least breakage (with no major breakage on about 75% of the tested websites).

Surprisingly, we find that the blanket API restriction causes more breakage than request blocking. We posit this is caused by the fact that blanket API restriction is applied to all scripts on the page, regardless of whether they are fingerprinting, since even benign functionality may be impacted. By comparison, request blocking only impacts scripts known to fingerprint. Next, we observe that targeted API restrictions has the least breakage. This is expected, as we do not block requests and only limit scripts that are suspected of fingerprinting; the functionality of benign scripts is not impacted.

We find that the hybrid countermeasure causes less breakage than request blocking but more breakage than the targeted API restrictions. The hybrid countermeasure performs better than request blocking because it does not block network requests to load first-party fingerprinting resources and instead applies targeted API restrictions to protect against first-party fingerprinting. Whereas it performs worse

[5]These websites employ fingerprinting scripts and/or are reported to be broken due to fingerprinting-specific countermeasures. Thus, they represent a particularly challenging set of websites to evaluate breakage by fingerprinting countermeasures.

than targeted API restrictions because it still blocks network requests to load third-party fingerprinting resources that are not blocked by the targeted API restrictions. Though hybrid blocking causes more breakage than targeted API restriction, it offers the best protection. Hybrid blocking mitigates both active and passive fingerprinting from third-party resources, and active fingerprinting from first-party resources and inline scripts. The only thing missed by hybrid blocking—passive first-party fingerprinting—is nearly impossible to block without breaking websites because any first-party resource loaded by the browser can passively collect device information.

We find that the most common reason for website breakage is the dependence of essential functionality on fingerprinting code. In severe cases, registration/login or other core functionality on a website depends on computing the fingerprint. For example, the registration page on `freelancer.com` is blank because we restrict the fingerprinting script from `f-cdn.com`. In less severe cases, websites embed widgets or ads that rely on fingerprinting code. For example, the social media widgets on `ucoz.ru/all/` disappears because we apply restrictions to the fingerprinting script from `usocial.pro`.

5.5 Measuring Fingerprinting In The Wild

Next, we use the detection component of FP-Inspector to analyze the state of fingerprinting on top-100K websites. To collect data from the Alexa top-100K websites, we first start with the 20K website crawl described in Section 5.4.1.2, and follow the same collection procedure for the remaining 80K websites not included in that measurement. Out of this additional 80K, we successfully visit 71,112 websites.

The results provide an updated view of fingerprinting deployment following the large-scale 2016 study by Englehardt and Narayanan [186]. On a high-level we find: (1) the deployment of fingerprinting is still growing—reaching over a quarter of the Alexa top-10K sites, (2) fingerprinting is almost twice as prevalent on news sites than in any other category of site, (3) fingerprinting is used for both anti-ad fraud and potential cross-site tracking.

5.5.1 Over a quarter of the top sites now fingerprint users

We first examine the deployment of fingerprinting across the top sites; our results are summarized in Table 5.4. In alignment with prior work [186], we find that fingerprinting is more prevalent on highly ranked sites. We also detect more fingerprinting than prior work [186], with over a quarter of the top sites now deploying fingerprinting. This increase in use holds true across all site ranks—we observe a notable increase even within less popular sites (i.e., 10K - 100K). Overall, we find that more than 10.18% of top-100K websites deploy fingerprinting.

We also find significantly more domains serving fingerprinting than past work—2,349 domains on the top 100K sites (Table 5.5) compared to 519 domains[6] on the top 1 million sites [186]. This suggests two things: our method is detecting a more comprehensive set of techniques than measured by Englehardt and Narayanan [186], and/or that the use of fingerprinting—both in prevalence and in the number of parties

[6]Englehardt and Narayanan [186] do not give an exact count of the number of domains serving fingerprinting across all measured techniques, and instead give a count for each individual fingerprinting technique. To get an upper bound on the total count, we assume there is no overlap between the reported results of each technique and take the sum.

involved—has significantly increased between 2016 and 2019.

Rank Interval	**Websites (count)**	**Websites (%)**
1 to 1K	266	30.60%
1K to 10K	2,010	24.45%
10K to 20K	981	11.10%
20K to 50K	2,378	8.92%
50K to 100K	3,405	7.70%
1 to 100K	9,040	10.18%

Table 5.4. Distribution of Alexa top-100K websites that deploy fingerprinting. Results are sliced by site rank.

5.5.2 Fingerprinting is most common on news sites

Fingerprinting is deployed unevenly across different categories of sites.[7] The difference is staggering—ranging from nearly 14% of news websites to just 1% of credit/debit related websites. Figure 5.4 summarizes our findings.

The distribution of fingerprinting scripts in Figure 5.4 roughly matches the distribution of trackers (i.e., not only fingerprinting, but any type of tracking) measured in past work [186]. One possible explanation of these results is that—like traditional tracking methods—fingerprinting is more common on websites that rely on advertising for monetization. Our results in Section 5.5.3 reinforce this interpretation, as the most prevalent vendors classified as fingerprinting provide anti-ad fraud and tracking services. The particularly high use of fingerprinting on news websites could also

[7]We use Webshrinker [103] for website categorization API.

point to fingerprinting being used as part of paywall enforcement, since cookie-based paywalls are relatively easy to circumvent [272].

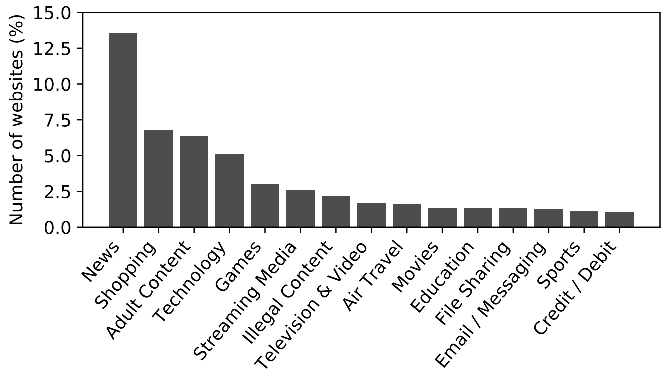


Figure 5.4. The deployment of fingerprinting scripts across different categories of websites.

5.5.3 Fingerprinting is used to fight ad fraud but also for potential cross-site tracking

Fingerprinting scripts detected by FP-Inspector are often served by third-party vendors. Three of the top five vendors in Table 5.5 (`doubleverify.com`, `adsafeprotected.com`, and `adsco.re`) specialize in verifying the authenticity of ad impressions. Their privacy policies mention that they use "device identification technology" that leverages "browser type, version, and capabilities" [10, 35, 62]. Our results also corroborate that bot detection services rely on fingerprinting [144], and indicate that prevalent fingerprinting vendors provide anti-ad fraud services. The two remaining vendors of the top five, i.e., `alicdn.com` and `yimg.com`, appear to be CDNs for Alibaba and Oath/Yahoo!, respectively.

Vendor Domain	**Tracker**	**Websites (count)**
doubleverify.com	Y	2,130
adsafeprotected.com	Y	1,363
alicdn.com	N	523
adsco.re	N	395
yimg.com	Y	246
2,344 others	Y(86)	5,702
Total		10,359 (9,040 distinct)

Table 5.5. The presence of the top vendors classified as fingerprinting on Alexa top-100K websites. Tracker column shows whether the vendor is a cross-site tracker according to Disconnect's tracking protection list. Y represents yes and N represents no.

Several fingerprinting vendors disclose using cookies "to collect information about advertising impression opportunities" [62] that is shared with "customers and partners to perform and deliver the advertising and traffic measurement services" [35]. To better understand whether these vendors participate in cross-site tracking, we first analyze the overlap of the fingerprinting vendors with Disconnect's tracking protection list [31].[8] Disconnect employs a careful manual review process [34] to classify a service as tracking. For example, Disconnect classifies c3tag as tracking [21,33] and adsco.re as not tracking [10, 32] because, based on their privacy policies, the former shares Personally Identifiable Information (PII) with its customers while the latter does not. We find that 3.78% of the fingerprinting vendors are classified as tracking by Disconnect.

We also analyze whether fingerprinting vendors engage in cookie syncing [271],

[8]We exclude the cryptomining and fingerprinting categories of the Disconnect list. The list was retrieved in June 2019.

which is a common practice by online advertisers and trackers to improve their coverage. For example, a tracker may associate browsing data from a single device to multiple distinct identifier cookies when cookies are cleared or partitioned. However, a fingerprinting vendor can use a device fingerprint to link those cookie identifiers together [183]. If the fingerprinting vendor had previously cookie synced with other trackers, it can use its fingerprint to link cookies for other trackers. We use the list by Fouad et al. [192] to identify fingerprinting domains that also participate in cookie syncing. We find that 17.28% of the fingerprinting vendors participate in cookie syncing. More importantly, we find that fingerprinting vendors often sync cookies with well-known ad-tech vendors. For example, `adsafeprotected.com` engages in cookie syncing with `rubiconproject.com` and `adnxs.com`. We also find that many fingerprinting vendors engage in cookie syncing with numerous third-parties. For example, `openx.net` engages in cookie syncing with 332 other domains, out of which 14 are classified as tracking by Disconnect. We leave an in-depth large-scale investigation of the interplay between fingerprinting and cookie syncing as future work.

5.6 Analyzing APIs used by Fingerprinters

In this section, we are interested in systematically investigating whether any newly proposed or existing JavaScript APIs are being exploited for browser fingerprinting. There are serious concerns that newly proposed or existing JavaScript APIs can be exploited in unexpected ways for browser fingerprinting [95].

We start off by analyzing the distribution of Javascript APIs in fingerprinting scripts. Specifically, we extract Javascript API keywords (i.e., API names, properties, and methods) from the source code of scripts and sort them based on the ratio of their fraction of occurrence in fingerprinting scripts to the fraction of occurrence in non-fingerprinting scripts. This ratio captures the relative prevalence of API keywords in fingerprinting scripts as compared to non-fingerprinting scripts. A higher value of the ratio for a keyword means that it is more prevalent in fingerprinting scripts than non-fingerprinting scripts. Note that ∞ means that the keyword is only present in fingerprinting scripts. Table 5.6 lists some of the interesting API keywords that are disproportionately prevalent in fingerprinting scripts. We note that some APIs are primarily used by fingerprinting scripts, including APIs which have been reported by prior fingerprinting studies (e.g., `accelerometer`) and those which have not (e.g., `getDevices`). We present a more comprehensive list of the API keywords disproportionately prevalent in fingerprinting scripts in Appendix 5.9.6.

Keywords	**Ratio**	**Scripts (count)**	**Websites (count)**
MediaDeviceInfo	∞	1	1363
magnetometer	∞	215	241
PresentationRequest	∞	16	16
onuserproximity	543.77	18	18
accelerometer	326.71	219	247
chargingchange	302.10	20	20
getDevices	187.62	59	80
maxChannelCount	184.44	29	40
baseLatency	181.26	3	8
vibrate	57.68	232	1793

Table 5.6. A sample of frequently used JavaScript API keywords in fingerprinting scripts and their presence on 20K websites crawl. Scripts (count) represents the number of distinct fingerprinting scripts in which the keyword is used and Websites (count) represents the number of websites on which those scripts are embedded.

Since the number of API keywords is quite large, it is practically infeasible to manually analyze all of them. Thus, we first group the extracted API keywords into a few clusters and then manually analyze the cluster which has the largest concentration of API keywords that are disproportionately used in the fingerprinting scripts detected by FP-Inspector. Our key insight is that browser fingerprinting scripts typically do not use a technique (e.g., canvas fingerprinting) in isolation but rather combine several techniques together. Thus, we expect fingerprinting-related API keywords to separate out as a distinct cluster.

To group API keywords into clusters, we first construct the co-occurrence graph of API keywords. Specifically, we model API keywords as nodes and include an edge between them that is weighted based on the frequency of co-occurrence in a script. Thus, co-occurring API keywords appear together in our graph representation. We then partition the API keyword co-occurrence graph into clusters by identifying strongly connected communities of co-occurring API keywords. Specifically, we extract communities of co-occurring keywords by computing the partition of the nodes that maximize the modularity using the Louvain method [150]. In total, we extract 25 clusters with noticeable dense cliques of co-occurring API keywords. To identify the clusters of interest, we assign the API keyword's fraction of occurrence in fingerprinting scripts to the fraction of occurrence in non-fingerprinting scripts as weights to the nodes. We further classify nodes based on whether they appear in fingerprintjs2 [45], which is a popular open-source browser fingerprinting library.

We investigate the cluster with the highest concentration of nodes that tend to appear in the detected fingerprinting scripts and those that appear in fingerprintjs2. While we discover a number of previously unknown uses of JavaScript APIs by fingerprinting scripts, for the sake of concise discussion, instead of individually listing all of the previously unknown JavaScript API keywords, we thematically group them. We discuss how each new API we discover to be used by fingerprinting scripts may be abused to extract identifying information about the user or their device. While our method highlights *potential* abuses, a deep manual analysis of each script is required to confirm abuse.

Functionality fingerprinting. This category covers browser fingerprinting techniques that probe for different functionalities supported by the browser. Modern websites rely on many APIs to support their rich functionality. However, not all browsers support every API or may have the requisite user permission. Thus, websites may need to probe for APIs and permissions to adapt their functionality. However, such feature probing can potentially leak entropy.

1. *Permission fingerprinting:* `Permissions` API provides a way to determine whether a permission is granted or denied to access a feature or an API. We discover several cases in which the `Permissions` API was used in fingerprinting scripts. Specifically, we found cases where the status and permissions for APIs such as `Notification`, `Geolocation`, and `Camera` were probed. The differences in permissions across browsers and user settings can be used as part of a fingerprint.

2. *Peripheral fingerprinting:* Modern browsers provide interfaces to communicate with external peripherals connected with the device. We find several cases in which

peripherals such as gamepads and virtual reality devices were probed. In one of the examples of peripherals probing, we find a case in which keyboard layout was probed using `getLayoutMap` function. The layout of the keyboard (e.g., size, presence of specific keys, string associated with specific keys) varies across different vendors and models. The presence and the various functionalities supported by these peripherals can potentially leak entropy.

3. *API fingerprinting:* All browsers expose differing sets of features and APIs to the web. Furthermore, some browser extensions override native JavaScript methods. Such implementation inconsistencies in browsers and modifications by user-installed extensions can potentially leak entropy [284]. We find several cases in which certain functions such as `AudioWorklet` were probed by fingerprinting scripts. `AudioWorklet` is only implemented in Chromium-based browsers (e.g., Chrome or Opera) starting version 66 and its presence can be probed to check the browser and its version. We also find several cases where fingerprinting scripts check whether certain functions such as `setTimeout` and `mozRTCSessionDescription` were overridden. Function overriding can also leak presence of certain browser extensions. For example, Privacy Badger [76] overrides several prototypes of functions that are known to be used for fingerprinting.

Algorithmic fingerprinting. This category covers browser fingerprinting techniques that do not just simply probe for different functionalities. These browser fingerprinting techniques algorithmically process certain inputs using different JavaScript APIs and exploit the fact that different implementations process these inputs differently to leak entropy. We discuss both newly discovered uses of JavaScript APIs that were previously not observed in fingerprinting scripts *and* known fingerprinting techniques that seem to have evolved since their initial discovery.

1. *Timing fingerprinting:* The `Performance` API provides high-resolution timestamps of various points during the life cycle of loaded resources and it can be used in various ways to conduct timing related fingerprinting attacks [83, 282]. We find several instances of fingerprinting scripts using the `Performance` API to record timing of all its events such as `domainLookupStart`, `domainLookupEnd`, `domInteractive`, and `msFirstPaint`. Such measurements can be used to compute the DNS lookup time of a domain, the time to interactive DOM, and the time of first paint. A small DNS lookup time may reveal that the URL has previously been visited and thus can leak the navigation history [83], whereas time to interactive DOM and time to first paint for a website may vary across different browsers and different underlying hardware configurations. Such differences in timing information can potentially leak entropy.

2. *Animation fingerprinting:* Similar to timing fingerprinting, we found fingerprinting scripts using `requestAnimationFrame` to compute the frame rate of content rendering in a browser. The browser guarantees that it will execute the callback function passed to `requestAnimationFrame` before it repaints the view. The browser callback rate generally matches the display refresh rate [81] and the number of callbacks within an interval can capture the frame rate. The differences in frame rates can potentially leak entropy.

3. *Audio fingerprinting:* Englehardt and Narayanan [186] first reported the audio fin-

gerprinting technique that uses the `AudioContext` API. Specifically, the audio signal generated with `AudioContext` varies across devices and browsers. Audio fingerprinting seems to have evolved. We identify several cases in which fingerprinting scripts used the `AudioContext` API to capture additional properties such as `numberOfInputs`, `numberOfOutputs`, and `destination` among many others properties. In addition to reading `AudioContext` properties, we also find cases in which `canPlayType` is used to extract the audio codecs supported by the device. This additional information exposed by the `AudioContext` API can potentially leak entropy.

4. *Sensors fingerprinting:* Prior work has shown that the device sensors can be abused for browser fingerprinting [152, 175, 178]. We find several instances of previously known and unknown sensors being used by fingerprinting scripts. Specifically, we find previously known sensors [175] such as `devicemotion` and `deviceorientation` and, more importantly, previously unknown sensors such as `userproximity` being used by fingerprinting scripts.

5.7 Limitations

In this section, we discuss some of the limitations of FP-Inspector's detection and mitigation components. Since FP-Inspector detects fingerprinting at the granularity of a script, an adversarial website can disperse fingerprinting scripts into several chunks to avoid detection or amalgamate all scripts—functional and fingerprinting—into one to avoid enforcement of mitigation countermeasures.

Evading detection through script dispersion. For detection, FP-Inspector only considers syntactic and semantic relationship within scripts and does not considers relationship across scripts. Because of its current design, FP-Inspector may be challenged in detecting fingerprinting when the responsible code is divided across several scripts. However, FP-Inspector can be extended to capture interaction among scripts by more deeply instrumenting the browser. For example, prior approaches such as AdGraph [216] and JSGraph [243] instrument browsers to capture cross-script interaction. Future versions of FP-Inspector can also implement such instrumentation; in particular, FP-Inspector can be extended to capture the parent-child relationships of script inclusion. To avoid trivial detection through parent-child relationships, the script dispersion technique would need to be embed each chunk into a website from an independent ancestor node, and return the results to seemingly independent servers. Thus, script dispersion also has a maintenance cost: each update to the fingerprinting script will require the distribution of script into several chunks along with extensive testing to ensure correct implementation.

Evading countermeasures through script amalgamation. To restrict fingerprinting, FP-Inspector's most effective countermeasure (i.e. targeted API restriction) is applied at the granularity of a script. FP-Inspector may break websites where all of the scripts are amalgamated in a single script. However, more granular enforcement can be used to effectively prevent fingerprinting in such cases. For example, the instrumentation used by future versions of FP-Inspector can be extended to track the execution of callbacks and target those related to fingerprinting. It is noteworthy that—similar to script dispersion—script amalgamation has a maintenance cost: each update to any of the script will require the amalgamation of all scripts into one. Script amalgamation could also be used as a countermeasure against ad and tracker blockers, which would introduce the same type of breakage. However, anecdotal evidence suggests that the barriers to use are sufficiently high to prevent widespread deployment of amalgamation as a countermeasure against privacy tools.

5.8 Conclusion

We presented FP-Inspector, a machine learning based syntactic-semantic approach to accurately detect browser fingerprinting behaviors. FP-Inspector outperforms heuristics from prior work by detecting 26% more fingerprinting scripts and helps reduce website breakage by 2X. FP-Inspector's deployment showed that browser fingerprinting is more prevalent on the web now than ever before. Our measurement study on the Alexa top-100K websites showed that fingerprinting scripts are deployed on 10.18% of the websites by 2,349 different domains.

We plan to report the domains serving fingerprinting scripts to tracking protection lists such as Disconnect [31] and EasyPrivacy [39]. FP-Inspector also helped uncover exploitation of several new APIs that were previously not known to be used for browser fingerprinting. We plan to report the names and statistics of these APIs to privacy-oriented browser vendors and standards bodies. To foster follow-up research, we will release our patch to OpenWPM, fingerprinting countermeasures prototype extension, list of newly discovered fingerprinting vendors, and bug reports submitted to tracking protection lists, browser vendors, and standards bodies at `https://uiowa-irl.github.io/FP-Inspector`.

5.9 Appendix

5.9.1 Extensions to OpenWPM JavaScript instrumentation

OpenWPM's instrumentation does not cover a number of APIs used for fingerprinting by prominent libraries—including the Web Graphics Library (WebGL) and `performance.now`. These APIs have been discovered to be fingerprintable [232]. The standard use case of WebGL is to render 2D and 3D graphics in HTML canvas element, however, it has potential to be abused for browser fingerprinting. The WebGL renderer and vendor varies by the OS and it creates near distinct WebGL images with same configurations on different machines. The WebgGL properties and the rendered image are used by current state-of-the-art browser fingerprinting [45, 68] scripts. Since WebGL is used by popular fingerprinting scripts, we instrument WebGL JavaScript API. `performance.now` is another JavaScript API method whose standard use case is to return time in floating point milliseconds since the start of a page load but it also have fingerprinting potential. Specifically, the timing information extracted from `performance.now` can be used for timing specific fingerprint attacks such as typing cadence [49, 89]. We extend OpenWPM to also capture execution of `performance.now`.

For completeness, we instrument additional un-instrumented methods of already instrumented JavaScript APIs in OpenWPM. Specifically, we enhance our execution trace by instrumenting methods such as `drawImage` and `sendBeacon` for `canvas` and `navigation` JavaScript APIs, respectively.

Since most fingerprinting scripts use JavaScript APIs that are also used by gaming and interactive websites (e.g. `canvas`), we instrument additional JavaScript APIs to capture script's interaction with DOM. Specifically, to capture DOM interaction specific JavaScript APIs, we instrument `document`, `node`, and `animation` APIs. JavaScript is an event driven language and it has capability to execute code when events trigger. To extend our execution trace, we instrument JavaScript events such as `onmousemove` and `touchstart` to capture user specific interactions.

In addition, we notice that some scripts make multiple calls to JavaScript API methods such as `createElement` and `setAttribute` during their execution. We limit our recording to only first 50 calls of each method per script, except for `CanvasRenderingContext2D.measureText` and `CanvasRenderingContext2D.font`, which are called multiple times for canvas font fingerprinting. Furthermore, the event driven nature of JavaScript makes it challenging to capture the complete execution trace of scripts. To this end, to get a comprehensive execution of a script, we synthetically simulate user activity on a webpage. First, we scroll the wbepage from top to bottom and do random mouse movements to trigger events. Second, we record all of the events (e.g. `onscroll`) as they are registered on different elements on a webpage and execute them after 10 seconds of a page load. Doing so, we synthetically simulate events and capture JavaScript API methods that were waiting for those events to trigger.

5.9.2 Sample Features Extracted From ASTs & Execution Traces

Table 5.7 shows a sample of the features extracted from the AST in Figure 5.2b and Table 5.8 shows a sample of the dynamic features extracted from execution trace of Script 5.3a.

Static Features
ArrayExpression:monospace
MemberExpression:font
ForStatement:var
MemberExpression:measureText
MemberExpression:width
MemberExpression:length
MemberExpression:getContext
CallExpression:canvas

Table 5.7. A sample of features extracted from AST in Figure 5.2b.

5.9.3 Fingerprinting Heuristics

Below we list down the slightly modified versions of heuristics proposed by Englehardt and Narayanan [186] to detect fingerprinting scripts. Since non-fingerprinting adoption of fingerprinting APIs have increased since the study, we make modifications to the heuristics to reduce the false positives. These heuristics are used to build our initial ground truth of fingerprinting and non-fingerprinting scripts.

Canvas Fingerprinting. A script is identified as canvas fingerprinting script according to the following rules:

Feature Name	**Feature Value**
Document.createElement	True
HTMLCanvasElement.width	True
HTMLCanvasElement.height	True
HTMLCanvasElement.getContext	True
CanvasRenderingContext2D.measureText	True
Element Tag Name	Canvas
HTMLCanvasElement.width	100
HTMLCanvasElement.height	100
CanvasRenderingContext2D.measureText	7 (no. of chars.)
CanvasRenderingContext2D.measureText	N (no. of calls)

Table 5.8. A sample of the dynamic features extracted from the execution trace of Script 5.3a.

1. The canvas element text is written with `fillText` or `strokeText` and style is applied with `fillStyle` or `strokeStyle` methods of the rendering context.

2. The script calls `toDataURL` method to extract the canvas image.

3. The script does not calls `save`, `restore`, and `addEventListener` methods on the canvas element.

WebRTC Fingerprinting. A script is identified as WebRTC fingerprinting script according to the following rules:

1. The script calls `createDataChannel` or `createOffer` methods of the WebRTC peer connection.

2. The script calls `onicecandidate` or `local Description` methods of the WebRTC peer connection.

Canvas Font Fingerprinting. A script is identified as canvas font fingerprinting script according to the following rules:

1. The script sets the `font` property on a canvas element to more than 20 different fonts.

2. The script calls the `measureText` method of the rendering context more than 20 times.

AudioContext Fingerprinting. A script is identified as AudioContext fingerprinting script according to the following rules:

1. The script calls any of the `createOscillator`, `createDynamicsCompressor`, `destination`, `start Rendering`, and `oncomplete` method of the audio context.

5.9.4 Examples of Dormant and Deviating Scripts

Script 5.3 shows an example dormant script and Script 5.4 shows an example deviating script.

5.9.5 Why Machine Learning?

To conduct fingerprinting, websites often embed off-the-shelf third-party fingerprinting libraries. Thus, one possible approach to detect fingerprinting scripts is to simply compute the textual similarity between the known fingerprinting libraries and the scripts embedded on a website. Scripts that have higher similarity with known fingerprinting libraries are more likely to be fingerprinting scripts. To test this hypothesis, we compare the similarity of fingerprinting and non-fingerprinting

```
(function(g) {
......
n.prototype = {
  getCanvasPrint: function() {
  var b = document.createElement("canvas"),d;
  try {
      d = b.getContext("2d")
  } catch (e) {
      return ""
  }
  d.textBaseline = "top";
  d.font = "14px 'Arial'";
  ...
  d.fillText("http://valve.github.io", 4, 17);
  return b.toDataURL()
  }
};
"object" === typeof module &&
    "undefined" !== typeof exports && (module.exports = n);
g.ClientJS = n
})(window);
```

Script 5.3. A truncated example of a dormant script from `sdk1.resu.io/scripts/resclient.min.js` in which function prototypes are assigned to the `window` object and can be called at a later point in time.

scripts detected by FP-INSPECTOR against fingerprintjs2, a popular open-source fingerprinting library. Specifically, we tokenize scripts into keywords by first beautifying them and then splitting them on white spaces. We then compute a tokenized script's Jaccard similarity, pairwise, with all versions of fingerprintjs2. The highest similarity score among all versions is attributed to a script.

Our test set consists of the fingerprinting scripts detected by FP-INSPECTOR and an equal number of randomly sampled non-fingerprinting scripts. Figure 5.5, plots the similarity of FP-INSPECTOR's detected fingerprinting and non-fingerprinting scripts with fingerprintjs2. We find that the majority of the detected fingerprinting scripts (54.06%) have less than 6% similarity to fingerprintjs2 and only 13.49% of the scripts have more than 30% similarity. Whereas most of the detected non-fingerprinting scripts (90.94%) have less than 5% similarity to fingerprintjs2 and only 9.05% of the

```
...
canvas: function(t) {
var e = document.createElement("canvas");
if ("undefined" == typeof e.getContext)
t.push("UNSUPPORTED_CANVAS");
else {
e.width = 780, e.height = 150;
var n = "UNICODE STRING", i = e.getContext("2d");
i.save(), i.rect(0,0,10,10), i.rect(2,2,6,6), t.push(!1 === i.isPointInPath(5, 5,
    "evenodd")
? "yes" : "no"), i.restore(), i.save();
var r = i.createLinearGradient(0, 0, 200, 0);
.....
i.shadowColor="rgb(85,85,85)",i.shadowBlur=3 , i.arc(500,15,10,0,2*Math.PI,!0),
    i.stroke(),
i.closePath(),i.restore(),t.push(e.toDataURL())
}
return t
}
...
```

Script 5.4. A truncated example of a deviating script from `webresource.c-ctrip.com/code/ubt/_bfa.min.js?v=20195_22.js`. The heuristic is designed to ignore scripts that call `save` or `restore` on `CanvasRenderingContext2D` as a way to reduce false positives.

scripts have more than 5% similarity. We find that the true positive rate is at the highest (69.20%) and false positive rate is at the lowest (5.97%) with an accuracy of 81.69%, when we set the similarity threshold to 5.28%. The shaded portion of the figure represents the scripts classified as non-fingerprinting and the clear portion of the figure represents the scripts classified as fingerprinting using this threshold. There is a significant overlap between the similarity of both fingerprinting and non-fingerprinting scripts and there is no optimal way to use similarity as a classification threshold.

Overall, our analysis shows that most websites do not integrate fingerprinting libraries as-is but instead make alterations. Alterations often include embedding minified or obfuscated versions of the fingerprinting libraries, embedding only a subset of the fingerprinting functionality, or fingerprinting libraries inspired re-implementation. Such alterations cause a lower similarity between fingerprinting scripts and popular

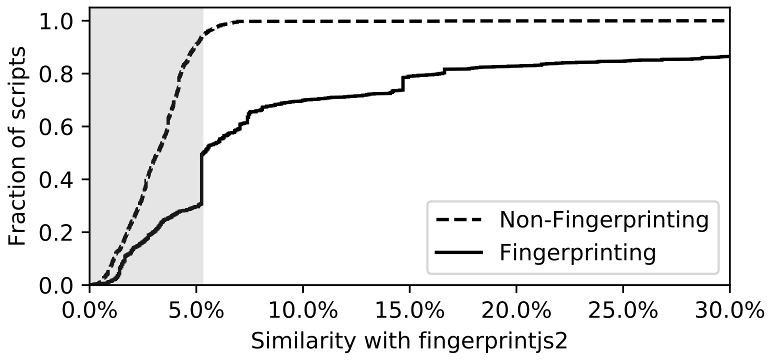


Figure 5.5. Jaccard similarity of fingerprinting and non-fingerprinting scripts with fingerprintjs2. The shaded portion of the figure represents the scripts classified as non-fingerprinting and the clear portion of the figure represents the scripts classified as fingerprinting based on the similarity threshold.

fingerprinting libraries. We also find that several APIs are frequently used in both fingerprinting and non-fingerprinting scripts. Common examples include the use of utility APIs such as Math and window, and non-fingerprinting scripts using fingerprinting APIs for functional purposes e.g. canvas API being used for animations. The presence of such APIs results in increase of similarity between non-fingerprinting scripts and fingerprinting libraries. A simple similarity metric cannot generalize on alterations to fingerprinting libraries and functional uses of APIs, and thus fails to detect fingerprinting scripts. Whereas, our syntactic-semantic machine learning approach is able to generalize. Our analysis justifies the efficacy of a learning based approach over simple similarity metric.

5.9.6 JavaScript APIs Frequently Used in Fingerprinting Scripts

Below we provide a list of JavaScript API keywords frequently used by fingerprinting scripts. To this end, we measure the relative prevalence of API keywords in fingerprinting scripts by computing the ratio of their fraction of occurrence in fingerprinting scripts to their fraction of occurrence in non-fingerprinting scripts. A higher value of the ratio for a keyword means that it is more prevalent in fingerprinting scripts than non-fingerprinting scripts. Note that ∞ means that the keyword is only present in fingerprinting scripts. Table 5.9 includes keywords that have pervasiveness values greater than or equal to 16 and are present on 3 or more websites.

Keywords	**Ratio**	**Scripts** (count)	**Websites** (count)
onpointerleave	∞	4	1366
StereoPannerNode	∞	1	1363
FontFaceSetLoadEvent	∞	1	1363
PresentationConnection AvailableEvent	∞	1	1363
msGetRegionContent	∞	1	1363
peerIdentity	∞	1	1363
MSManipulationEvent	∞	1	1363
VideoStreamTrack	∞	1	1363
mozSetImageElement	∞	1	1363
requestWakeLock	∞	1	174
audioWorklet	∞	3	8
onwebkitanimationiteration	∞	3	3
onpointerenter	∞	3	3
onwebkitanimationstart	∞	3	3
onlostpointercapture	∞	3	3
ongotpointercapture	362.52	3	3
onpointerout	362.52	3	3
onafterscriptexecute	217.51	18	1380
channelCountMode	199.03	28	39
onpointerover	181.26	3	3
onbeforescriptexecute	181.26	18	1380
onicegatheringstatechange	179.78	61	61
MediaDevices	161.12	4	1366
numberOfInputs	157.09	26	36
channelInterpretation	147.69	11	22
speedOfSound	140.98	7	11
dopplerFactor	140.98	7	11
midi	138.72	225	251

ondeviceproximity	131.35	25	282
HTMLMenuItemElement	121.40	218	244
updateCommands	120.84	1	1363
exportKey	105.97	57	57
onauxclick	90.63	3	3
microphone	90.43	223	250
iceGatheringState	90.30	68	1481
ondevicelight	88.31	19	36
renderedBuffer	87.17	189	439
WebGLContextEvent	82.52	28	44
ondeviceorientationabsolute	80.56	4	1366
startRendering	79.33	193	458
createOscillator	78.77	191	445
knee	76.65	170	419
OfflineAudioContext	74.68	199	721
timeLog	72.50	12	12
getFloatFrequencyData	72.50	6	10
WEBGL_compressed_texture_atc	72.50	3	4
illuminance	72.50	3	3
reduction	69.64	170	419
modulusLength	69.39	58	58
WebGL2RenderingContext	68.71	29	30
enumerateDevices	64.12	208	666
AmbientLightSensor	63.60	10	267
attack	61.31	173	434
AudioWorklet	60.42	22	32
Worklet	60.42	22	32
AudioWorkletNode	60.42	22	32
lastStyleSheetSet	60.42	1	1363
DeviceProximityEvent	60.42	1	1363
DeviceLightEvent	60.42	1	1363
enableStyleSheetsForSet	60.42	1	1363
UserProximityEvent	60.42	1	1363
mediaDevices	60.03	230	850
vendorSub	56.17	251	1728
setValueAtTime	55.29	167	417
getChannelData	55.18	195	460
MAX_DRAW_BUFFERS_WEBGL	54.93	10	12
reliable	52.36	39	103
WEBGL_draw_buffers	52.09	25	27
EXT_sRGB	51.79	3	4
setSinkId	50.35	5	1367
namedCurve	50.29	67	74
WEBGL_debug_shaders	45.31	3	4

productSub	42.79	734	2819
hardwareConcurrency	41.92	716	3661
publicExponent	41.52	67	74
requestMIDIAccess	40.28	1	1363
mozIsLocallyAvailable	40.28	1	174
ondevicemotion	40.28	4	4
XPathResult	39.73	218	417
mozBattery	39.04	42	322
IndexedDB	38.73	25	25
generateKey	37.46	62	62
buildID	36.52	272	414
getSupportedExtensions	36.46	534	1007
MAX_TEXTURE_MAX_ ANISOTROPY_EXT	35.85	521	980
oscpu	35.33	681	1196
oninvalid	34.75	65	1428
vpn	34.53	24	24
createDynamicsCompressor	33.54	189	442
privateKey	33.46	67	74
EXT_texture_filter_anisotropic	32.91	479	949
isPointInPath	32.17	481	949
getContextAttributes	31.76	460	920
BatteryManager	31.23	23	50
getShaderPrecisionFormat	30.81	450	915
depthFunc	30.81	452	921
uniform2f	30.71	460	930
rangeMax	30.36	449	902
rangeMin	30.24	446	897
EXT_disjoint_timer_query	30.21	3	4
scrollByPages	30.21	1	1363
CanvasCaptureMediaStreamTrack	30.21	1	18
onlanguagechange	30.21	4	4
clearColor	29.16	457	916
createWriter	28.93	17	17
getUniformLocation	28.61	466	948
getAttribLocation	28.58	464	945
drawArrays	28.53	466	948
useProgram	28.37	467	949
enableVertexAttribArray	28.37	466	948
createShader	28.31	467	949
compileShader	28.30	467	936
shaderSource	28.27	466	936
attachShader	28.25	464	934
bufferData	28.24	466	938

linkProgram	28.23	464	933
vertexAttribPointer	28.22	464	933
bindBuffer	28.14	463	932
createProgram	27.95	464	934
OES_standard_derivatives	27.46	20	1384
appCodeName	27.03	325	1890
getAttributeNodeNS	26.49	16	21
ARRAY_BUFFER	25.36	471	941
suffixes	25.14	775	1441
TouchEvent	25.01	481	1130
MIDIPort	24.17	2	19
onaudioprocess	23.64	9	17
showModalDialog	23.56	39	1419
globalStorage	23.48	245	1681
camera	22.76	229	255
onanimationiteration	22.66	3	3
textBaseline	21.76	888	3234
MediaStreamTrackEvent	21.32	3	1365
deviceproximity	21.13	25	26
taintEnabled	20.89	14	24
alphabetic	20.65	671	2986
userproximity	20.28	24	25
globalCompositeOperation	20.15	507	975
outputBuffer	20.14	12	34
WebGLUniformLocation	20.14	1	1363
WebGLShaderPrecisionFormat	20.14	1	1363
createScriptProcessor	20.14	11	20
createBuffer	19.98	472	954
UIEvent	19.93	47	63
toSource	19.54	416	2224
createAnalyser	19.33	12	17
fillRect	19.22	898	3432
evenodd	18.49	504	960
fillText	18.09	957	3502
candidate	18.03	178	1847
WEBGL_debug_renderer_info	17.83	406	2214
toDataURL	17.64	951	3507
dischargingTime	17.53	38	54
bluetooth	17.28	225	424
FLOAT	16.89	467	939
battery	16.82	152	1853
devicelight	16.51	25	26
onanimationstart	16.48	3	3
getExtension	16.43	575	1115

onemptied	16.11	4	4

Table 5.9. JavaScript API keywords frequently used in fingerprinting scripts, and their presence on 20K websites crawl. Scripts (count) represents the number of distinct fingerprinting scripts in which the keyword is used and Websites (count) represents the number of websites on which those scripts are embedded.

CHAPTER 6

THE AD WARS: RETROSPECTIVE MEASUREMENT AND ANALYSIS OF ANTI-ADBLOCK FILTER LISTS

6.1 Introduction

Millions of users around the world use adblockers on desktop and mobile devices [74]. Users employ adblockers to get rid of intrusive and malicious ads as well as improve page load performance and protect their privacy. Since online publishers primarily rely on ads to monetize their services, they cannot monetize their services if a user employs an adblocker to remove ads. Online publishers have lost billions of dollars in advertising revenues due to adblocking [248]. Online publishers use two strategies to recoup lost advertising revenues.

First, many online publishers and advertisers have become part of the acceptable ads program [2], which allows their ads to be whitelisted if they conform to the acceptable ads guidelines. Small- and medium-sized publishers can enroll in the acceptable ads program for free, however, large publishers need to pay about 30% of the additional revenue recouped by whitelisting of acceptable ads. Popular adblockers, such as Adblock Plus, use the acceptable ads filter list to whitelist acceptable ads. While some stakeholders in the advertising ecosystem think that the acceptable ads program offers a reasonable compromise for users and publishers, there are lingering concerns about the acceptable ads criteria and the transparency of the whitelisting process [307].

Second, many online publishers have started to interrupt adblock users by employing anti-adblock scripts. These anti-adblock scripts allow publishers to detect adblock users and respond by showing notifications that ask users to disable their adblocker altogether, whitelist the website, or make a donation to support them. Most publishers rely on third-party anti-adblock scripts provided by vendors such as PageFair and Outbrain as well as community scripts provided by the IAB and BlockAdblock.

Adblockers employ anti-adblock filter lists to remove anti-adblock scripts and notifications by anti-adblockers. Similar to the filter lists such as EasyList and EasyPrivacy for blocking ads and trackers respectively, there are several crowdsourced anti-adblock filter lists that are used by adblockers to circumvent anti-adblokcers. These anti-adblock filter lists represent the state-of-the-art solution to anti-adblockers for adblock users, but little is known about the origin, evolution, and effectiveness of anti-adblock filter lists in circumventing anti-adblockers.

To fill this gap, in this chapter we present the first comprehensive study of anti-adblock filter lists. We analyze the complete history of crowdsourced anti-adblock filter lists and conduct a retrospective measurement study to analyze their coverage over time. We also develop a lightweight machine learning based approach to automatically detect anti-adblock scripts using static JavaScript code analysis. Our approach can complement crowdsourcing to speed up the creation of new filter rules as well as improve the overall coverage of filter lists.

We address three research questions in this chapter.

1. *How have the filter rules in anti-adblock filter lists evolved over time?* We analyze two anti-adblock filter lists, Anti-Adblock Killer List and Combined EasyList (Adblock Warning Removal List + Anti-Adblock sections in EasyList). Our analysis reveals that they are implemented differently. More specifically, the Combined EasyList uses a few broadly defined filter rules and then uses many more exception rules to take care of false positives. In contrast, the Anti-Adblock Killer List tends to contain high precision filter rules that target specific websites. We also find that while the Combined EasyList is updated almost daily, the Anti-Adblock Killer List is being updated approximately once every month for the last one year.

2. *How has the coverage of anti-adblock filter lists evolved over time?* We test different versions of anti-adblock filter lists on historical snapshots of Alexa top-5K websites archived by the Wayback Machine to study their coverage. We find that the Anti-Adblock Killer List triggers on 8.7% websites and the Combined EasyList only triggers on 0.4% websites currently. We further test both anti-adblock filter lists on Alexa top-100K live websites. We find that the Anti-Adblock Killer List triggers on 5.0% websites and the Combined EasyList only triggers on 0.2% websites. While the Anti-Adblock Killer List clearly seems to provide better coverage than the Combined EasyList, it has not been updated by its authors since November 2016.

3. *How can we improve creation of anti-adblock filter lists?* To aid filter list au-

thors in maintaining anti-adblock filter lists, we investigate a machine learning based automated approach to identify anti-adblock scripts. Our method conducts static analysis of JavaScript code to fingerprint syntactic features of anti-adblock scripts. The evaluation shows that our method achieves up to 99.7% detection rate and 3.2% false positive rate. Our proposed machine learning based automated approach can aid filter list authors to more quickly update anti-adblock filter lists as well as improve its coverage.

Our work is motivated by recent studies of different filter lists [186, 255, 307] that are used by adblockers. Our findings highlight inherent limitations of manual curation and informal crowdsouricng to maintain anti-adblock filter lists. Our work can help filter list authors to automatically and quickly update anti-adblock filter lists.

Chapter Organization. The rest of the chapter is organized as follows. §6.2 provides a brief background and discussion of related work on anti-adblocking. §6.3 discusses the evolution of anti-adblock filter lists. §6.4 presents our methodology to crawl historical snapshots of Alexa top-5K websites from the Wayback Machine and our retrospective coverage analysis of anti-adblock filter lists. §6.5 discusses our machine learning based approach to automatically detect anti-adblock scripts using static analysis. Finally, we conclude in §6.6 with an outlook to future research directions.

6.2 Background & Related Work

6.2.1 Background

Online publishers, advertisers, and data brokers track users across the Web to serve them personalized content and targeted advertisements. Online tracking is conducted using cookies, beacons, and fingerprinting [186]. Online tracking has raised serious privacy and surveillance concerns. Web tracking allows advertisers to infer sensitive information about users such as their medical and financial conditions [251]. Nation states can piggyback on web tracking information to conduct mass surveillance [187]. To combat privacy and surveillance concerns, one solution is to block trackers and the other is to block advertisements.

Tracker blockers remove network requests to known tracker domains. Tracker blocking extensions are available for all major web browsers. Ghostery [51] is one of the most popular tracker blocking extensions. It is used by more than 2.6 million Google Chrome users [52] and 1.3 million Mozilla Firefox users [53]. Another popular tracker blocking extension is the EFF's Privacy Badger [76], which is used by more than 532K Google Chrome users [77] and 116K Mozilla Firefox users [78]. Mainstream browsers such as Apple Safari and Mozilla Firefox have developed built-in tracking prevention solutions. Apple has recently launched Intelligent Tracking Prevention [310] in Safari to mitigate excessive tracking. Mozilla also has a tracking prevention solution [180] in the private browsing mode of Firefox.

Adblocking extensions can remove both advertisements and trackers. Like tracker blockers, adblocking extensions are also available for all major web browsers.

Two of the popular adblockers are Adblock Plus and AdBlock. Adblock Plus [5] is used by more than 10 million Google Chrome users [6] and 19 million Mozilla Firefox users [8]. AdBlock [3] is used by more than 10 million Google Chrome users [4]. Some new web browsers such as Cliqz [25] and Brave [16] now have built-in adblockers. Google Chrome has also recently announced that they will block ads [277] on websites that do not comply with the Better Ads Standards set by the Coalition for Better Ads [27].

Adblockers have become much more popular than tracker blockers because they provide benefits such as removal of intrusive ads, protection from malware, and improved page load time in addition to privacy protection against online tracking. A recent survey [74] showed that 43% users install adblockers to get rid of interruptive or too many ads, 30% to avoid spread of malware, 16% to boost the page load time, and 6% to protect their privacy.

Adblockers rely on crowdsourced filter lists to block advertisements and trackers. EasyList [36] is the most widely used filter list to block advertisements. There are also some language-specific filter lists to block advertisements such as EasyList Dutch [38], EasyList Germany [38], and EasyList Spanish [38]. EasyPrivacy [38] is the most widely used filter list to block trackers. Other tracker blocking filter lists include Fanboy's Enhanced Tracking List [42], Disconnect.me [31], Blockzilla [15], and NoTrack Blocklist [72]. Adblockers such as Adblock Plus and AdBlock, as well as web browsers such as Cliqz and Brave, are subscribed to EasyList. Adblockers typically allow users to subscribe to different filter lists and incorporate custom filter rules.

Filter list rules are regular expressions that match HTTP requests and HTML elements. Adblockers block HTTP requests and hide HTML elements if they match any of the filter rules. Below we provide a brief overview of the syntax of HTTP request filter rules and HTML element filter rules [43].

HTTP Request Filter Rules: HTTP request rules match against URLs in the HTTP request headers. As shown in Code 6.1, these rules may be domain specific with a domain anchor (||) or a domain tag (**domain=**). Rule 1 blocks HTTP requests to example1.com. Rule 2 blocks HTTP requests to example1.com to load JavaScript. Rule 3 blocks HTTP requests to example1.com to load JavaScript on example2.com. Rule 4 blocks HTTP requests to download example.js JavaScript on example2.com.

```
! Rule 1
example1.com
! Rule 2
example1.com$script
! Rule 3
example1.com$script,domain=example2.com
! Rule 4
/example.js$scipt,domain=example2.com
```

Script 6.1. HTTP request filter rules.

HTML Element Filter Rules: HTML element rules match against HTML elements loaded in a web page and hide the matched elements. Code 6.2 shows three examples of HTML element filter rules. Rule 1 hides the HTML element with ID examplebanner on example.com. Rule 2 hides the HTML element with class name examplebanner on example.com. Rule 3 hides the HTML element with ID examplebanner on any website.

```
!Rule 1
example.com###examplebanner
```

```
!Rule 2
example.com##.examplebanner
!Rule 3
###examplebanner
```

Script 6.2. HTML element filter rules.

Exception Rules: Exception rules override filter rules by allowing HTTP requests and HTML elements that match other filter rules. Exception rules are generally used to protect against false positives by filter rules that cause site breakage. Code 6.3 shows two examples of exception rules. Rule 1 allows HTTP requests to example.com to load JavaScript. Rule 2 allows HTML element with ID examplebanner on example.com.

```
! Rule 1
@@example.com$script
! Rule 2
example.com#@##elementbanner
```

Script 6.3. Exception rules for HTTP requests and HTML elements.

6.2.2 Related Work

Online Tracking. Prior research has demonstrated the widespread nature of online tracking. Krishnamurthy and Wills [231] showed that top 10 third party trackers had grown from 40% in October 2005 to 70% in September 2008 for 1200 popular websites. In another study, Ihm et al. [208] reported that the popularity of search engines (google.com or baidu.com) and analytics sites (google-analytics.com) had increased from 2006 to 2010. Lerner et al. [242] conducted a retrospective study of online tracking on top-500 websites using the Internet Archive's Wayback Machine [98]. They reported that the percentage of websites contacting at least 5 separate third parties has increased from 5% in early 2000s to 40% in 2016. They also reported that the coverage of top trackers on the web is increasing rapidly, with top 5 trackers now covering more than 60% of the top 500 websites as compared to less than 30% ten years ago. Englehardt and Narayanan [186] showed that a handful of third parties including Google, Facebook, Twitter, Amazon, and AdNexus track users across a significant fraction of the Alexa top one million websites. For example, they reported that Google alone tracks users across more than 80% of Alexa top one million websites.

Tracker Blocking. Tracker blocking tools have had mixed success in blocking online trackers. Roesner et al. [279] found that defenses such as third-party cookie blocking, Do Not Track (DNT), and popup blocking are not effective in blocking social media widgets tracking. Englehardt and Narayanan [186] demonstrated that existing tracker blocking tools like Ghostery, Disconnect, EasyList, and EasyPrivacy are less effective against obscure trackers. Merzdovnik et al. [255] found that popular tracker blocking tools have blind spots against stateless trackers and trackers with smaller footprints. To improve tracker blockers, Gugelmann et al. [203] proposed an automated approach that learns properties of advertisements and analytics services on existing filter lists and identifies new services to be included in adblockers' filter lists. Yu et al. [318] proposed an approach, inspired by k-anonymity, to automatically filter data sent to trackers that have the potential to uniquely identify a user. Their approach aims to block third-party tracker requests while avoiding blanket blocking of 3rd parties based on blacklists. Ikram et al. [209] proposed a one-class machine learning approach to detect trackers with high accuracy based on their JavaScript code structure. In the same spirit as prior research on improving tracker blocking tools, in this study we propose a machine learning approach to detect anti-adblockers. As we discuss later, our approach is customized to capture syntactic behaviors of anti-adblockers.

Adblocking and Anti-Adblocking. Prior research has focused on analyzing the prevalence of adblockers. Pujol et al. [274] analyzed network traces from a major European ISP and found that 22% of the most active users used Adblock Plus. Malloy et al. [247] conducted a measurement study of 2 million users and found that 18% of users in the U.S. employ adblockers. The increasing popularity of adblockers has pushed online publishers to retaliate against adblock users.

First, some online publishers have started to manipulate ad delivery to evade filter lists. For example, publishers can keep changing third-party domains that serve ads to bypass filter list rules. More recently, Facebook manipulated HTML element identifiers [153] to bypass their ads through filter lists. To address this problem, Storey et al. [299] proposed a perceptual adblocking method for visually identifying and blocking ads based on optical character recognition and fuzzy image matching techniques. The key idea behind perceptual adblocking is that ads are distinguishable from organic content due to government regulations (e.g., FTC [93]) and industry self-regulations (e.g., AdChoices [104]). The authors [299] reported that their perceptual adblocking approach is robust to ad delivery manipulation by online publishers.

Second, some online publishers have started to employ anti-adblock scripts to detect adblockers. Anti-adblock scripts detect adblock users and prompt them to disable their adblocker. Adblockers currently rely on manually curated anti-adblock filter lists, e.g., Anti-Adblock Killer List, to block anti-adblock scripts. Rafique et al. [276] manually analyzed top 1,000 free live streaming websites and reported that 16.3% websites attempted to detect adblockers. Nithyanand et al. [264] crawled Alexa top-5K websites and manually analyzed JavaScript snippets to find that 6.7% websites used anti-adblocking scripts. In our prior work, Mughees et al. [258] used a machine learning based approach with A/B testing to analyze changes in page content due to anti-adblockers. We found that only 686 out of Alexa-100K websites visibly react to adblockers. In contrast to prior efforts to study anti-adblock deployment, in this study we conduct a retrospective measurement study of anti-adblock filter lists that are currently used by adblockers to circumvent anti-adblockers. In our prior work [258] we used machine learning to detect anti-adblockers that visibly react to adblockers, while in this study we use machine learning to fingerprint anti-adblock scripts. Storey et al. [299] proposed stealth (hide adblocking) and active (actively counter adblock detection) adblocking approaches. For stealth adblocking, they partially implemented a rootkit-style approach that intercepts and modifies JavaScript API calls that are used by publishers to check the presence of ad elements. Their approach is complementary to our approach. For active adblocking, they implemented a signature-based approach to remove anti-adblock scripts using manually crafted regular expressions. In contrast, our proposed machine learning based approach can automatically identify anti-adblock scripts based on their syntactic features.

Next, we analyze the evolution of popular anti-adblock filter lists (§6.3), measure their historic coverage on popular websites (§6.4), and develop machine learning based approach to detect anti-adblock scripts (§6.5).

6.3 Analyzing Anti-Adblock Filter Lists

We first introduce anti-adblockers and then analyze popular anti-adblock filter lists.

6.3.1 How Anti-Adblocking Works?

Anti-adblockers employ baits to detect adblockers. These baits are designed and inserted in web pages such that adblockers will attempt to block them. To detect the presence of adblockers, anti-adblockers check whether these baits are blocked. Anti-adblockers use HTTP and HTML baits to detect adblockers. Below we discuss both of them separately.

For HTTP baits, anti-adblockers check whether the bait HTTP request is blocked by adblockers. Code 6.1 illustrates the use of HTTP bait by the anti-adblocker on `businessinsider.com`, which requests a bait URL `www.npttech.com/advertising.js` and checks whether it is successfully retrieved. Code 6.1 dynamically creates an HTTP request bait and calls `onLoad` event in case of success and `onError` event in case of failure. Both events call the `setAdblockerCookie` function with a parameter of either true or false. `setAdblockerCookie` event sets the value of the cookie `__adblocker` to either true or false depending on the input value.

```
var script = document.createElement("script");
script.setAttribute("async", true);
script.setAttribute("src", "//www.npttech.com/advertising.js");
script.setAttribute("onerror", "setAdblockerCookie(true);");
script.setAttribute("onload", "setAdblockerCookie(false);");
document.getElementsByTagName("head")[0].appendChild(script);

var setAdblockerCookie = function(adblocker) {
    var d = new Date();
    d.setTime(d.getTime() + 60 * 60 * 24 * 30 * 1000);
    document.cookie = "__adblocker=" + (adblocker ? "true" : "false") + ";
        expires=" + d.toUTCString() + "; path=/";
}
```

Code 6.1. Anti-adblocker JavaScript code for HTTP bait.

For HTML baits, anti-adblockers check if the CSS properties of the bait HTML element are modified. Code 6.2 illustrates the use of HTML bait by a popular third-party anti-adblocker called BlockAdBlock [14], which creates a div bait and checks whether the bait is removed. The `_creatBait` function creates a div element and sets its CSS properties. The `_checkBait` function checks whether the div element's CSS properties such as `offsetHeight`, `offsetTop`, and `offsetWidth` are changed.

```
BlockAdBlock.prototype._creatBait = function() {
  var bait = document.createElement('div');
  bait.setAttribute('class', this._options.baitClass);
  bait.setAttribute('style', this._options.baitStyle);
  this._var.bait = window.document.body.appendChild(bait);
  this._var.bait.offsetParent;
  this._var.bait.offsetHeight;
  this._var.bait.offsetLeft;
  this._var.bait.offsetTop;
  this._var.bait.offsetWidth;
  this._var.bait.clientHeight;
  this._var.bait.clientWidth;
  if (this._options.debug === true) {
   this._log('_creatBait', 'Bait has been created');
  }
};

BlockAdBlock.prototype._checkBait = function(loop) {
    var detected = false;
    if(window.document.body.getAttribute('abp') !== null
     this._var.bait.offsetParent === null
     this._var.bait.offsetHeight == 0
     this._var.bait.offsetLeft == 0
     this._var.bait.offsetTop == 0
     this._var.bait.offsetWidth == 0
     this._var.bait.clientHeight == 0
     this._var.bait.clientWidth == 0) {
      detected = true;
    }
};
```

Code 6.2. BlockAdBlock JavaScript code for creating and checking a bait.

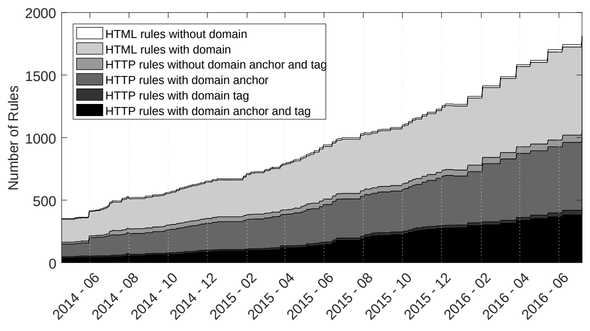
6.3.2 Anti-Adblock Filter Lists

Using the aforementioned techniques, anti-adblockers detect adblockers and prompt users to disable their adblockers if they want to view page content. To circumvent anti-adblockers, adblockers currently rely on anti-adblock filter lists. The rules of these filter lists are designed to handle HTTP requests and HTML elements that are used by anti-adblockers. For example, the filter list rules may allow or block HTTP requests and HTML elements to avoid detection by anti-adblockers. Code 6.3 shows two examples of anti-adblock rules. Rule 1 blocks third-party HTTP requests to pagefair.com which is a well-known anti-adblock vendor. Rule 2 hides the HTML element with ID `noticeMain` which displays an anti-adblock notice on smashboards.com.

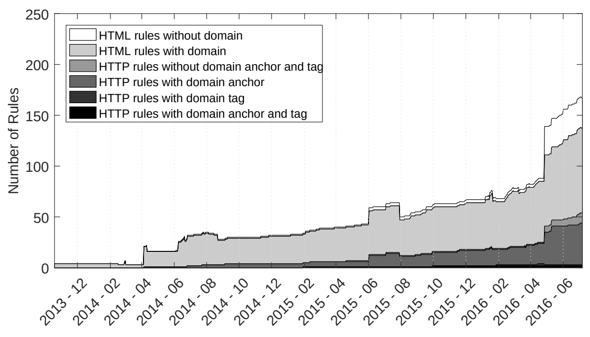
```
! Rule 1
pagefair.com^$third-party
! Rule 2
smashboards.com###noticeMain
```

Code 6.3. Example anti-adblock filter rules.

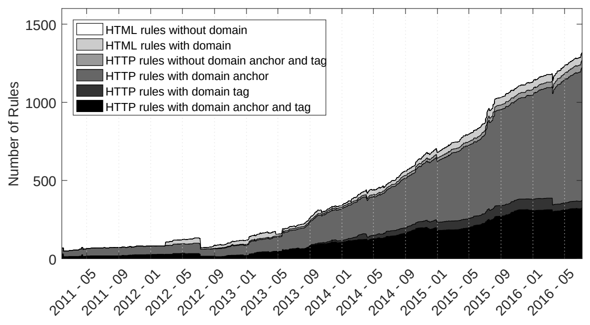
The most widely used anti-adblock filter lists are: (1) Anti-Adblock Killer List [11], (2) Adblock Warning Removal List [97], and (3) EasyList [36]. The first two are dedicated to target anti-adblockers, however, EasyList's main purpose is to block ads. Several sections in EasyList specifically contain anti-adblocking filter rules. In this study, we only analyze anti-adblock sections of EasyList. Anti-Adblock Killer List started in 2014, Adblock Warning Removal List started in 2013, and anti-adblock sections in EasyList were created in 2011. These anti-adblock filter lists rely on informal crowdsourced input from their users (e.g., via feedback forums) to add new filter rules or remove/modify old filter rules. These anti-adblock filter lists have been regularly updated since their creation.



(a) Anti-Adblock Killer



(b) Adblock Warning Removal List



(c) EasyList

Figure 6.1. Temporal evolution of anti-adblock filter lists.

Anti-Adblock Killer. Anti-Adblock Killer List was created by "reek" in 2014. The list is maintained on GitHub [11] and users submit feedback by reporting issues on the GitHub page [12]. On average, the list adds or modifies 6.2 filter rules for every revision. Figure 6.1a visualizes the temporal evolution of different types of filter rules in the Anti-Adblock Killer List. We observe a steady increase in number of filter rules. The filter list started with 353 initial filter rules and it has expanded to

1,811 filter rules by July 2016. The stair step pattern in the number of filter rules starting November 2015 indicates that the update cycle of the filter list increased to approximately once a month. For this span, the list adds or modifies 60 filter rules for every revision on average. The most common types of filter rules are HTTP request rules (with domain anchor and both domain anchor and tag) and HTML element rules (with domain). The most recent version of the filter list has 58.5% HTTP request rules and 41.5% HTML element rules. 31.0% filter rules are HTTP request rules with only domain anchor, 2.1% are HTTP request rules with only domain tag, 22.0% are HTTP request rules with both domain anchor and tag, and 3.4% are HTTP request rules without domain anchor and tag. 40.0% filter rules are HTML element rules with domain and 1.5% are HTML element rules without domain.

Adblock Warning Removal List. Adblock Warning Removal List was created by the EasyList filter list project [36] in 2013. The list is maintained by multiple authors and relies on user feedback on their forum [37]. On average, the list adds or modifies 0.2 filter rules every day. Figure 6.1b visualizes the temporal evolution of different types of filter rules in the Adblock Warning Removal List. In contrast to the Anti-Adblock Killer List, it is noteworthy that this list contains a larger fraction of HTML filter rules. The HTML element filter rules hide anti-adblock warning popups that are displayed when anti-adblockers detect adblockers. The filter list started in 2013 with 4 filter rules and it has expanded to 167 filter rules by July 2016. While the filter list initially grows slowly, we note a significant spike in the number of filter rules in April 2016 after which the rate of addition of new rules increases as well. The spike observed in April 2016 corresponds to the addition of French language section in the filter list. The most recent version of the filter list has 32.3% HTTP request rules and 67.7% HTML element rules. 24.5% filter rules are HTTP request rules with only domain anchor, 0.6% are HTTP request rules with only domain tag, 1.2% are HTTP request rules with both domain anchor and tag, and 6.0% are HTTP request rules without domain anchor and tag. 49.7% filter rules are HTML element rules with domain and 18.0% are HTML element rules without domain.

EasyList. EasyList's primary purpose is to block ads. All major adblockers are subscribed to EasyList by default. As discussed earlier, several sections in EasyList specifically contain anti-adblocking filter rules. EasyList started adding anti-adblock rules in 2011. Our analysis here focuses only on the anti-adblock sections of EasyList. On average, the list adds or modifies 0.6 anti-adblock filter rules every day. Figure 6.1c visualizes the temporal evolution of different types of anti-adblock filter rules in EasyList. The filter list started with 67 filter rules in 2011 and it has expanded to 1,317 filter rules by July 2016. We observe a steady increase in the number of HTTP request filter rules. It is noteworthy that the filter list contains a relatively small fraction of HTML element filter rules. The most recent version of the filter list has 96.3% HTTP request rules and 3.7% HTML element rules. 64.6% filter rules are HTTP request rules with only domain anchor, 3.6% are HTTP request rules with only domain tag, 24.6% are HTTP request rules with both domain anchor and tag, and 3.5% are HTTP request rules without domain anchor and tag. 3.7% filter rules are HTML element rules with domain. The filter list does not contain any HTML element rule without domain.

6.3.3 Comparative Analysis of Anti-Adblock Lists

Next, we compare and contrast different anti-adblocking filter lists. To this end, we decide to combine EasyList and Adblock Warning Removal List because (1) they are both managed by the EasyList filter list project [36] and (2) they are complementary — Adblock Warning Removal List mostly contains HTML element filter rules and EasyList mostly contains HTTP request filter rules. The most recent version of the Combined EasyList contains 1,483 rules.

Since a vast majority of filter list rules contain domain information, we first compare the number of domains in both filter lists. Anti-Adblock Killer List and Combined EasyList include 1,415 and 1,394 domains, respectively. To our surprise, these filter lists have only 282 domains in common. To analyze why the filter lists are targeting different sets of domains, we break down the set of domains based on the Alexa popularity ranks and their category. Table 6.1 provides the breakdown of domains in both filter lists based on their Alexa ranks. While there are some differences, popularity distributions of domains in both filter lists are fairly similar. For domain categorization, we use McAfee's URL categorization service [66] and manually merge similar categories together. Figure 6.2 shows the distribution of domains in both filter lists based on McAfee's URL categorization. We plot top 15 categories and group remaining categories as others. Similar to the distribution of Alexa popularity ranks, the categorization trend is also similar across both filter lists. Below, we further investigate the differences in the filter lists by analyzing whether

Alexa Rank	**Anti-Adblock Killer List**	**Combined EasyList**
1-5K	112	124
5K-10K	49	69
10K-100K	280	312
100K-1M	334	359
>1M	640	530

Table 6.1. Distribution of domains in filter lists across Alexa rankings.

they are implemented differently.

To analyze whether these filter lists are implemented differently, we study exception and non-exception conditions in filter rules. We label the domains as exception or non-exception on the basis of rules in which they appeared. If the rule is an exception rule, we label the domain as exception domain. If the rule is a non-

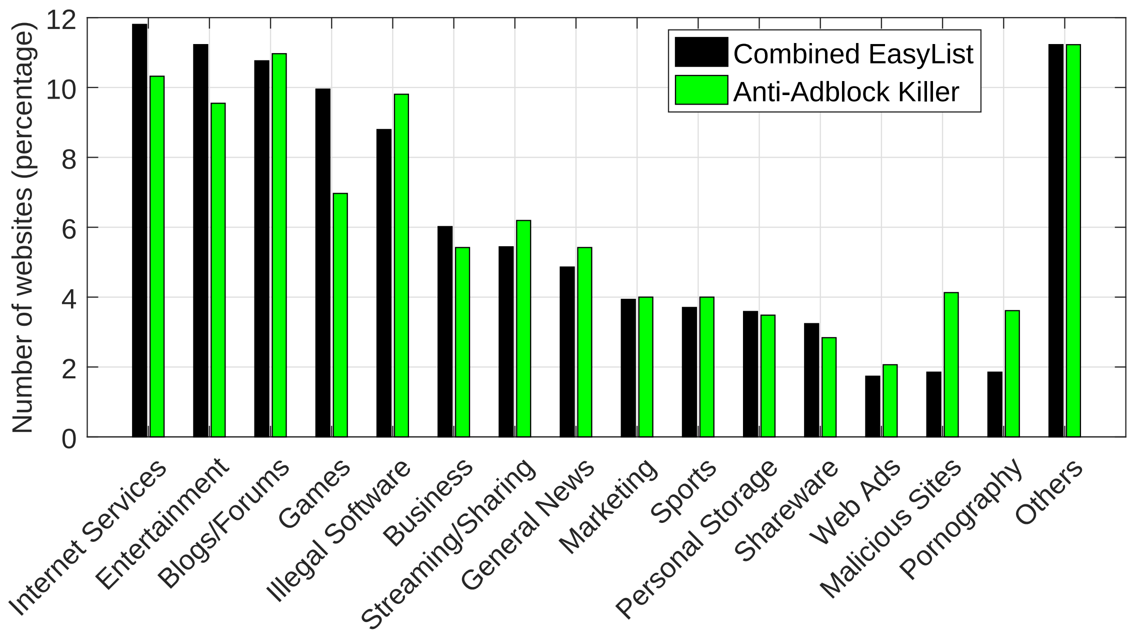


Figure 6.2. Categorization of domains in anti-adblock filter lists.

exception rule, we label the domain as non-exception domain. For the Combined EasyList, we note a ratio of approximately 4:1 for exception to non-exception domains. For the Anti-Adblock Killer List, we note a ratio of approximately 1:1 for exception to non-exception domains.

We next investigate the difference in proportion of exception and non-exception domains in both filter lists. We surmise that the Combined EasyList has many exception domains because it works with a large number of adblock rules in the full EasyList. For example, rule 1 in Code 6.4 is an adblocking rule that blocks every URL that ends with `/ads.js?` and rule 2 is an exception rule that overrides rule 1 and allows `/ads.js?` on `numerama.com`. Our analysis revealed that `numerama.com` was using `/numerama.com/ads.js` as a bait HTTP request. More specifically, in Code 6.5, we note that `canRunAds` will be undefined if bait HTTP request is blocked and `adblockStatus` will be set to `active`. Combined EasyList has many such exception rules. In contrast, we note that the Anti-Adblock Killer List has a larger fraction of non-exception filter rules that block anti-adblock scripts on specific domains. Our results highlight that these anti-adblock filter lists have different approaches to writing filter rules.

```
! Rule 1
/ads.js?
! Rule 2
@@numerama.com/ads.js
```

Code 6.4. URL that is blocked on other websites but allowed on `numerama.com`.

```
canRunAds = true;
var adblockStatus = 'inactive';
if (window.canRunAds === undefined) {
  adblockStatus = 'active';
}
```

Code 6.5. Use of HTTP bait by `numerama.com`.

We next analyze the implementation of filter rules for 282 overlapping domains in the Combined EasyList and Anti-Adblock Killer List. Domain specific rules are implemented through domain tag (**domain**=), domain anchor (||), and HTML element tag (##). Code 6.6 shows the filter rule implementation by both filter lists for `yocast.tv` using HTML element tag. The Combined EasyList hides HTML element with ID `ra9e` on `yocast.tv`. Whereas the Anti-Adblock Killer List hides HTML element with ID `notice` on `yocast.tv`. Code 6.7 shows the filter rule implementation by both filter lists for `pagefair.com` using domain anchor. The Combined EasyList blocks all HTTP requests from `pagefair.com` by its general adblocking rules and focuses on specific websites in its anti-adblocking rules. Whereas the Anti-Adblock Killer List blocks all HTTP requests from `pagefair.com` on any domain. Overall, we note that both filter lists often have different rules to circumvent anti-adblockers even for the same set of domains.

```
! Combined EasyList
yocast.tv###ra9e
! Anti-Adblock Killer list
yocast.tv###notice
```

Code 6.6. Implementation of yocast.tv for the Combined EasyList and Anti-Adblock Killer List.

```
! Combined EasyList
pagefair.com/static/adblock_detection/js/d.min.js$domain=majorleaguegaming.com
! Anti-Adblock Killer list
pagefair.com^$third-party
```

Code 6.7. Implementation of pagefair.com for the Combined EasyList and Anti-Adblock Killer List.

Finally, we investigate which of these anti-adblock filter lists is more prompt in adding new filter rules. To this end, we compare and contrast the addition time of 282 overlapping domains in both filter lists. We find that 185 domains appear first in the

Combined EasyList, 92 domains appear first in the Anti-Adblock Killer List, and 5 domains appear on the same day on both filter lists. Figure 6.3 plots the distribution of overlapping domains in terms of the difference in days when they appear in each list. These results seem to indicate that the Combined EasyList is more prompt in adding new rules. The difference can be explained in part because the Combined EasyList is used by default in many popular anti-adblockers, thus we expect it to get more user feedback from its much larger user base than that for the Anti-Adblock Killer List. We also note as a caveat that we should expect many domains to appear in the Combined EasyList before the Anti-Adblock Killer List because the Combined EasyList started more than two years before the Anti-Adblock Killer List.

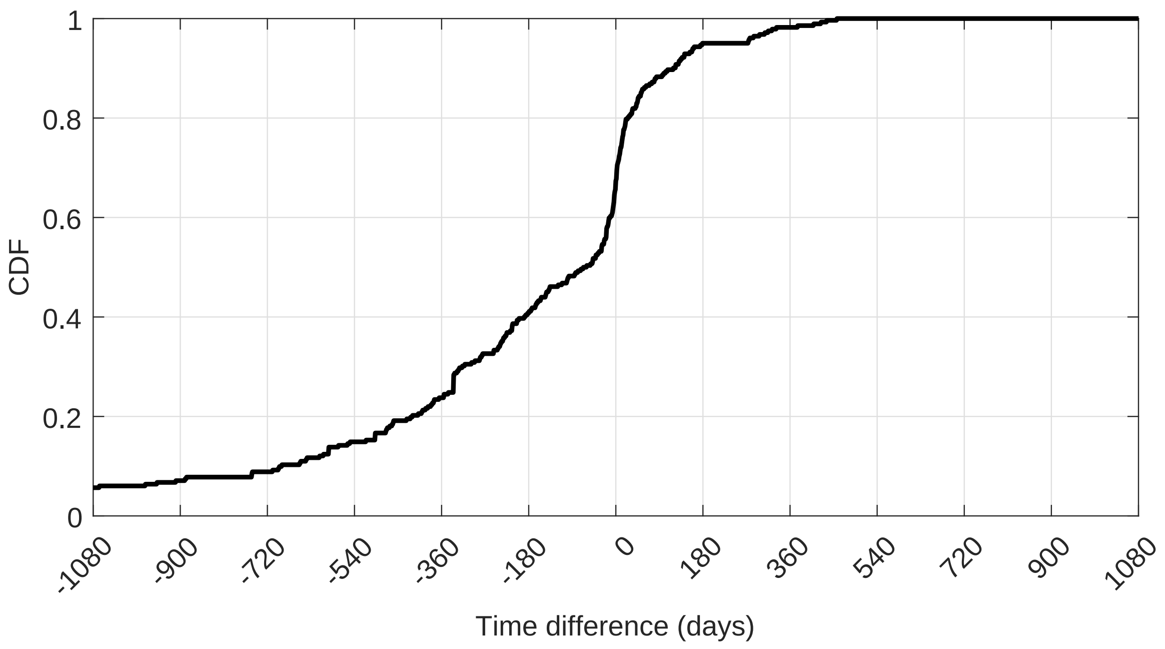


Figure 6.3. Distribution of time difference (number of days) between the Combined EasyList and Anti-Adblock Killer List for addition of rules that target overlapping domains.

We have identified several key differences in the filter rule implementation across two popular anti-adblock filter lists. However, our comparative analysis of filter lists provides an incomplete picture of their behavior. First, the behavior of individual filter rules is dependent on other rules in the filter list. Second, the effectiveness of these filter lists is dependent on complex and often dynamic website behavior. Therefore, to fully understand and compare the behavior of the Combined EasyList and Anti-Adblock Killer List, we need to run them on actual websites and see how different filter rules are triggered.

6.4 Analyzing Filter List Coverage using the Wayback Machine

We now conduct a retrospective study to analyze how anti-adblock filter lists trigger on popular websites. Our goal is to study the evolution of anti-adblock prevalence. To collect historical snapshots of popular websites, we rely on the Internet Archive's Wayback Machine [98] which has archived historic snapshots of 279 billion web pages since 1996 [100].

We crawl Alexa top-5K websites for the last five years from the Wayback Machine and check for the presence of anti-adblockers using the Combined EasyList and Anti-Adblock Killer List. Figure 6.4 provides an overview of our measurement methodology. We first identify domains that are not archived by the Wayback Machine. We then request the Wayback Availability JSON API to collect the URLs for the monthly archives of websites. We remove outdated URLs for which the Wayback Machine does not have a snapshot close to our requested date. We automatically crawl the remaining URLs and store their request/responses in the HTTP Archive

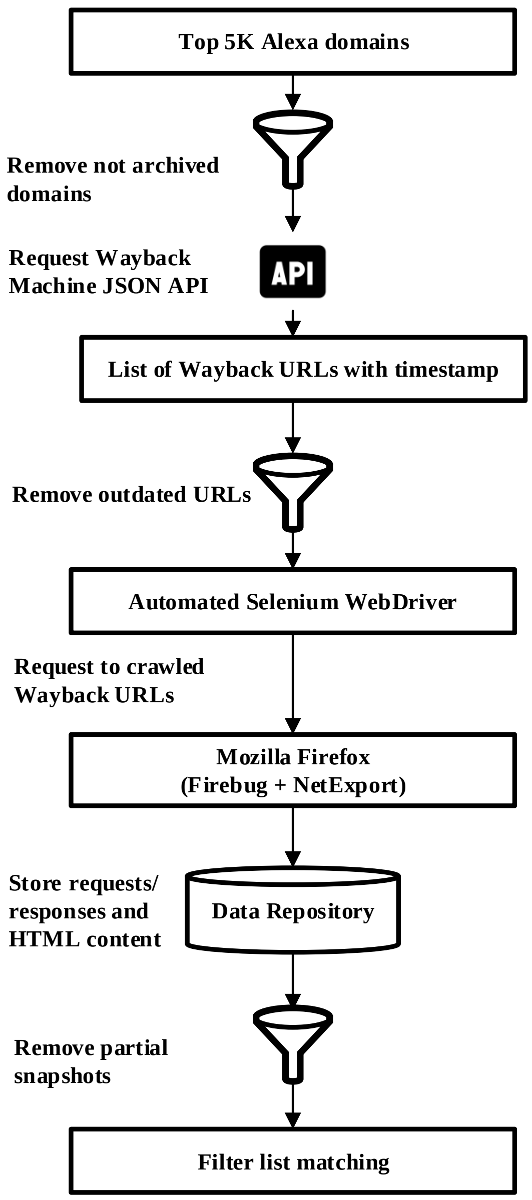

Top 5K Alexa domains
Remove not archived domains
Request Wayback Machine JSON API
API
List of Wayback URLs with timestamp
Remove outdated URLs
Automated Selenium WebDriver
Request to crawled Wayback URLs
Mozilla Firefox (Firebug + NetExport)
Store requests/ responses and HTML content
Data Repository
Remove partial snapshots
Filter list matching

Figure 6.4. Overview of our measurement methodology to crawl historical snapshots from the Wayback Machine and match against anti-adblock filter lists.

(or HAR) format. We then remove partial HAR files and use the remaining to match against anti-adblock filter lists. Below we explain our methodology in more detail.

6.4.1 Crawling the Wayback Machine

The Wayback Machine archives snapshots of popular websites several times everyday. While popular websites frequently change their content, they much less frequently change their codebase and dependance on third-party scripts such as anti-adblocking scripts. This allows us to reduce the data set that we need to crawl. We decide to only crawl one snapshot per month for a website. We attempt to crawl a total of approximately 300K URLs from the Wayback Machine. A serial crawler implementation would take several months to completely crawl all URLs. To speed up crawling, we parallelize crawlers using 10 independent browser instances.

For each of the Alexa top-5K domains, we first check whether the domain is archived by the Wayback Machine. The Wayback Machine may decide to not archive a domain due to that domain's robots.txt exclusion policy, at the request of the domain's administrator, or for undefined reasons.[1] The Wayback Machine does not archive 153 domains because of their robots.txt exclusion policy, 26 domains because of domain's administrator request, and 54 domains because of undefined reasons.

For the remaining domains, we request the Wayback Availability JSON API [99] to collect the URLs for the monthly archives of the domains' homepages. The Wayback Availability JSON API returns a URL that is closest to the requested date. If a website is not archived by the Wayback Machine, an empty JSON response is returned by the Wayback Availability JSON API. If a URL is returned we check its

[1]The Internet Archive project recently announced [202] a policy change to ignore robots.txt directives.

timestamp and discard URLs for which the time difference between the requested date and the actual date of URL is more than 6 months.

We open the remaining URLs in a fully functional web browser using Selenium WebDriver [84] and collect all HTTP requests and responses. We configure a profile of Mozilla Firefox [69] browser with Firebug [46] and NetExport [71] plugins. For each website we visit, we store all HTTP requests/responses in a HAR file [56] and the page content in a HTML file. We get multiple HAR files for some websites that keep on refreshing. For such websites, we take a union of all HTTP requests in HAR files. We get incomplete HAR files for websites that are not completely archived by the Wayback Machine. We discard the partial HAR files, whose size is less than 10% of the average size of HAR files over a year.

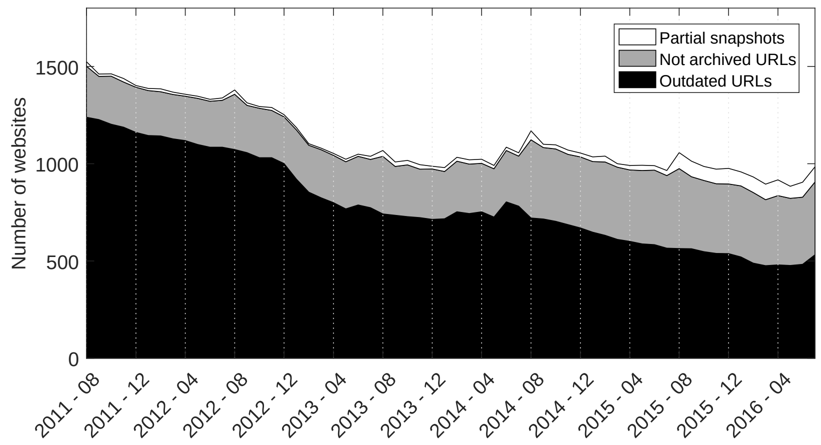


Figure 6.5. Number of websites over time excluded from analysis.

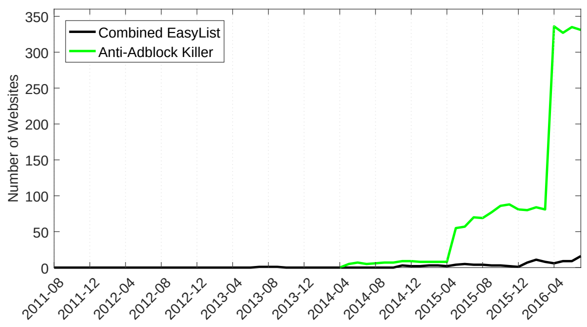
Figure 6.5 shows the timeseries of number of missing monthly snapshots by the Wayback Machine. We note that the number of missing snapshots has decreased from

1,524 in August 2011 to 984 in July 2016. Outdated URLs account for the highest proportion of missing snapshots. The number of outdated URLs has decreased from 1,239 in August 2011 to 532 in July 2016. However, the number of not archived URLs has increased from 262 in August 2011 to 374 in July 2016. Our analysis of not archived URLs shows that it is generally due to HTTP 3XX redirects. For 3XX redirects, the Wayback Availability JSON API returns an empty JSON object. The number of partial snapshots has also increased from 23 in August 2011 to 78 in July 2016. Our analysis of partial snapshots shows that it is due to anti-abuse policies implemented by websites against bots. Some domains show an error page if they detect the Wayback Machine's crawling bots. For correctly archived websites, we verified that the Wayback Machine serves the same snapshots to a normal user (using a regular browser) and our crawlers. To this end, we manually analyzed a sample of randomly selected 20 URLs and did not find any difference in content served to a normal user and our crawler. Our analysis corroborates the findings of Lerner et al. [242] that the Wayback Machine archives most websites reliably. It is noteworthy that we miss location specific content because we rely on the content archived by the Wayback Machine. The archived content is specific to the location of the Wayback Machine's servers.

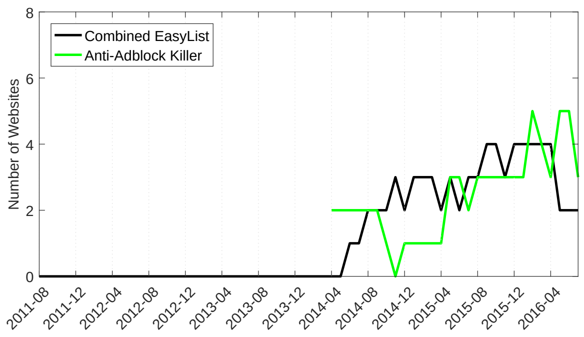
6.4.2 Anti-Adblock Detection on the Wayback Machine

After crawling the historic data of Alexa top-5K domains, we check for the presence of anti-adblockers using the Anti-Adblock Killer List and Combined EasyList. We use the historic versions of these filter lists at that point in time because it will portray the retrospective effectiveness of these filter lists. For detection with HTTP request filter rules, we extract the HTTP requests from the HAR files and see whether they trigger HTTP rules in the filter lists. For detection with HTML element filter rules, we open the stored HTML page in a web browser with adblocker enabled and see whether it triggers HTML rules in the filter lists.

To detect the presence of anti-adblockers with HTTP request filter rules, we extract HTTP request URLs from the HAR files crawled from the Wayback Machine. To archive a website, the Wayback Machine replaces all live URLs with its own reference by prepending `http://web.archive.org` to the URL. To run HTTP request filter rules against these URLs, we truncate the Wayback Machine references. Note that we do not truncate Wayback escape [241,242] URLs because they are not archived with the Wayback Machine reference. We use `adblockparser` [9] to match all URLs of a website against HTTP request filter rules. We label a website as anti-adblocking if any of its URLs is matched against any of the HTTP request filter rules. Figure 6.6a shows the number of anti-adblocking websites in Alexa top-5K websites that are detected by HTTP request filter rules in Anti-Adblock Killer List and Combined EasyList. For the Anti-Adblock Killer List, the number of matched websites has increased from 0 in April 2014 to 331 in July 2016. For the Combined EasyList, the number of matched websites has increased from 0 in August 2011 to 16 in July 2016. In contrast to our comparative analysis of these filter lists in §6.3, it is noteworthy that the number of matched websites for the Anti-Adblock Killer List is much higher than that for the Combined EasyList. Our analysis of matched URLs also reveals that a vast majority of websites use third party scripts for anti-adblocking. More specifically, more than 98% of matched websites for the Anti-Adblock Killer List use scripts from third party anti-adblocking vendors such as Optimizel, Histats, PageFair, and Blockadblock.



(a) Number of websites that trigger HTTP rules



(b) Number of websites that trigger HTML rules

Figure 6.6. Temporal evolution of websites that trigger HTTP and HTML filter rules of the Anti-Adblock Killer List and Combined EasyList.

To detect the presence of anti-adblockers with HTML element filter rules, we open the HTML webpage crawled from the Wayback Machine in a fully functional web browser and analyze whether it triggers any HTML element filter rules in the filter lists. We configure a profile of Mozilla Firefox browser with Adblock Plus enabled and subscribe to the Anti-Adblock Killer List and Combined EasyList. For each crawled HTML webpage, we open the webpage in the browser and wait 180 seconds for it to load completely and for HTML element filter rules to trigger. After that, we analyze the Adblock Plus logs and extract the triggered HTML element filter rules for the crawled HTML webpages. Figure 6.6b shows the number of anti-adblocking websites in Alexa top-5K websites that are detected by HTML element filter rules in the Anti-Adblock Killer List and the Combined EasyList. We note that the number of matching HTML element rules is much less than that with HTTP request filter rules. For the Anti-Adblock Killer List, the number of matched websites remains between 0 and 5 from April 2014 to July 2016. For the Combined EasyList, the number of matched websites remains between 0 and 4 from August 2011 to July 2016.

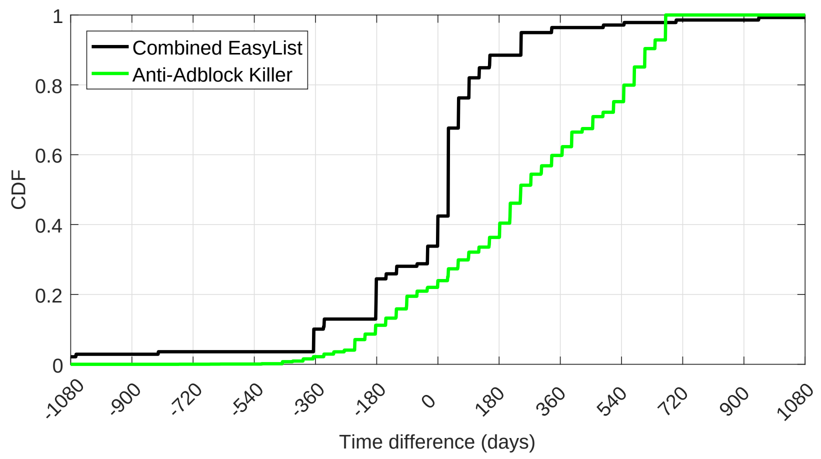


Figure 6.7. Distribution of time difference (number of days) between the day an anti-adblocker was added to a website and the day it was detected by Combined EasyList and Anti-Adblock Killer List.

Next, we compare the speed of the Anti-Adblock Killer List and the Combined EasyList in adding a new filter rule for an anti-adblocker after its addition. Figure 6.7 plots the distribution of time difference in terms of number of days when an anti-adblocker was added to a website and when a filter rule was added to detect it. The results indicate that the Combined EasyList is more prompt than the Anti-Adblock

Killer List in adding new filter rules for anti-adblockers that are added to a website. In the Combined EasyList, filter rules are defined for 82% anti-adblockers within 100 days of their addition to a website. However, in the Anti-Adblock Killer List, filter rules are defined for only 32% anti-adblockers within 100 days of their addition to a website. It is noteworthy that for a fraction of anti-adblockers filter rules are present on the Combined EasyList and Anti-Adblock Killer List even before they are added to a website. This can happen when the filter list uses generic rules to block third-party anti-adblockers. In the Combined EasyList, filter rules are present for 42% anti-adblockers before their addition to a website. In the Anti-Adblock Killer List, filter rules are present for 23% anti-adblockers before their addition to a website.

6.4.3 Anti-Adblock Detection on the Live Web

After retrospectively analyzing the coverage of anti-adblock filter lists using the Wayback Machine, we next study their coverage on the live Web in April 2017. We crawl Alexa top-100K websites and check for the presence of anti-adblockers using the Anti-Adblock Killer List and Combined EasyList. For detection on live Web, we use the most recent version of filter lists. Overall, our results on the live Web corroborate the findings of our retrospective analysis using the Wayback Machine. For example, we observe that the coverage of the Anti-Adblock Killer List is much more than that of the Combined EasyList. For the Anti-Adblock Killer List, the number of websites that trigger HTTP request filter rules is 4,931 out of 99,396. For the Combined EasyList, the number of websites that trigger HTTP request filter rules is 182 out of 99,396. Furthermore, we find that the number of websites that trigger

HTML element filter rules is much smaller. Specifically, the number of websites that trigger HTML element filter rules is 11 for the Anti-Adblock Killer List and 15 for the Combined EasyList. We again note that a vast majority of websites use third party anti-adblock scripts. For the Anti-Adblock Killer List, 97% of the matched websites use anti-adblocking scripts from third party vendors.

6.5 Detecting Anti-Adblock Scripts

Since anti-adblock filter lists are currently manually maintained, it is challenging for their authors to keep them up-to-date. The two popular anti-adblock filter lists are implemented differently and have varying coverage and update frequency. For example, the anti-adblock filter list with better coverage tends to be updated less frequently. We note that it is challenging for anti-adblock filter lists to keep pace with anti-adblockers that quickly evolve in response to adblockers [258]. Therefore, to help anti-adblock filter list authors, we next investigate a machine learning based automated approach to detect anti-adblock scripts.

Online publishers use JavaScript to implement client side anti-adblocking logic. We plan to fingerprint anti-adblocking JavaScript behavior by doing static code analysis. The basic idea is to extract syntactic features from anti-adblocking JavaScript code and build a light weight machine learning classifier. Our approach is inspired by prior machine learning based JavaScript analysis approaches to detect malware [172] and trackers [209].

Figure 6.8 shows the workflow of our proposed approach. We first unpack JavaScript files using the Chrome V8 engine. We construct Abstract Syntax Trees

(ASTs) of the parsed scripts and then extract different syntactic features. We use supervised machine learning to train a classifier (AdaBoost + SVM) for distinguishing between anti-adblocking and non anti-adblocking scripts. Below we discuss different steps of our static JavaScript code analysis approach to detect anti-adblockers.

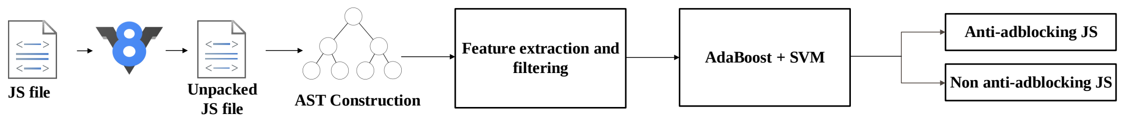


Figure 6.8. Overview of anti-adblock script detection approach.

Unpacking Dynamic JavaScript. The dynamic nature of JavaScript makes it challenging to do static analysis. For example, JavaScript code is often packed using `eval()` function. Such code unpacks itself right before it is executed. To cate for dynamically generated code, we use Chrome V8 engine to unpack `eval()` function by intercepting calls to the `script.parsed` function. `script.parsed` function is invoked every time `eval()` is evaluated or new code is added with `<iframe>` or `<script>` tags.

Gathering Labeled Data. In order to train a machine learning classifier, we need labeled examples of anti-adblocking and non anti-adblocking scripts. We utilize more than one million JavaScript snippets that were collected as part of our retrospective measurement study of Alexa top-5K websites using the Wayback Machine. Our anti-adblock data set consists of JavaScript snippets that matched HTTP request filter rules of crowdsourced anti-adblock filter lists. We use 372 of these anti-adblocking scripts as positive examples. To collect negative examples, we use the remaining scripts that the filter lists did not identify as anti-adblockers. We aim for a class imbalance of approximately 10:1 (negative:positive) in our labeled data. We manually verify a randomly selected 10% sample of ground truth for positive examples. We find that a vast majority of the scripts are served from known anti-adblocking vendors such as Optimizely and Blockadblock.

Feature Extraction. To extract features, we map the parsed scripts to ASTs, which are syntactic representations of JavaScript code in a tree format. After constructing the ASTs, we extract features according to the hierarchical tree structure. We define a feature as a combination of *context* and *text*. *Context* is the place where the feature appears, such as loop, try statement, catch statement, if condition, switch condition, etc. *Text* is code that appears in the context. We extract three types of feature sets based on different selection criterion of *text*. For the first type (**all**), we consider all text elements including JavaScript keywords, JavaScript Web API keywords, iden-

Features	**Types**
MemberExpression:BlockAdBlock	**all**
MemberExpression:_creatBait	**all**
MemberExpression:_checkBait	**all**
Literal:abp	**all**, **literal**
Literal:0	**all**, **literal**
Literal:hidden	**all**, **literal**
Identifier:clientHeight	**all**, **keyword**
Identifier:clientWidth	**all**, **keyword**
Identifier:offsetHeight	**all**, **keyword**
Identifier:offsetWidth	**all**, **keyword**

Table 6.2. Some features extracted from BlockAdBlock JavaScript.

tifiers, and literals. For the second type (**literal**), we consider text elements only from literals, i.e., we remove JavaScript keywords, JavaScript Web API keywords, and identifiers. These features are very general because they do not contain identifiers and keyword specific text in JavaScript code. For the third type (**keyword**), we consider text elements only from native JavaScript keywords and JavaScript Web API keywords, i.e., we remove identifiers and literals. As we do not consider text elements from identifiers and literals, **keyword** features are not impacted by the randomization of identifiers and literals. However, these features are susceptible to polymorphism. These three feature sets provide us varying levels of generalization. We expect more general features to be robust to minor implementation changes, however they may lose some useful information due to generalization.

Table 6.2 shows some extracted features and their types for Code 6.2. **Literal** features capture textual properties of JavaScript code such as `abp` and `0` in Table 6.2. **Keyword** features capture syntactic properties of JavaScript code such as `clien-tHeight` and `clientWidth` in Table 6.2. In contrast, **all** features can contain text such as `_checkBait` and `_creatBait` that represents names of variables, functions, etc.

Feature Selection. For these three types of feature sets, we extract a total of 1,714,827, 1,211,029, and 16,620 distinct features. Note that each feature is binary, i.e., its value is 1 or 0 depending on whether the feature is present/absent in a script. We construct a vector space to map scripts. Scripts with similar features are placed close to each other as compared to scripts with dissimilar features. We construct such a vector space by defining the mapping function as:

$$\phi : x \rightarrow (\phi_s(x))_{s \in S}$$

$$\phi_s(x) = \begin{cases} 1 & \text{if } x \text{ contains the feature } s \\ 0 & \text{otherwise} \end{cases}$$

We map each script x to a vector space with the mapping function ϕ. The mapping function ϕ assign 1 for a feature s that is present in script x and 0 otherwise, and S is the set of all possible features. Each instance of anti-adblocking and non anti-adblocking class is mapped with a mapping function $\phi(x)$.

After constructing our feature space, we remove irrelevant features. First, we remove features that do not vary much. To this end, we compute variance of each feature and remove it if its variance is less than 0.01. After applying this filter we are left with 68,510, 32,226, and 6,171 features for three types of feature sets. Second, we also remove duplicate features. After applying this filters we are left with 33,832, 12,974, and 5,785 features for three types of feature sets. Third, since we still have a large number of features, we want to select important features that strongly correlate with either positive or negative class. We further filter the remaining features using chi-square correlation [317].

$$\chi^2 = \frac{N \times (AD - CB)^2}{(A + C) \times (B + D) \times (A + B) \times (C + D)}$$

Where N is the total number of scripts. A is the number of positive samples where the binary feature is present. B is the number of negative scripts where the binary feature is present. C is the number of positive samples where the binary feature is absent. D is the number of negative samples where the binary feature is absent. Based on chi-square values, we rank features in the order of their importance. We select top 10K, 5K, 1K, and 100 features from three types of feature sets for further analysis.

Classifier Training. We decide to use AdaBoost, a boosting algorithm, due to the imbalance of anti-adblockers in the wild. AdaBoost [194] is an ensemble classifier that aims to create a strong meta-classifier from multiple weak classifiers. Models are built from the training data, each subsequent model attempts to correct the errors from the previous model. AdaBoost is expressed as:

$$f(x) = sign(\Sigma_{t=1}^{T} \alpha_t h_t(x))$$

Here x is the input vector, $h_t(x)$ is the component classifier, and α_t is the weight of each classifier. The distribution of weights is uniform at the start. At each cycle t, weight distribution is updated according to the result of component classifier $h_t(x)$. Training samples that are misclassified get higher weights. This process continues for T cycles and the results are combined at the end. Component classifiers with lower training errors get higher weights. We use SVM as the component classifier for AdaBoost. AdaBoost with SVM using RBF (Radial Basis Function) as its kernel tends to perform better for imbalanced classification problems [244].

Classifier	**# Features**	**TP rate (%)**	**FP rate (%)**
Feature set: all			
AdaBoost + SVM	10K	99.6	3.9
AdaBoost + SVM	1K	99.2	8.9
AdaBoost + SVM	100	99.2	8.9
SVM	10K	99.2	8.6
SVM	1K	99.2	8.9
SVM	100	99.2	8.4
Feature set: literal			
AdaBoost + SVM	10K	99.6	3.9
AdaBoost + SVM	1K	99.2	9.1
AdaBoost + SVM	100	99.2	8.9
SVM	10K	99.2	8.4
SVM	1K	99.2	8.9
SVM	100	99.2	8.6
Feature set: keyword			
AdaBoost + SVM	5K	99.6	3.7
AdaBoost + SVM	1K	99.7	3.2
AdaBoost + SVM	100	99.2	8.9
SVM	5K	99.2	8.4
SVM	1K	99.2	8.4
SVM	100	99.2	8.6

Table 6.3. Accuracy of our machine learning based approach in detecting anti-adblocking scripts.

Results & Evaluation. Next we evaluate the effectiveness of classifier using our labeled set of anti-adblocking and non anti-adblocking scripts. To check the classifier performance we do 10-fold cross validation. We use 9-folds as training samples and the remaining fold as testing sample and repeat this process 10 times. We report results in the form of True Positive (TP) rate and False Positive (FP) rate. TP rate is the fraction of correctly classified anti-adblock scripts. FP rate is the fraction of incorrectly classified non anti-adblock scripts. Table 6.3 lists the classification results for different feature sets. We include SVM without AdaBoost for baseline comparison. The TP rate is above 99.2% for all configurations. The FP rate varies from 3.2% to 9.1% for different configurations. We achieve the highest TP rate of 99.7% and the lowest FP rate of 3.2% for AdaBoost + SVM classifier using top-1K features from the **keyword** feature set.

We further test our model on anti-adblocking scripts detected on Alexa top 100K live websites (§6.4). Out of 5,070 detected anti-adblocking websites, we extract 2,701 unique anti-adblocking scripts. We exclude the scripts in Alexa top 5K websites because they are used to train our model. We classify the 2,701 extracted scripts with the AdaBoost + SVM classification model with top-1K features configuration. We achieve a TP rate of 92.5%.

Our machine learning approach can be used in an offline or an online manner by adblockers. In the offline scenario, filter list authors can periodically crawl popular websites and run our trained model to identify new anti-adblock scripts. This will substantially reduce the manual labor required by filter list authors in maintaining filter lists as they only have to analyze the scripts detected by our model. The manual analysis of anti-adblocking scripts by filter list authors will help in reducing false positives that cause site breakage. In the online scenario, our trained machine learning model can be directly shipped in adblockers which would scan all scripts to detect and remove anti-adblock scripts on the fly.

6.6 Conclusion

In this chapter, we presented a retrospective measurement study of anti-adblocking. We reported that anti-adblocking has significantly increased over the last few years. Our analysis of anti-adblock filter lists revealed limitations of manually curated filter lists. We note that two popular anti-adblock filter lists are not only implemented differently, they have different coverages and update frequency. We also presented a machine learning based approach to automatically identify anti-adblock scripts. Our proposed approach can be used to augment the manual efforts filter list in an offline manner or incorporated in adblockers for online detection.

The tussle playing out between online publishers and adblockers is set to have a major impact on the Internet. Adblockers are changing the status quo of ad-driven monetization of online content and services. We believe that the large-scale retrospective analysis of anti-adblocking provided by our work is important to inform future discussions surrounding adblocking, both technical and economic.

CHAPTER 7

ShadowBlock: A LIGHTWEIGHT AND STEALTHY ADBLOCKING BROWSER

7.1 Introduction

The deployment of adblocking technology has been steadily increasing over the past few years. PageFair reports that more than 600 million devices globally use adblockers as of December 2016 [107]. Many reasons contribute to the popularity of adblocking. First, lots of websites show flashy and intrusive online ads that negatively impact user experience. Second, the pervasiveness of targeted or personalized ads has incentivized a global ecosystem of online trackers and data brokers, which in turn raises concerns for user privacy. Third, the inclusion of numerous advertising and tracking scripts causes excessive website bloat resulting in slower page loads. The rise of adblocking has jeopardized the ad-powered business model of many online publishers. For example, U.K. publishers lose nearly 3 billion GBP in revenue annually due to adblocking [108].

In response to adblocking, many publishers have deployed JavaScript-based, client-side anti-adblockers to detect and circumvent adblockers. An anti-adblocker typically consists of two components: detection and reaction. For adblocker detection, common practices include checking the absence of ad elements and proactively injecting bait ad elements [258]. Both practices exploit the fact that adblockers make observable changes to the DOM by either blocking relevant requests or hiding

DOM elements directly [299]. As a result, these DOM changes can be perceived by the detection part of anti-adblockers through invocation of JavaScript APIs such as `getElementById()`. After adblocker detection, the reaction component can perform different subsequent operations. It can be aggressive paywalls that prevent users from accessing the content or even switching ad sources.

Adblockers have addressed anti-adblockers in one of the following three ways: (i) blocking the JavaScript code of anti-adblockers using filter lists [215], (ii) disrupting anti-adblocker code based on program analysis [321], and (iii) hiding the trace of adblocking to fool anti-adblockers [299]. The first countermeasure is currently adopted by the adblocking community using filter lists such as Anti-Adblock Killer and Adblock Warning Removal [215]. However, the coverage and accuracy of these filter list is lacking. Our manual evaluation on 207 websites using anti-adblockers with visible reactions (i.e. warning message or paywall), only less than 30% of them are correctly identified by Adblock Warning Removal [110] or Anti-Adblock Killer [111]. This is likely due to the manual nature of filter list curation and maintenance which is cumbersome and error-prone. The second countermeasure of rewriting JavaScript to deactivate anti-adblockers is prone to false positives causing site breakage with unacceptable user experience degradation [321]. The third countermeasure of hiding the trace of adblocking, as implemented in prior work [299], is not stealthy because it injects new JavaScript which is easily detectable by anti-adblockers.

In this chapter, we aim to completely hide the traces of adblocking in a stealthy manner by going deep into the browser core. This is analogous to the rootkits in the

OS kernel where user applications are unable to detect the presence of a malicious process [280]. Since the browser core is at a lower level (more privileged), it is in theory capable of hiding the states of adblockers from the anti-adblockers while presenting an ad-free view to the user. Specifically, since anti-adblocker is implemented as client-side JavaScript, it can only access web-page states through a number of predefined Web APIs including the ones used to probe the the presence/absence of ad elements. These APIs are standardized by W3C and implemented eventually by web browsers. ShadowBlock hooks any JavaScript API that can potentially distinguish the difference before and after hiding ad elements, and assures no information about the element hiding can be leaked through such API.

We tackle three major challenges in designing and implementing ShadowBlock. First, existing adblockers' model of blocking ad-related URLs (e.g., scripts, iframes, images) does not fit well in our requirement of presenting the same exact DOM view as if no adblocker is employed. For example, if a DOM element is not even retrieved, ShadowBlock would have no way to fake its size, dimension, and other properties. Second, given that the Web APIs suite and the rendering process implemented in modern web browsers are highly complex and intertwined, there may exist unexpected channels that leak information about adblocking deployment. To achieve stealthy adblocking, we need to ensure that no such channel discloses differentiable information about our element hiding action in a conclusive way. Third, modern web browsers make significant efforts in improving their page loading and rendering performance. As we develop ShadowBlock on open-sourced Chromium, we need to minimize the overhead it incurs during the page load process.

Contributions. We summarize our key contributions as follows.

1. We design a well-reasoned solution where we present two different views to anti-adblockers and users. On one hand, ad elements are never directly blocked (so they remain visible to anti-adblockers); on the other hand, these ads are stealthily hidden from users.
2. We reuse the rules from filter lists used by adblocking browser extensions to element hiding decisions. On top of existing lists that are community-backed and have been widely adopted, we replicate 98.3% of their ad coverage according to manual inspection over Alexa Top 1K websites, with less than 0.6% breakage rate.
3. We design and implement a fully functional prototype which is open-sourced at the time of publication. We evaluate the effectiveness of ShadowBlock prototype on 207 websites with visible anti-adblockers. We pick real anti-adblockers with different trigger mechanisms and of different complexity. All of them are successfully evaded by our new adblocker design.
4. Our performance evaluation of ShadowBlock shows that it loads pages comparably fast as Adblock Plus on average, in terms of Page Load Time and SpeedIndex.

Chapter Organization The rest of the chapter is organized as follows. Section 7.2 summarizes background and related work. Section 7.3 presents the overview of our system and discusses challenges encountered and tackled during the development of ShadowBlock in detail. Section 7.4 presents experimental evaluation. Section 7.5 discusses limitation of ShadowBlock. Section 7.6 concludes the paper.

7.2 Background and Related Work

7.2.1 Adblockers And Filter Lists

Mechanisms. Adblockers rely on manually curated filter lists to identify ads on web pages. EasyList and EasyPrivacy are two most widely used filter lists to block online ads and trackers, with about 71000 and 15000 rules [114], respectively. These lists consist of two types of rules which are basically regular expressions. One of them is HTTP rules that block HTTP requests to URL addresses that are known to serve ads. For example, the first filter rule below blocks all third-party HTTP requests to `amazon-adsystem.com`, preventing any resource on this domain from being downloaded. The other type is HTML rules, which generally hides HTML elements that are identified as ads. For example, the second filter below hides all HTML elements with ID `promo_container` on `wsj.com`.

```
||amazon-adsystem.com^$third-party
wsj.com###promo_container
```

It is noteworthy that HTML rules are mostly only introduced to complement HTTP rules while dealing with first-party text ads. This is because these text ads are directly embedded into the HTML itself with no associated additional resource loads, making it inevitable to be included while the browser downloads the web page. Otherwise, HTTP rules are preferred as they prevent ad resources from being loaded in the first place, saving unnecessary network traffic and avoiding execution of ad-related scripts to speed up page loading and rendering.

Limitations. A group of volunteers that maintain filter lists carry out the manual process to add new rules, correct and remove erroneous or redundant rules all based on informal feedback from users [113]. Due to its crowd-sourced nature, this laborious effort faces challenges from both the completeness and soundness. In the context of adblocking, the former results in missed ads and the latter often translates to site breakage or malfunction [129]. At the same time, as adblockers gain their popularity rapidly (11% of global Internet population is blocking ads as of December 2016 [107]), online publishers are also fast adopting countermeasures against adblockers that we summarize below.

7.2.2 Countermeasures Against Adblockers

Concealing ad signatures. First, online advertisers can bypass rules on filter lists by concealing the signatures these lists use to identify ads. At a high level, this line of countermeasures attempts include first-party advertising and rotation of ad serving domains. First-party advertising exploits the fact that many rules on filter lists are designed to block ads from being loaded from third-party servers. Instead, ads are served from the same domain of the web page hosting them and their nature as ads is concealed as normal content [117]. However, adblockers can easily hide any HTML element on a web page by applying HTML rules that are crafted to target elements based on any combination of their CSS/HTML properties. In other words, any CSS selectors used to create and/or locate ads elements, can also let adblockers identify and hide these elements in turn.

Domain name rotation is another tactic for obfuscating advertising content. It relies on ad rotation networks that serve ads from frequently-changing, or even automatically generated [308] domain names, which overwhelms volunteers that maintain the filter lists so they are hard to keep pace with the rule updates. This can result in, however, the indifferent blocking of all third-party resources on websites that show such ads [115]. By only whitelisting legitimate scripts that support core functionalities, any possible ads or tracking JavaScript are prevented from running, leaving no chance for loading domain rotating ads. Moreover, AdBlock Plus recently launched its Anti-Circumvention Filter List [109] that specifically counter "circumvention ads", including ads adapting the two countermeasures above.

Deploying anti-adblockers. Second, many publishers choose to deploy anti-adblocker JavaScript code to battle the rise of adblocking. Specifically, such client-side scripts consist of two main components, *trigger* that detects the presence of adblockers by checking whether ads or bait elements are still present, and *reaction* that can display warning messages and/or simply report the results to a remote server [258, 321]. Prior work [215] showed that 686 out of Alexa top-100K websites detect and visibly react to adblockers on their homepages. Even worse, these visible ones only account for less than 10% of all anti-adblockers [321]. Zhu et al. [321] showed that among Alexa top-10k websites, 30.5% are countering adblockers in some form, with most of them being silent reporting. In summary, because of their flexibility and ease of deployment, anti-adblockers are considered the most widely used countermeasure against adblockers adopted by online publishers.

7.2.3 Countermeasures Against Anti-adblockers

Dedicated anti-adblocking filter lists. As a response, adblockers attempt to circumvent anti-adblockers by blocking their JavaScript code snippets, whitelisting bait scripts/elements, or hiding warning notifications. To this end, adblockers once again rely on manually curated filter lists such as Anti-Adblock Killer [111] and Adblock Warning Removal [110]. These lists either trick anti-adblockers' trigger so they cannot detect adblockers, or mute their reaction component to prevent responses after successful detection.

```
/kill-adblock/js/function.js$script
@@||removeadblock.com/js/show_ads.js$script
ilix.in,urlink.at,priva.us###blockMsg
```

For example, the first HTTP rule above blocks the code snippets containing implementation of an anti-adblocking library `KillAdBlock`, and the second whitelists a bait script file named `show_ads.js` that is used to detect adblockers. The third HTML rule hides the warning message with ID `blockMsg` issued by the associated anti-adblocker. However, our manual evaluation (§ 7.4) shows that these lists targeting anti-adblockers are generally ineffective. Only less than 30% of the anti-adblocking warning messages can be removed by the state-of-the-art filter lists. This is again partly because of the crowdsourcing nature of these lists, and also the rising popularity of third-party anti-adblocking services that deploy sophisticated techniques dedicated for detecting/circumventing adblockers [258].

Disrupting anti-adblocker code. Other than the filter lists that have been officially adopted by adblockers, there are also research efforts for detecting and evading anti-adblockers. One solution to measure the anti-adblockers is to perform program analysis techniques that automatically determine if a script functions for anti-adblocking purposes. Such analysis can be static that is based on syntactic and structural features extracted from JavaScript code, and utilizes machine learning approaches to classify the code from ground-truth-labeled training data [215]. It can also be dynamic that captures JavaScript behavior at runtime by collecting and analyzing differential execution trace with the adblocker turned on and off [321]. After successfully pinpointing the critical conditions that are used by anti-adblockers to assert/react against the presence of adblockers, one can choose to rewrite these conditions to prevent the anti-adblockers from functioning. This approach is generally intrusive (patching Javascript can be tricky and cause breakage) and easy to evade. Indeed, the overall success rate of this strategy is shown to be only less than 80%.

Hiding adblockers. Besides disrupting the functionalities of anti-adblockers, researchers also have proposed a way to hide the trace of using adblockers, or known as *stealthy adblocking*. In [299], Storey et al. created a shadow copy of the DOM that anti-adblockers operate on before any adblocking actions take place, and then redirects all JavaScript APIs (e.g. `getElementById()`) that can be used to detect the presence of ad elements to the copy instead of the original DOM. However, this so-called rootkit-style stealthy adblocker has inherent drawbacks. First, unless it lives in browser core and with significant engineering efforts, the underlying DOM mirroring and propagation are difficult to be complete in all cases. This is especially problematic in the context of web browsing as any site breakage causes unacceptable user experience degradation. Second, even with a perfect implementation, maintaining a live copy of complicated data structures such as DOM poses a prohibitively high overhead onto the rendering performance of modern web browsers. Given that modern browsers place significant emphasis on performance, heavy operations like such at runtime are generally not acceptable.

7.3 ShadowBlock

In this section, we first provide an overview of ShadowBlock's architecture. We then discuss ShadowBlock's two building blocks: (1) the identification of ad elements by translating filter list rules to per-element hiding decisions and (2) the concealment of our hiding actions. Finally, we summarize the modifications we make in the relevant modules of Chromium.

7.3.1 ShadowBlock Overview

Figure 7.1 illustrates ShadowBlock's architecture. It consists of two subsystems: one translates rules from filter lists and use them for identifying ad elements in DOM to hide; the other hooks necessary points in Chromium to ensure that the hiding actions are transparent to the trigger/detection component of anti-adblockers. Recall from Section 7.2.1 that filter lists contain tens of thousands of rules that either block HTTP requests to fetch ad resources or hide HTML ad elements. To prevent exposing adblocking actions to anti-adblockers, we need to hide the changes in DOM or other states (e.g. resource loads) introduced by adblocking because these changes

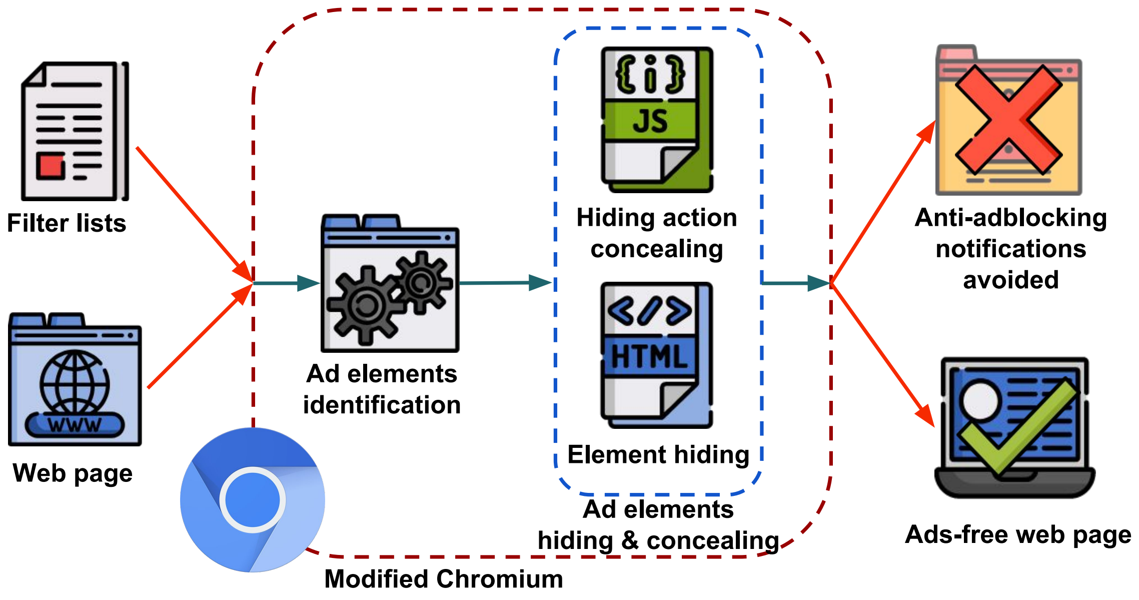

Filter lists
Web page
Ad elements identification
Hiding action concealing
Element hiding
Ad elements hiding & concealing
Modified Chromium
Anti-adblocking notifications avoided
Ads-free web page

Figure 7.1. Architectural overview of ShadowBlock

can be detected by anti-adblockers through JavaScript APIs such as `getElement-ById()`. In order to do so, we first allow all HTTP requests to proceed, then mark any element that results in ads, and subsequently hide the marked elements. It is important that we allow these elements to be retrieved so when the anti-adblocking script queries the state of the element, ShadowBlock will be able to generate valid responses (e.g., dimension of the element).

7.3.2 Identifying Ad Elements

Next, we explain our approaches for marking different types of ad elements. In general, there are two types of elements: (1) those that are statically embedded in the HTML, and (2) those that are dynamically created by JavaScript. First, some ad elements are statically embedded in the HTML, including ad images, iframes, media files (i.e. video and audio). Such ad elements can be typically identified by matching against the HTML rules on filter lists. Second, ad scripts usually *create* new ad elements that display advertising content. These dynamically created elements should be identified, marked, and hidden.

Ad elements loaded statically. ShadowBlock does not need to do anything special for such elements. Ad filter lists already contain extensive rules that cover them. For example, the rule `||googlesyndication.com/safeframe/` in EasyList blocks the HTML file from `https://tpc.googlesyndication.com/safeframe/1-0-23/html/container.html` on site `cnbc.com`, which prevents a container frame used by Google Ads from being loaded. To hide such an ad element, ShadowBlock needs to first identify the iframe element with the blocked URL as its source attribute and then hide it.

Alternatively, Easylist rules may hide ads based on element properties (e.g., element id). We simply reuse these rules to match elements in the page, and mark them accordingly.

Ad elements generated dynamically by ad scripts. There are generally two cases. The easy and the hard. For the easy case, the dynamically generated ads may contain URLs or ids that already show up on Easylist. This allows us to directly mark them as ads using very much the same strategy as mentioned above.

For the hard case, the dynamically generated elements are not on Easylist. This is because it is assumed that ad scripts are blocked upfront and therefore there is no need to block the elements generated by them. In ShadowBlock, in contrast, we need to allow ad scripts to load and execute, which mandates us to track such elements.

To identify elements dynamically created by ad scripts, we need to attribute each element to the script that created it. More formally, we can define this process of attribution as tracking the *data provenance* of HTML elements using taint analysis. Note that there are in general two types of element creation that can be initiated by a script: (1) control-flow-based creation, in which the script directly invokes JavaScript API (e.g. `createElement(tagName)`) and only propagates data from itself into the new element; and (2) data-flow-based creation, in which the script uses data from sources other than itself into the new element (e.g. `createElement(fetchTagNameFromServer())`). Dynamic taint analysis [228] can accurately track the data provenance through taint propagation for both types. Simply put, taint analysis involves *taint source* (where data comes from), *taint sink* (where data ends), and *propagation policies* that define how that tainted data are propagated through the program execution. In our case, all data derived from an identified ad script as "tainted" (i.e., data downloaded through an ad URL, or retrieved/generated by an ad script), then such data can be tracked standard taint analysis propagation policies for JavaScript (e.g., [163]). Finally we will hide any tainted HTML elements (i.e. taint sinks).

Unfortunately, such dynamic taint analysis at runtime usually incurs significant overhead for web browsing [228]. In addition, we argue that the cases where taint analysis will be required are limited. Specifically, even if ad elements take dynamically fetched data, e.g., `createElement(fetchTagNameFromServer())`, they are most likely created by ad scripts. This is sufficient for us to mark the element and hide it (irrespective of what data are actually fetched from the server). The reasoning is that if we consider extension-based adblockers as our baseline, the dynamically created element wouldn't even exist in the first place (as the script as a whole would have been blocked). Based on the above, we devise a simple technique which we call *execution projection* using the call stack information extensively (which are used for various other purposes as well [216, 243]). At a high level, we maintain an "execution stack" that keeps track of what scripts are being executed at any given time point, and mark an element as ad if there is *any* ad script in the stack when the element is being created. For example, consider a simple ad script `ad_loader.js` in Code 7.1.

```
var ad_img = document.createElement("img");
ad_img.src = "https://some_ad_publisher.com/ad.jpg";
document.body.appendChild(ad_img);
```

Code 7.1. Example ad script `ad_loader.js`

In this example, at the time of the image element creation, `ad_loader.js` would be at the top of the execution stack because it is being executed.

After attributing elements to ad scripts, which are identified using filter lists, we can mark all ad elements and hide them accordingly. We illustrate this projection/marking process in Figure 7.2. In more complex cases, there can be many scripts in the executing stack, because code in one script can invoke functions in other scripts, and so on. More importantly, ad scripts can invoke non-ad libraries (e.g. jQuery) to create ad elements so the top script in stack at the time of element creation is not necessarily ad script. To tackle this challenge, we need to scan the entire execution

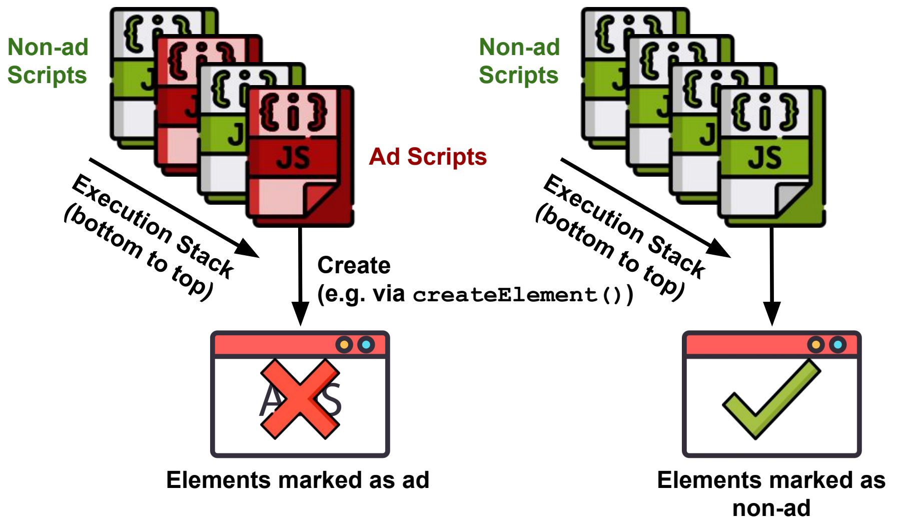

Non-ad Scripts
Ad Scripts
Execution Stack (bottom to top)
Create (e.g. via createElement())
Elements marked as ad
Non-ad Scripts
Execution Stack (bottom to top)
Elements marked as non-ad

Figure 7.2. Execution projection for marking script-created ad elements

stack. If there is any ad script in the execution/invocation chain, ShadowBlock should mark the element being created as ad. This is because any script (ad-related or not) invoked by a known ad script should never have been executed in the first place, given that adblocking extensions simply block the whole ad script altogether.

Note that our approximate solution does not handle a special case when an element is created via a JavaScript API that gets overridden (hooked) by an ad scripts . Since client-side JavaScript is allowed to override (i.e. injecting code containing a callback to its own) arbitrary JavaScript API functions (e.g. `createElement()`) with its own version, we would see ad script in the stack when an element is being created via the API overridden by an ad script. In this case, we may mistakenly mark a non-ad element as ad when the overriding ad script does not propagate any data into the newly created element. To tackle this issue, at the time of element creation, we would need to further check whether the code injected by the overriding ad script alters the element. If it alters the element then we mark the element as ad, and non-ad otherwise.

```
var original = document.createElement;
document.createElement = function (tag) {
  new_elem = original.call(document, tag);
  report_statistic_to_server(new_elem);
  return new_elem;
};
```

Code 7.2. API overriding with the element intact

```
var original = document.createElement;
document.createElement = function (tag) {
  new_elem = original.call(document, tag);
  ad_elem = change_to_ad_elem(new_elem);
  return ad_elem;
};
```

Code 7.3. API overriding with the element altered

7.3.3 Stealthily Hiding Ad Elements

After identifying ad elements, ShadowBlock needs to stealthily hide them so as to not leaving its trace to anti-adblockers. Next, we discuss how we realize such stealthiness through CSS property access API hooking.

Choice of hiding mechanism. We first need to decide how to hide ad elements within Chromium. Given the complexity of modern web browsers and API standards, we can hide an HTML element in several different ways. To better understand different hiding mechanisms, we illustrate Blink's rendering process in Figure 7.3 [122]. Blink's rendering path, from parsing an HTML file to the pixel display on user's screen, can be summarized in the following phases.

1. **Parse** flat HTML and CSS in plain-text to DOM and CSS Object Model (CSSOM) in tree structure.

2. **Combine** DOM and CSSOM to Render Tree, which captures all the visible DOM content and all the CSSOM style information for each node.

3. **Paint** the rendered pixels to user's display according to Render Tree.

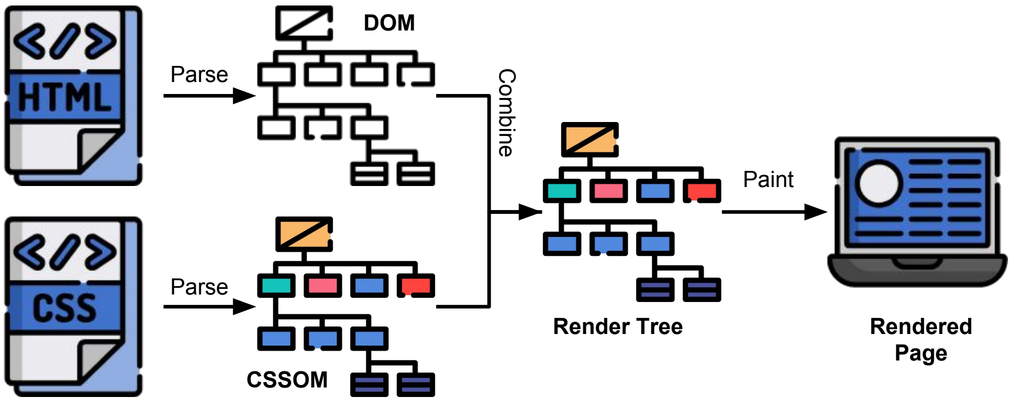


Figure 7.3. Rendering Path for Blink

In theory, each of these 3 phases contain APIs/modules that we can leverage for hiding an HTML element (or in other words, preventing it from being rendered). We outline different possible strategies to hide HTML elements below:

1. **DOM/CSS layer**: (i) remove the element from DOM; (ii) set the element's style to `display:none`, `visibility:hidden` or `opacity:0`

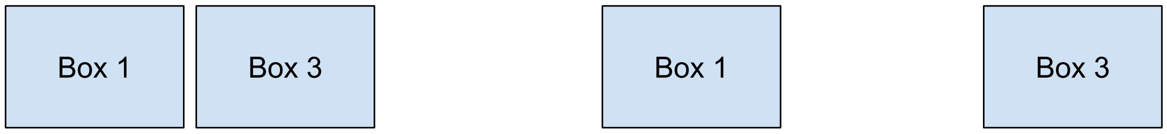
2. **Render Tree layer**: remove the `LayoutObject` (i.e. a styled node) from Render Tree

3. **Paint layer**: prevent the region in pixels associated with the element from being painted

Note that generally, the higher-layer (i.e. closer to DOM/CSS layer) we tweak around in the hierarchy, more engineering effort is required for ensuring its stealthiness against anti-adblockers, while narrower gap we have to bridge for translating the identified ad HTML elements to the data structure corresponding to that layer (e.g. CSS properties for DOM/CSS layer, `PaintBlock` for Paint layer etc.). In contrary, the lower-layer (i.e. closer to Paint layer) our modifications reside, fewer unexpected channels there are that can potentially leak the hiding activities, but at the same time more engineering efforts are required for translating the identified ad elements to that layer's data structure.

After investigating different possibilities, we find the CSS property `visibility:hidden` achieves the most suitable trade-off for our objective. It persists in phase (1), functions in phase (2) of the rendering path, and eventually affects both Render Tree and painted/rendered page in phase (2) and (3). On one hand, `visibility` is a property associated with every HTML element so we can easily identify after marking its effective element as ad, without the need of further tracing. On the other hand, unlike `display:none`, it by design preserves the space taken up by the hidden element so it causes no side-effect to the layout of the document, minimizing the impacted points that need to be hooked for covering the hiding action.

Figure 7.4 shows the visual difference between their respective effects. Compared to `opacity:0` that also preserves the occupied space, `visibility` is a categori-



(a) `display:none` (b) `visibility:hidden`

Figure 7.4. Comparison of toggling different CSS properties (Box 2 is hidden)

cal CSS property instead of numerical as `opacity` so its implementation in Chromium is considerably less complex, simplifying our hooking logic as well.

Hooking for concealing hiding actions. After hiding the target ad element by setting its `visibility` CSS property value [127] to `hidden`, we next need to cover any traces that result from this change of value and can be detected by anti-adblockers. Since anti-adblocking scripts are client-side JavaScript code, which can only access the change of states happened in the page through JavaScript Web APIs [128], we search through the source code of Chromium, analyze relevant modules and locate the following three categories of APIs that are impacted by the `visibility` CSS property:

- **CSS/Style-related**: changing a CSS property value immediately affects its own return value to JavaScript APIs via `getComputedStyle()`. We hook this value to `visible` to fool anti-adblockers. Fortunately, for our case, setting `visibility:hidden` preserves the space the element occupies, so it does not collapse the page layout nor affect other relevant CSS properties such as `off-setHeight/offsetWidth`.

- **Event-related**: flipping the `visibility` property prevents an element from receiving some DOM events, such as `onfocus`. Since anti-adblockers can leverage these events as a side-channel to infer the real visibility of an element, we hook relevant modules in Chromium so hidden elements can receive these events just like visible elements.

- **Hit-testing-related**: another effect of setting `visibility: hidden` is Blink treats elements with it as inapplicable to Hit Testing, which checks if an element is clickable by users in their viewpoints. This removes the hidden element from return values to APIs like `elementFromPoint()`, which can potentially be used by anti-adblockers to differentiate hidden elements from visible ones. We hook relevant modules in Blink to cover it.

Since all JavaScript APIs, to our best knowledge, directly or indirectly rely on the modules above to determine an element's visibility, we ensure the completeness of our hooking against potential information leakage to client-side anti-adblockers. It is worth noting that to avoid affecting existing `visibility:hidden` CSS property value, in Chromium we create a new `visibility:fake-visible` value that completely mimics what `visibility:hidden` behaves, except for the points that are intentionally hooked. Moreover, since `visibility` is a CSS property that inherits from parent node to child nodes, we can make the identified ad elements invisible to user's display even though we only identify the top-most element as ad according to filter lists.

7.3.4 Chromium Modification

Next, we describe our modifications to Chromium for implementing ShadowBlock. We start with a brief introduction of Chromium's architecture, then move to the instrumentation we use for identifying ad elements, and lastly discuss the modules we leverage for hiding identified ad elements and hook for eliminating the traces resulted from the hiding action. We implement the prototype of ShadowBlock with 1307 LOC (1265 LOC addition and 42 LOC deletion) in C++ on top of Chromium [1]. Note that we re-use `SubresourceFilter` [124] and `libadblockplus` [116] for parsing filter lists with production-level robustness.

Chromium Architecture. Chromium's rendering engine is called Blink and its JavaScript engine is called V8. Blink [112] is responsible for fulfilling the rendering path shown in Figure 7.3, in which its core module renders all HTML elements and handles their `visibility` CSS properties we use for hiding ad elements. V8 [125] handles the compilation and execution of all JavaScript code, including the ad scripts ShadowBlock needs to identify and anti-adblocker scripts that intend to detect our hiding action over ad elements. Blink has a bindings module to handle interactions between rendering and JavaScript execution. Rendering related script tasks are passed through bindings module. For example, JavaScript API calls such as `getComputedStyle()` are handled through the bindings module.

Instrumentation for identifying ad elements. As discussed in Section 7.3.2,

[1] We open source our implementation at `https://github.com/seclab-ucr/ShadowBlock` to allow reproducibility as well as help future extensions by the research community.

there are three types of ad elements ShadowBlock needs to identify.

First, for ad elements generated by ad scripts, we rely on execution projection. In Chromium, we leverage `blink::SourceLocation` `::Capture` to capture the full `v8::v8_inspector::V8StackTrace` that includes the entire JavaScript call stack[2] at any given time point. It serves as the underlying stack tracing mechanism for V8's debugger/inspector, and has therefore been optimized with low overhead [126].

Second, we instrument the constructor of `blink::Element` class in Blink, which captures the earliest point of creation event for all HTML elements. Because of V8's single-threading nature, we can safely associate the current stack trace to the element creation event, and scan the stack to match the scripts in it against filter lists. If it is a match, we then mark the element as ad. Additionally, we instrument the `DispatchWillSendRequest` event in both `blink::FrameFetchContext` and `blink::WorkerFetchContext` to intercept the point when an ad script loads another script, and mark the loaded script as ad script. By adding such loaded ad scripts to a set, we match the stack trace against them as well at element creation, ensuring we can mark all ad elements.

Third, for elements loaded with resources that match rules in the filter lists, we intercept the `AttributeChanged` event in `blink::` `Element` and match the URL against filter lists, if it is a match we mark this el-

[2]We admit that there are cases where asynchronous tasks are not correctly captured by the default V8 call stack trace. However, we argue this incompleteness only translates to very limited number of missing ads according to our manual evaluation in Section 7.4.2.

ement as ad. For element hiding rules, we adopt `libadblockplus` [116], a C++ wrapper library around the core functionality of Adblock Plus to parse filter lists and generate the CSS selectors for matching ad elements for a particular domain. Then, we mark the ad elements that match the generated CSS selectors by calling `ContainerNode::QuerySelectorAll()`.

Since many web pages are dynamic due to JavaScript execution over time, we also need to monitor attribute changes of each element. For this purpose, we instrument the `AttributeChanged` event and match any element with newly changed attributes against CSS selectors from HTML rules. We mark an element as ad if it is a match, or un-mark the element if this element has been marked but it is not matched this time. Note that in order for minimizing the number of matches needed to perform, we conduct the first batch match (via `QuerySelectorAll()`) after the `load` event of DOM is fired, and then match elements upon their attribute changes. This design choice leaves a short period of time (few milliseconds) between page navigation and `load` DOM event in which ads are displayed. We make this trade off to reduce the overhead incurred by `QuerySelectorAll()`. In comparison, adblocking extensions such as Adblock Plus inject CSS rules when `document.readyState` turns `interactive` [132], which happens before the `load` event. However, it is important to note that most ads in current web ecosystem are loaded in an asynchronous manner and are unlikely to appear before the `load` event in first place.

Stealthy modifications for hiding ad elements. As mentioned earlier, we leverage `visibility` CSS property to hide identified ad elements by creating a new `fake-`

`visible` enumerate and visually hide elements with this enumerate, as if it behaves as `hidden`. In the meantime, we hook relevant modules in both Blink and its bindings with V8 to ensure the stealthiness of our hiding action. More specifically, for eliminating traces accessible by CSS/Style-related APIs, we hook `CSSComputedStyleDeclaration::GetPropertyCSSValue` in Blink and force return `visible` to queries about hidden elements. For event-related APIs, we hook `Element::IsFocusableStyle()` and other conditions that determine if an element can receive events. Lastly, we hook `ComputedStyle::VisibleToHitTesting()` so ad elements are still regarded "visible" from the viewpoint perspective of Blink. In principle, our hooking guarantees that the identified ad elements are invisible to user's display as pixels on screen but appear as visible to APIs accessible to client-side JavaScript.

7.4 Evaluation

We evaluate ShadowBlock in terms of its (1) stealthiness against anti-adblockers, (2) ad coverage, and (3) performance as as compared to adblocking extensions.

7.4.1 Stealthiness Analysis

Takeaway: ShadowBlock has 100% success rate against anti-adblockers whereas state-of-the-art anti-adblocking filter lists have only 29% success rate.

Experimental Setup. To evaluate the stealthiness of ShadowBlock, we use previously reported [321] 682 websites with visual anti-adblockers. We manually analyze these websites and find that 207 of them still use visible anti-adblockers. For each website, we perform stealthiness comparison as follows.

1. Open a website with four Chromium instances simultaneously. Each instance has a different profile configuration: (i) no modification or extension; (ii) Adblock Plus extension with EasyList only; (iii) Adblock Plus extension with EasyList, Anti-Adblock Killer list, and Adblock Warning Removal list; and (iv) ShadowBlock using EasyList.

2. Scroll the page down to the bottom and wait for 30 seconds after the `load` event has fired to ensure complete page load.

3. Capture the full-page screenshots (including content after scrolldown) for all browser instances.

4. Manually inspect the screenshots: compare (i) and (ii) to determine if the page has visual anti-adblocker. If so, further compare (ii) and (iii) to check whether anti-adblocking filter lists evade the anti-adblocker, compare (iii) and (iv) to check whether ShadowBlock achieves the evasion.

Results. In addition to visible anti-adblocking notifications, we also consider ad switching and crypto-mining reactions from websites. Table 7.1 compares ShadowBlock with anti-adblocking filter lists for each of these anti-adblocking reactions. "Notification" refers to websites that show anti-adblocking notifications such as paywalls. "Ad switching" refers to websites that switch their ad sources upon detection of adblockers. "Crypto-mining" refers to the websites that load crypto-mining scripts

Tool	Notification	Ad switching	Crypto-mining
Total	201	5	1
ShadowBlock	201 (100%)	5 (100%)	1 (100%)
Filter lists	59 (29%)	1 (20%)	0 (0%)

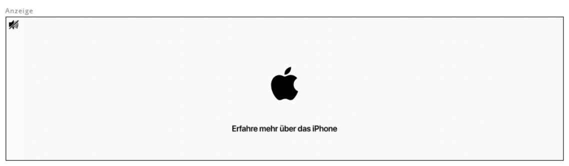
Table 7.1. Breakdown of stealthiness analysis

to mine crypto-currencies on detection of adblockers [206]. We note that ShadowBlock has 100% success rate as compared to 29% success rate of anti-adblocking filter lists.

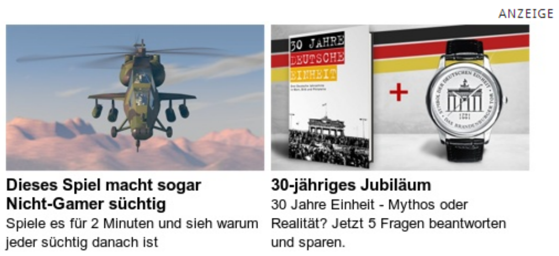
Case Studies. Below we discuss a few interesting examples of anti-adblockers that ShadowBlock successfully handles but filter lists do not. Note that besides visible anti-adblocking notifications, we also include one example discovered in the wild that uses non-visual countermeasure against adblocker users.

Ad source switching. On detecting adblockers, some websites switch their ad sources to sources that are currently not blocked by filter lists. Figure 7.5a and 7.5b show an example from `golem.de`. Since ShadowBlock stealthily hides the original ads, the ad source switching script is never triggered. Therefore, unlike what Figure 7.5b shows in which adblocking extensions fail to remove the replaced ad, ShadowBlock successfully hides it.

Silent reporting. Besides visible reaction, anti-adblockers can also choose to silently report the adblocking status to back-end servers to collect adblocking statistics. For example, `varmatin.com` uses Code 7.4 to place a bait with keywords on EasyList to track adblocking users and report the status to back-end server. ShadowBlock



(a) Original ad


ANZEIGE
Dieses Spiel macht sogar Nicht-Gamer süchtig
Spiele es für 2 Minuten und sieh warum jeder süchtig danach ist
30-jähriges Jubiläum
30 Jahre Einheit - Mythos oder Realität? Jetzt 5 Fragen beantworten und sparen.

(b) Replaced ad

Figure 7.5. Ad switching behavior on golem.de

handles such cases and these statistics are never reported.

```
function checkAds() {
  if ($(document.getElementById('adsense')).css('
      display')
  !== 'none' && $('#myadsblock').length === 1) {
    dataLayer.push({
      'DimAdBlock': 'Unblocked'
    });
    window.adblockdetected = false;
  } else {
    dataLayer.push({
      'DimAdBlock': 'Blocked'
    });
    window.adblockdetected = true;
  }
}
```

Code 7.4. Code snippet on `varmatin.com` of silent anti-adblocker

Crypto-currency mining. Some websites have started to employ cryptojacking as a response to adblocking [206]. To this end, websites use anti-adblockers to detect use of adblockers and load scripts to mine a crypto-currency on user's browser. Mining crypto-currencies consumes processing power on user's machine. For example, `know` let3389.blogspot.com blocks organic content on detection of adblockers and asks users of allow crypto-currency mining for monetization instead. Code 7.5 shows the crypto-currency mining script on knowlet3389.blogspot.com.

```
setInterval(function() {
  try {
    var a3 = document.getElementById('AdSense3');
    if (a3.offsetHeight < 33  a3.clientHeight < 33) {
      throw "Fuck U AdBlock!";
    }
  } catch (err) {
    miner.start(CoinHive.IF_EXCLUSIVE_TAB);
  }
}, 5487);
```

Code 7.5. Code snippet on knowlet3389.blogspot.com for mining crypto-currency

7.4.2 Ad Coverage Analysis

Takeaway: ShadowBlock achieves 97.7% accuracy, with 98.2% recall and 99.5% precision in blocking ads on Alex top-1K websites.

Experimental Setup. For ad coverage analysis, we use ShadowBlock on Alexa top-1K sites and measure its accuracy in terms of true positive (TP), false negative (FN), true negative (TN), and false positive (FP). We define TP, FN, TN, and FP as:

TP: All ad elements on a page are correctly hidden.

FN: At least one ad element on a page is not hidden.

TN: No non-ad element on a page is incorrectly hidden.

FP: At least one non-ad element on a page is incorrectly hidden.

For each website, we perform ad coverage comparison as follows.

1. Open a website with three Chromium instances simultaneously. Each profile has a different profile configuration: (i) no modification or extension; (ii) Adblock Plus extension with EasyList; and (iii) ShadowBlock using EasyList.

2. Scroll the page down to the bottom and wait for 30 seconds after the `load` event has fired to ensure complete page load.

3. Capture the full-page screenshots (including content after scrolldown) for all browser instances.

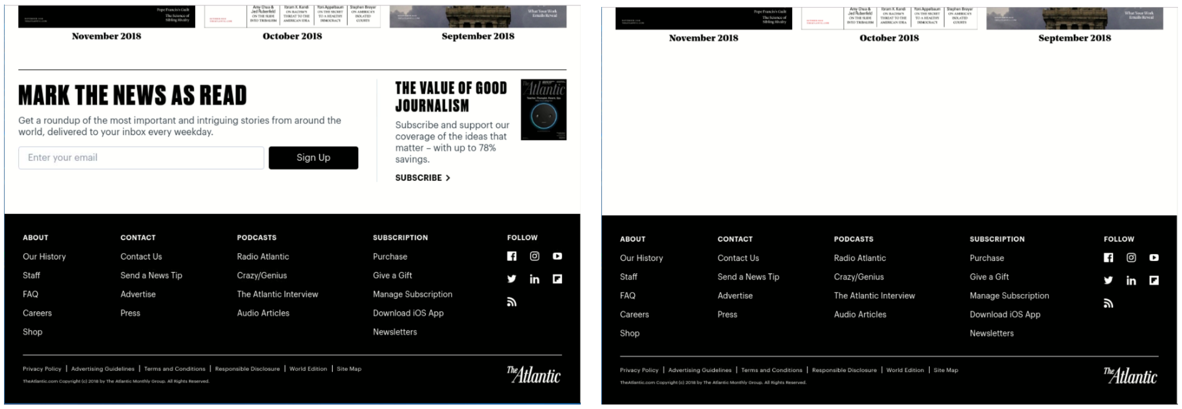
4. Manually inspect the screenshots: compare (i), (ii) and (iii) to determine if the website has any FPs or FNs.

Results. Table 7.2 shows the breakdown of our manual analysis. We evaluate results in terms of TPs, FNs, TNs, and FPs. Note that we are able to perform our analysis on 943 out of Alexa top-1000 websites. The remaining websites failed to properly load primarily due to server-side errors (e.g., 404).

Event	**TP**	**FN**	**TN**	**FP**
Count	926 (98.2%)	17 (2.8%)	938 (99.5%)	5 (0.5%)

Table 7.2. Breakdown of ad coverage analysis

False Positive Analysis. From Table 7.2, we note that ShadowBlock has only 0.5% false positive rate. However, they are still critical as they might lead to user experience degradation. We further investigate false positives to diagnose their root cause.



MARK THE NEWS AS READ

Get a roundup of the most important and intriguing stories from around the world, delivered to your inbox every weekday.

THE VALUE OF GOOD JOURNALISM

Subscribe and support our coverage of the ideas that matter – with up to 78% savings.

(a) Adblock Plus (b) ShadowBlock

Figure 7.6. Minor visual breakage caused by ShadowBlock

theatlantic.com is an example of false positive. Figure 7.6 shows that ShadowBlock incorrectly hides organic content at the bottom of the page. On further investigation, we find that an ad script ads.min.js loads another script script.js that hooks JavaScript API methods. In this case, even when a non-ad element is being processed it would go through same hooked JavaScript API methods. Since our ad marking heuristics check for the presence of ad scripts on execution stack it will incorrectly mark such elements as ads. In comparison, extension-based adblockers block the request for downloading ads.min.js in the first place, so the hooking script never gets executed. As discussed in Section 7.3.2, we can deal with this issue by checking whether or not the overridden API alters the elements and hiding the elements altered by ad scripts.

False Negative Analysis. From Table 7.2, it can be seen that ShadowBlock has only 2.8% FNs. We further investigate FNs and identify that they are again caused by corner cases not covered by ShadowBlock and that they can be handled by performing taint analysis.

`sohu.com` is an example of false negative. On further investigation, we find that `sohu.com` uses a non-ad script (not on Easylist) to load both ads and non-ad content on the page. Since ShadowBlock only attributes elements created by ad scripts as ads, it misses dual-purpose scripts. It's noteworthy that this should be a rare case, as it is contrary to the common practice of using dedicated third-party scripts to create and load ad elements that most ad publishers exercise today. These publishers normally deploy third-party ad scripts because they have a complex bidding system and prefer dominant control over their ad modules

```
"resource": {
  "type": "text",
  "text": "Guangzhou, Audi TT 82.2K RMB off",
  "md5": "",
  "click": "http://dealer.auto.sohu.com/882054/
      promotion/article?id=7360579",
  "imp": [],
  "clkm": [],
  "adcode": "Guangzhou, Audi TT 82.2K RMB off",
  "itemspaceid": "15770"
}
```

Code 7.6. JSON snippet on `sohu.com` for loading ads (translated from Chinese)

We can tackle this issue by implementing the taint analysis approach discussed in Section 7.3.2. Specifically, Code 7.6 shows the snippet of a JSON file on `sohu.com` containing parameters required to create ad elements. In this case, we will need to first mark the JSON object as ad-related, or tainted, and whenever any piece of the data derived from it propagates to any element field (e.g. the URL in JSON's `click` field is used to set an element's `src` attribute), we mark the element as ad. In comparison, extension-based adblockers intercept the network request to load such ad JSON based on its URL in the first place, which effectively prevents the resulting ad HTML element from being created.

Similarly, we observe FNs on `youtube.com` where ShadowBlock is unable to hide all video ads. Our manual analysis shows that `youtube.com` leverages the Media Source Extensions (MSE) API [131] to load video segments through AJAX requests as byte streams. Unlike the standard HTML `video` tag that loads videos as HTTP requests, `youtube.com` loads ad videos in `Blob` objects [130] which are downloaded by JavaScript on the fly. ShadowBlock cannot identify video ads loaded as `Blob` objects, because both ad and non-ad objects are generated by the same non-ad script and assigned to a single HTML `video` element. Unlike our strategy that relies on differentiating ad scripts, extension-based adblockers block the AJAX requests to fetch ad video segments based on their URLs, which achieves the goal of ad removal. As discussed earlier, taint tracking can be used to address this challenge.

Even through we show that tainting is the ultimate solution to the FN cases encountered during our evaluation, we argue that it a comprehensive taint engine poses prohibitively high runtime overhead in the context of web browsing [163, 228]. More importantly, our evaluations have shown the sufficient accuracy of ShadowBlock with the lightweight stack-based execution approximation, as discussed in Section 7.3.2.

7.4.3 Performance

Takeaway: we use two web performance metrics: Page Load Time (PLT) and SpeedIndex. ShadowBlock speeds up page loads by 5.96% in terms of median PTL and 6.37% in terms of median SpeedIndex, on Alexa top-1000 websites.

Page Load Time (PLT). PLT has been the de-facto standard metric for measuring web performance. PLT can be computed by timing the difference between certain browser events using the Navigation Timing API [118]. In order to minimize variations introduced by the initial network setup (e.g., establishing TCP connection with server), we measure the time between `responseStart` [121] and `loadEventStart` [120] events.

SpeedIndex. PLT does not capture a real user's visual perception of webpage rendering process. For example, two pages A and B can have exactly the same PLTs, but page A can have 95% of its visual content rendered by a certain time point while page B has only rendered 30%. From the user perception perspective, page A outperforms page B but they are equally good in terms of PLT. To address this issue, SpeedIndex [123] was proposed to capture the visual progress of *above-the-fold* content, i.e., content in the viewport without scrolling. Unlike PLT, SpeedIndex measures how visually complete a webpage looks at different points during its loading process. Specifically, the page loading process is recorded as a video and each frame is compared to the final frame, for measuring completeness. SpeedIndex is computed using the following formula:

$$SpeedIndex = \int_{t_{begin}}^{t_{end}} 1 - \frac{VisualCompleteness}{100}$$

, where t_{begin} and t_{end} represent the time points of the start (i.e. `responseStart` event in our case) and end (i.e. `loadEventStart` event in our case) of video recording, respectively. $VisualCompleteness$ measures the difference of the color histogram for each frame in the video versus the histogram at frame t_{begin}, and compares it to the baseline (difference of histogram at t_{begin} and t_{end}) to determine how "complete" that video frame is.

We emulate DSL network condition by throttling Chromium [119] to 4 Mbps downlink bandwidth and 5ms RTT latency for all responses to best mitigate measurement volatility across different browser instances. [3] For each site, we first load the webpage to generate its resource cache, then we re-load the webpage 10 times and average the measured PLT and SpeedIndex for each page load. Note that our warm-up strategy ensures most of the static non-ad resources are cached, while ad resources dynamically generated by JavaScript execution are not. This is intended, because we want to minimize the variability introduced by irrelevant factors such as processing non-ad network traffic.

We compute relative ratio of PLT/SpeedIndex across two different configuration pairs (A) ShadowBlock and vanilla Chromium; and (B) ShadowBlock and Adblock Plus (EasyList + Anti-blocking lists), which are denoted as $Conf_A$ and $Conf_B$, respectively.

$$\frac{PLT_{Group_A} - PLT_{Group_B}}{PLT_{Group_B}}$$

$$\frac{SpeedIndex_{Group_A} - SpeedIndex_{Group_B}}{SpeedIndex_{Group_B}}$$

Overall, both PLT and SpeedIndex show that ShadowBlock speeds up page load time as compared to not only Adblock Plus but also vanilla Chromium. In comparison to Adblock Plus, ShadowBlock speeds up page loads by 5.96% in terms of median PTL and 6.37% in terms of median SpeedIndex. In comparison to vanilla Chromium, ShadowBlock speeds up page loads by 1.03% in terms of

[3]We also run another configuration with 750 Kbps downlink bandwidth and 100ms RTT latency to emulate a regular 3G condition [119] and observe similar median trends for both PLT (DSL -5.96% vs 3G +0.30%) and SpeedIndex (DSL -6.37% vs 3G -7.07%) with respect to Adblock Plus.

median PTL and 5.22% in terms of median SpeedIndex.

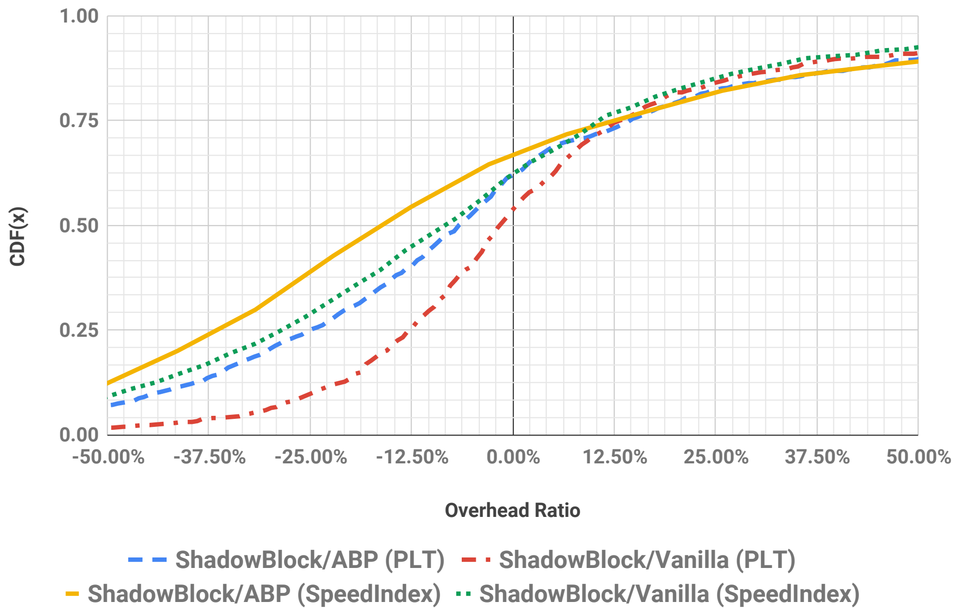


Figure 7.7. CDF for performance metrics

The distributions of PLT and SpeedIndex in Figure 7.7, also confirm this trend. We surmise that ShadowBlock's speed up with respect to Adblock Plus is because ShadowBlock's in-browser modifications, versus Adblock Plus's JavaScript-level implementation, inherently incur less overhead without the necessity of communications between browser core and extension code. For the speed up with respect to vanilla Chromium, it can be explained because ShadowBlock avoids the rendering and painting work for hidden ad elements.

7.5 Discussions and Limitations

Hiding instead of blocking. As discussed in Section 7.3.3, ShadowBlock is designed to visually hide ad elements. Compared to extension-based adblockers that prevent ad resources from loading, ShadowBlock's hiding strategy is inherently limited in two ways. First, ShadowBlock loads the ads and then hides them, thus does not save any network bandwidth. Second, it allows ad resources to load and ad-related scripts to execute, thus exposes users to online tracking. However, we argue that ShadowBlock can complement other tracker blocking approaches that obfuscate and anonymize user-identifiable data [262, 269] which do not require blocking ad requests or stopping script execution.

Completeness of implementation. As discussed in Section 7.3.2, a sufficiently complete yet lightweight taint analysis engine is required to tackle all the FN cases we encounter during ad element identification. However, given the adequate accuracy and practical runtime overhead level, we consider our execution projection technique a sufficient and necessary simplification of taint tracking conceptually.

Adversary from publishers. Modern websites enforce strong isolation among different scripts running in the same document. Violating such policies would normally raise substantial awareness to owners of other scripts or the website itself. This isolation also helps establish the assumption that ad scripts should never interact with non-ad elements in the same page. However, As soon as the publishers become aware of our approach, they might attack ShadowBlock by pro-actively breaking this assumption to cause collateral damage. For example, an ad script can intentionally modify an attribute of a non-ad element without changing its semantics (e.g. by changing the text encoding). In this case, if we blindly follow the taint tracking and mark the element with taint as ads, we might end up hiding benign elements. To address this challenge, we will need an equivalence test on the semantics of the cases with and without tainting.

Alternatively, an adversary may attempt to detect ShadowBlock. Even though we have closed all normal channels (JavaScript APIs) from leaking information about the presence of ShadowBlock. The adversary may still use more extreme means such as side channels. For instance, if we conduct taint analysis, we slow down the JavaScript execution and therefore they can potentially detect ShadowBlock by timing. However, we argue that this will be extremely challenging if not impossible, because there exist many browsers with different versions of JavaScript engines. There are simply too many possibilities if an adversary observes that the execution is slightly slower (it can even be just a slow machine).

7.6 Conclusions

In this chapter, we propose ShadowBlock— a Chromium based stealthy adblocking browser. In addition to blocking ads it hides the traces of adblocking, making it insusceptible to anti-adblocking. Compared to the current state-of-the-art adblocking extensions, that block resources, ShadowBlock allows resources to load and keeps track of them. Later it hides the loaded resources and fakes their states to JavaScript APIs used by anti-adblockers. Through manual evaluation, we find that ShadowBlock (i) achieves 100% success rate in evading visual anti-adblockers; (ii) replicates 98.2% of ads coverage achieved by adblocking extensions; and (iii) causes minor visual breakage on less than 0.6% of the tested websites. In addition, we evaluate ShadowBlock's performance and find that it loads web pages as fast as adblocking extensions in terms of SpeedIndex and Page Load Time, on average. In summary, ShadowBlock constitutes a substantial advancement for building adblockers invisible to anti-adblockers and presents an important advancement in the rapidly escalating adblocking arms race.

CHAPTER 8

THESIS CONTRIBUTIONS & FUTURE RESEARCH DIRECTIONS

This thesis has walked the reader over my journey towards a privacy-preserving web. I first demonstrated that the web continues to suffer from chronic privacy issues. Then I discussed my research that mitigates some of these privacy issues using system instrumentation, program analysis, machine learning, and internet measurement techniques. However, the privacy threats on the web are continuously evolving and presenting new challenges to the privacy-enhancing tools. There is still a lot left to be done, and this thesis serves as a template to address the new and evolving privacy threats. Below I highlighting the contributions that I have made with my research and some of the open research problems in the web privacy space.

8.1 Thesis Contributions

I have made fundamental research contributions on two fronts. First, with system instrumentation, I have defined a framework that can be used to map very granular cross-layer execution of any desktop, mobile, or web application. Mapping out detailed execution allows to investigate and detect different type of anomalies, like tracking and malware, or more generally any behaviors, which could be outside the domain of security and privacy. Second, on the machine learning end, I have applied classical machine learning, that requires hand-crafted features, on complex graph structures and innovated at assembling frameworks that are better at handling weak supervision classification. However, I have just scratched the surface of leveraging information from the complex graph structures, and I believe that a lot more can be accomplished. For example, with Graph Neural Networks (GNNs), there would not be any need to hand-craft features and launching adversarial evasions would be much more difficult My research has been incorporated, both directly and indirectly, by almost all browsers and the privacy-enhancing browser extensions, such as AdBlock Plus. Overall, I believe that this thesis has significantly advanced the state-of-the-art in a thriving sub area of security and privacy, which could be defined as systems privacy.

8.2 Future Research Directions

Reining Stateful Tracking. The whack-a-mole strategy to mitigate stateful tracking has led to an arms-race between malicious third parties and privacy-enhancing tools. Malicious third parties use intrusive cross-site tracking techniques to evade privacy-enhancing systems, who are not equipped to tackle active adversaries. The cat-and-mouse chase demands that new privacy-enhancing tools continuously need to be built to detect and counter intrusive new stateful tracking vectors. Further, existing ML based cross-site tracker detection approaches are susceptible to adversarial evasions, e.g., due to their disproportionate reliance on tracker content (e.g. URLs). To stay a step ahead of the game, there is an immense need to launch evasion attacks on privacy-enhancing tools to check their resilience and build robust privacy-enhancing systems. Lastly, due to adversarial evasions, the usability of the web is disrupted, which requires to build website breakage detection techniques to improve the usability of the web.

Unraveling Stateless Tracking. Cross-site tracking and anti-fraud are the two main use cases of browser fingerprinting [214]. Both of these areas require active exploration. In cross-site tracking use case exploration, new stateless tracking vectors need to be systematically discovered before trackers find ways to exploit JavaScript APIs for browser fingerprinting. In anti-fraud use case exploration, alternate privacy-preserving methods need to be investigated that only require to distinguish fraudsters from real users but not real users users from each other. Further, there is a thin line between the malicious and functional use cases of tracking, which makes the enforcement of countermeasures extremely challenging. Thus, there is a need to investigate techniques that can discern the intent of tracking, allowing countermeasures to be applied without causing website breakage.

CHAPTER 9

CONCLUSION

In this thesis, I built novel privacy-enhancing systems that make the web secure and private. Specifically, in the first half of this thesis, I built four privacy-enhancing systems named AdGraph, WebGraph, Khaleesi, and FP-Inspector that detect and block trackers. AdGraph and WebGraph are both graph-based machine learning approach that detect and mitigate generic and adversarial stateful tracking, respectively. Khaleesi is a sequential-context based machine learning approach that detects and mitigates emerging stateful tracking threats. FP-Inspector is a syntactic-semantic machine learning approach that detects and mitigates stateless tracking. In the second half of this thesis, I proposed an approach named AdWars and built a system, named ShadowBlock that counter retaliation by circumvention services that evade tracker blocking. AdWars is a syntactic approach that counters the circumvention services by detecting and removing them. Whereas ShadowBlock is a stealthy approach that counters the circumvention services by stealthily concealing the actions of tracker blocking.

I addressed two main challenges in tackling online tracking, i.e. scalability and adversarial evasion. Scalability refers to the large number ($\sim$ 1.5 billion) of websites on the internet, where each website is unique from the other and embeds different third parties, and where third parties further include other third party resources on the fly. The dynamism required that the privacy-enhancing tools scale and adapt to changing websites. So to address the scalability issues, I leveraged machine learning techniques to detect *tracking behaviors* instead of *tracker entities.* Second, given the adversarial nature of the problem, malicious third parties often use evasive tactics to circumvent restriction.The adversarial aspect required that the privacy-enhancing tools needed to be robust to adversarial evasion by malicious third parties. So to address the adversarial evasions, I captured execution of trackers across all four layers (HTML, JavaScript, storage, and network) of the web stack with system instrumentation. Capturing the cross-layer context allowed me to trace the provenance of trackers and reveal any evasion attempts. Overall, in the thesis, I showed that we can effectively tackle tracking by modeling the tracker behavior with deep system instrumentation and program analysis techniques and by using machine learning to automatically detect the tracker behavior.

The vision of this thesis can be extended to a variety of other similar and as well as extended research directions. Specifically, mobile and IoT are the closest platforms where this research can be applied because in principle mobile and IoT devices suffer from same security and privacy threats. More broadly, the system instrumentation and the usage of machine learning, for detecting a particular behavior, has applications in solving the performance issues, e.g. detecting bottlenecks, in the web, mobile, and IoT applications.

REFERENCES

[1] A Quick Look at P3P. https://blogs.msdn.microsoft.com/ieinternals/2013/09/17/a-quick-look-at-p3p/.

[2] Acceptable ads program. https://adblockplus.org/acceptable-ads.

[3] AdBlock. https://getadblock.com/.

[4] AdBlock, Chrome web store. https://chrome.google.com/webstore/detail/adblock/gighmmpiobklfepjocnamgkkbiglidom?hl=en-US.

[5] Adblock Plus. https://adblockplus.org/.

[6] Adblock Plus, Chrome web store. https://chrome.google.com/webstore/detail/adblock-plus/cfhdojbkjhnklbpkdaibdccddilifddb.

[7] Adblock Plus Github Repo. http://github.com/adblockplus/adblockplus.

[8] Adblock Plus, Mozilla Firefox add-on. https://addons.mozilla.org/en-US/firefox/addon/adblock-plus/.

[9] Adblock rules list parser. https://github.com/shawa/adblockparser.

[10] Adscore privacy policy. https://www.adscore.com/privacy-policy.

[11] Anti-Adblock Killer. https://github.com/reek/anti-adblock-killer.

[12] Anti-Adblock Killer List Forum. https://github.com/reek/anti-adblock-killer/issues.

[13] Apple Declares War on Browser Fingerprinting, the Sneaky Tactic That Tracks You in Incognito Mode. https://gizmodo.com/apple-declares-war-on-browser-fingerprinting-the-sneak-1826549108.

[14] BlockAdBlock. https://github.com/sitexw/BlockAdBlock/blob/master/blockadblock.js.

[15] Blockzilla. https://zpacman.github.io/Blockzilla/.

[16] Brave Browser. https://brave.com/.

[17] Brave Browser Fingerprinting Protection Mode. https://github.com/brave/browser-laptop/wiki/Fingerprinting-Protection-Mode.

[18] Brave's AdGraph Integration. https://github.com/brave/brave-core/tree/page-graph-integration.

[19] Browserify. http://browserify.org/.

[20] Bypassing ad blockers for Google Analytics. https://analytics-bypassing-adblockers.netlify.app/.

[21] C3 Metrics privacy policy. https://c3metrics.com/privacy/.

[22] Canvas Defender. https://multilogin.com/canvas-defender/.

[23] Chromium Blink Rendering Engine (Renderer). https://cs.chromium.org/chromium/src/third_party/blink/renderer/.

[24] Chromium V8 JavaScript Engine. https://v8.dev/.

[25] Cliqz Browser. https://cliqz.com/us/.

[26] Cliqz Content Blocking Library. https://github.com/cliqz-oss/adblocker.

[27] Coalition for Better Ads. https://www.betterads.org/.

[28] Combating Fingerprinting with a Privacy Budget Explainer. https://github.com/bslassey/privacy-budget.

[29] Default on Cookie Restrictions Excerpt. https://mozilla.report/post/projects/cookie_restrictions.kp/.

[30] Disable third-party cookies in Firefox to stop some types of tracking by advertisers. https://support.mozilla.org/en-US/kb/disable-third-party-cookies.

[31] Disconnect. https://disconnect.me/.

[32] Disconnect policy review for adscore. https://github.com/disconnectme/disconnect-tracking-protection/commit/9666265d0a26fbcc65a20c1021517a44a5ade580.

[33] Disconnect policy review for c3metrics. https://github.com/disconnectme/disconnect-tracking-protection/blob/940d5e6da8fbc738a747a30328c397c4f453683a/descriptions.md#policy-review-3.

[34] Disconnect tracking definition. https://disconnect.me/trackerprotection#definition-of-tracking.

[35] DoubleVerify, Product Privacy Notice. https://web.archive.org/web/20191130014642/https://www.doubleverify.com/privacy/.

[36] EasyList. https://easylist.to/.

[37] EasyList Forum. https://forums.lanik.us.

[38] EasyList variants. https://easylist.to/pages/other-supplementary-filter-lists-and-easylist-variants.html.

[39] EasyPrivacy. https://easylist.to/easylist/easylist.txt.

[40] EFF's Open Letter to Facebook. https://www.eff.org/files/filenode/social_networks/openlettertofacebook.pdf.

[41] Fanboy Annoyances List. https://www.fanboy.co.nz/.

[42] Fanboy's Enhanced Tracking List. https://fanboy.co.nz/.

[43] Filter lists syntax. https://adblockplus.org/en/filter-cheatsheet.

[44] Fingerprinting Defenses in The Tor Browser. https://www.torproject.org/projects/torbrowser/design/#fingerprinting-defenses.

[45] Fingerprintjs2 fingerprinting script. https://fingerprintjs.com/.

[46] Firebug. http://getfirebug.com/.

[47] Firefox Fingerprinting Blocking Breakage Bugs. https://bugzilla.mozilla.org/show_bug.cgi?id=1527013.

[48] Firefox Storage Access Policy. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Storage_access_policy.

[49] Firm uses typing cadence to finger unauthorized users. https://arstechnica.com/tech-policy/2010/02/firm-uses-typing-cadence-to-finger-unauthorized-users/.

[50] Full Third-Party Cookie Blocking and More. https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/.

[51] Ghostery. https://www.ghostery.com/.

[52] Ghostery, Chrome web store. https://chrome.google.com/webstore/detail/ghostery/mlomiejdfkolichcflejclcbmpeaniij?hl=en-US.

[53] Ghostery, Mozilla Firefox add-on. https://addons.mozilla.org/en-US/firefox/addon/ghostery/.

[54] Google Analytics. https://developers.google.com/analytics/devguides/collection/analyticsjs/events.

[55] Google Chrome AdTracker. https://cs.chromium.org/chromium/src/third_party/blink/renderer/core/frame/ad_tracker.h?rcl=fabe78ea42052335674f6cc9c809dd610a8eea29&l=32.

[56] HAR File. https://en.wikipedia.org/wiki/.har.

[57] How to block fingerprinting with Firefox. https://blog.mozilla.org/firefox/how-to-block-fingerprinting-with-firefox/.

[58] IAB Standard Ad Unit Portfolio. https://www.iab.com/wp-content/uploads/2017/08/IABNewAdPortfolio_FINAL_2017.pdf.

[59] Improving Privacy Without Breaking The Web. https://blog.mozilla.org/data/2018/01/26/improving-privacy-without-breaking-the-web/.

[60] Incorrectly Removed Content by Filer Lists. https://forums.lanik.us/viewforum.php?f=64.

[61] Instart Logic AppShield Ad Integrity. https://www.instartlogic.com/products/advertising-marketing-recovery.

[62] Integral Ad Science, Privacy Policy. https://web.archive.org/web/20191130014644/https://integralads.com/privacy-policy/.

[63] Interactive Advertising Bureau. http://www.iab.com/.

[64] Iovation Fraud Protection. https://web.archive.org/web/20191130164107/https://www.iovation.com/fraudforce-fraud-detection-prevention.

[65] JSMin. http://www.crockford.com/javascript/jsmin.html.

[66] McAfee's URL categorization service. https://www.trustedsource.org/.

[67] MDN Web APIs. https://developer.mozilla.org/en-US/docs/Web/API.

[68] MediaMath (MathTag) fingerprinting script. https://www.mediamath.com/.

[69] Mozilla Firefox. https://www.mozilla.org/en-US/firefox/.

[70] Mozilla postpones default blocking of third-party cookies in Firefox. https://www.computerworld.com/article/2497782/mozilla-postpones-default-blocking-of-third-party-cookies-in-firefox.html.

[71] NetExport. https://getfirebug.com/wiki/index.php/Firebug_Extensions.

[72] NoTrack Blocklist. https://github.com/quidsup/notrack.

[73] P3P: The Platform for Privacy Preferences. https://www.w3.org/P3P/.

[74] PageFair, 2017 Global Adblock Report. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf.

[75] Peter Lowe's list. http://pgl.yoyo.org/adservers/.

[76] Privacy Badger. https://www.eff.org/privacybadger.

[77] Privacy Badger, Chrome web store. https://chrome.google.com/webstore/detail/privacy-badger/pkehgijcmpdhfbdbbnkijodmdjhbjlgp?hl=en-US.

[78] Privacy Badger, Mozilla Firefox add-on. https://addons.mozilla.org/en-US/firefox/addon/privacy-badger17/.

[79] Protecting Against HSTS Abuse. https://webkit.org/blog/8146/protecting-against-hsts-abuse/.

[80] Putting Mobile Ad Blockers to the Test. https://www.nytimes.com/2015/10/01/technology/personaltech/ad-blockers-mobile-iphone-browsers.html.

[81] requestAnimationFrame API. https://developer.mozilla.org/en-US/docs/Web/API/window/requestAnimationFrame.

[82] RequireJS. https://requirejs.org/.

[83] Same-origin security model - Resource Timing APIs. https://w3c.github.io/perf-security-privacy/#same-origin-security-model.

[84] Selenium. http://docs.seleniumhq.org/.

[85] Sentinel - The artificial intelligence ad detector. https://adblock.ai/.

[86] Setting First-Party Cookies by Redirection. https://patents.google.com/patent/US20150052217A1.

[87] Squid blacklist. http://www.squidblacklist.org/.

[88] The Tapad Graph. https://www.tapad.com/the-tapad-graph.

[89] Tor browser bug - reduced time precison to mitimate fingerprinting. https://trac.torproject.org/projects/tor/ticket/1517.

[90] Tor Browser Fingerprinting Bugs. https://trac.torproject.org/projects/tor/query?keywords=~tbb-fingerprinting.

[91] Tracking Prevention in WebKit. https://webkit.org/tracking-prevention/.

[92] treeinterpreter. https://pypi.org/project/treeinterpreter/.

[93] Truth In Advertising, Federal Trade Commission. https://www.ftc.gov/news-events/media-resources/truth-advertising/.

[94] uBlock Origin. https://github.com/gorhill/uBlock.

[95] W3C Fingerprinting Guidance. https://w3c.github.io/fingerprinting-guidance.

[96] W3C. Privacy Interest Group Charter. https://www.w3.org/2011/07/privacy-ig-charter.

[97] Warning removal list. https://easylist-downloads.adblockplus.org/antiadblockfilters.txt.

[98] Wayback Machine. https://archive.org/web/.

[99] Wayback Machine API. https://archive.org/help/wayback_api.php.

[100] Wayback Machine Archive Details. https://archive.org/about/.

[101] Webpack. https://webpack.js.org/.

[102] webRequest API. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest.

[103] Webshrinker Website Categorization. https://www.webshrinker.com/.

[104] YourAdChoices. http://youradchoices.com/.

[105] PageFair 2015 Adblock Report. https://pagefair.com/blog/2015/ad-blocking-report/, 2015.

[106] PageFair 2016 Mobile Adblocking Report. https://pagefair.com/blog/2016/mobile-adblocking-report/, 2016.

[107] The state of the blocked web 2017 Global Adblock Report. PageFair. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf, 2017.

[108] Uk publishers lose nearly 3bn in revenue annually due to adblocking, 2017.

[109] Abp anti-circumvention filter list. https://github.com/abp-filters/abp-filters-anti-cv, 2018.

[110] Adblock warning removal list. https://easylist-downloads.adblockplus.org/antiadblockfilters.txt, 2018.

[111] Anti-adblock killer: Don't touch my adblocker! https://reek.github.io/anti-adblock-killer/, 2018.

[112] Blink - the chromium projects. https://www.chromium.org/blink, 2018.

[113] Easylist forum. https://forums.lanik.us/, 2018.

[114] Easylist: Overview. https://easylist.to/, 2018.

[115] Issues with yavli advertising. https://easylist.to/2015/08/19/issues-with-yavli-advertising.html, 2018.

[116] libadblockplus: A c++ library offering the core functionality of adblock plus. https://github.com/adblockplus/libadblockplus, 2018.

[117] Native advertising: A guide for businesses. https://www.ftc.gov/tips-advice/business-center/guidance/native-advertising-guide-businesses, 2018.

[118] Navigation timing api - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Navigation_timing_API, 2018.

[119] Optimize performance under varying network conditions | tools for web developers | google developers. https://developers.google.com/web/tools/chrome-devtools/network-performance/network-conditions, 2018.

[120] Performancetiming.loadeventstart - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/PerformanceTiming/loadEventStart, 2018.

[121] Performancetiming.responsestart - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/PerformanceTiming/responseStart, 2018.

[122] Render-tree construction, layout, and paint. https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction, 2018.

[123] Speed index - webpagetest documentation. https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index, 2018.

[124] Subresourcefilter in chromium source code. https://cs.chromium.org/chromium/src/components/subresource_filter/, 2018.

[125] V8 javascript engine. https://v8.dev/, 2018.

[126] V8stacktraceimpl in chromium source code. https://cs.chromium.org/chromium/src/v8/src/inspector/v8-stack-trace-impl.h, 2018.

[127] visibility - css: Cascading style sheets | mdn. https://developer.mozilla.org/en-US/docs/Web/CSS/visibility, 2018.

[128] Web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API, 2018.

[129] Yavli filters issues - easylist forum. https://forums.lanik.us/viewtopic.php?f=64&t=36091, 2018.

[130] Blob - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Blob, 2019.

[131] Media source extensions api - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Media_Source_Extensions_API, 2019.

[132] preload.js in adblock plus extension that injects css selectors into web pages. https://github.com/adblockplus/adblockpluschrome/blob/c742bcc37b459c03bd564aea941ef6f05834e7fd/include.preload.js#L259, 2019.

[133] Salesforce KUID. https://konsole.zendesk.com/hc/en-us/articles/115013802488-KUID, 2020.

[134] Understanding Calls to the Demdex Domain. https://experienceleague.adobe.com/docs/audience-manager/user-guide/reference/demdex-calls.html?lang=en#reference, 2021.

[135] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *CCS*, 2014.

[136] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[137] G. Acar, M. Juarez, N. Nikiforakis, C. Diaz, S. Gürses, F. Piessens, and B. Pre-

neel. FPDetective: dusting the web for fingerprinters. In *Proceedings of CCS*. ACM, 2013.

[138] N. M. Al-Fannah, W. Li, and C. J. Mitchell. Beyond Cookie Monster Amnesia: Real World Persistent Online Tracking. In *Information Security Conference*, 2018.

[139] F. Alaca and P. van Oorschot. Device Fingerprinting for Augmenting Web Authentication: Classification and Analysis of Methods. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*, 2016.

[140] M. Alrizah, S. Zhu, X. Xing, and G. Wang. Errors, Misunderstandings, and Attacks: Analyzing the Crowdsourcing Process of Ad-blocking Systems. In *IMC*, 2019.

[141] G. Anthes. Data Brokers Are Watching You. *Communications of the ACM*, 2015.

[142] D. Antoniades, I. Polakis, G. Kontaxis, E. Athanasopoulos, S. Ioannidis, E. P. Markatos, and T. Karagiannis. We.b: The web of short urls. In *World Wide Web (WWW) Conference*, 2011.

[143] M. D. Ayenson, D. J. Wambach, A. Soltani, N. Good, and C. J. Hoofnagle. Flash cookies and privacy ii: Now with html5 and etag respawning. *World Wide Web Internet and Web Information Systems*, 2011.

[144] B. A. Azad, O. Starov, P. Laperdrix, and N. Nikiforakis. Web runner 2049: Evaluating third-party anti-bot services. In *17th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2020.

[145] M. A. Bashir, S. Arshad, W. Robertson, and C. Wilson. Tracing information flows between ad exchanges using retargeted ads. In *25th USENIX Security Symposium*, 2016.

[146] M. A. Bashir and C. Wilson. Diffusion of user tracking data in the online adver-

tising ecosystem. *Proceedings on Privacy Enhancing Technologies*, 2018(4):85–103, 2018.

[147] J. Bau, J. Mayer, H. Paskov, and J. C. Mitchel. A Promising Direction for Web Tracking Countermeasures. In *W2SP*, 2013.

[148] W. Benchaita, S. Ghamri-Doudane, and S. Tixeuil. On the optimization of request routing for content delivery. In *SIGCOMM*, 2015.

[149] S. Bhagavatula, C. Dunn, C. Kanich, M. Gupta, and B. Ziebart. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *ACM Workshop on Artificial Intelligence and Security*, 2014.

[150] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. In *Journal of Statistical Mechanics: Theory and Experiment*, 2008.

[151] J. Bloomberg. Ad Blocking Battle Drives Disruptive Innovation. `https://www.forbes.com/sites/jasonbloomberg/2017/02/18/ad-blocking-battle-drives-disruptive-innovation`, 2017.

[152] H. Bojinov, Y. Michalevsky, G. Nakibly, and D. Boneh. Mobile Device Identification via Sensor Fingerprinting. In *arXiv*, 2014.

[153] A. Bosworth. A New Way to Control the Ads You See on Facebook, and an Update on Ad Blocking. `https://newsroom.fb.com/news/2016/08/a-new-way-to-control-the-ads-you-see-on-facebook-and-an-update-on-ad-blocking/`, 2016.

[154] Brave. A Long List of Ways Brave Goes Beyond Other Browsers to Protect Your Privacy. `https://brave.com/privacy-features/`.

[155] L. Breiman. Random Forests. In *Machine learning*, 2001.

[156] J. Burgess, D. Carlin, P. O'Kane, and S. Sezer. Redirekt: Extracting malicious

redirections from exploit kit traffic. In *2020 IEEE Conference on Communications and Network Security (CNS)*, 2020.

[157] E. Bursztein, A. Malyshev, T. Pietraszek, and K. Thomas. Picasso: Lightweight Device Class Fingerprintingfor Web Clients. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2016.

[158] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2011.

[159] S. Y. Cao and E. Wijmans. (cross-)browser fingerprinting via os and hardware level features. In *Proceedings of the 2017 Network & Distributed System Security Symposium, NDSS*, volume 17, 2017.

[160] L. Chang, H.-C. Hsiao, W. Jeng, T. H.-J. Kim, and W.-H. Lin. Security implications of redirection trail in popular websites worldwide. In *World Wide Web (WWW) Conference*, 2017.

[161] H. Chen, H. Zhang, D. Boning, and C.-J. Hsieh. Robust decision trees against adversarial examples. In *International Conference on Machine Learning*, 2019.

[162] H. Chen, H. Zhang, S. Si, Y. Li, D. Boning, and C.-J. Hsieh. Robustness verification of tree-based models. In *Advances in Neural Information Processing Systems*, 2019.

[163] Q. Chen and A. Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1687–1700, 2018.

[164] Q. Chen, P. Snyder, B. Livshits, and A. Kapravelos. Detecting filter list evasion with event-loop-turn granularity javascript signatures. In *Proceedings of the IEEE Symposium on Security and Privacy (May 2021)*, 2021.

[165] S. Chhabra, A. Aggarwal, F. Benevenuto, and P. Kumaraguru. Phi.sh/$ocial:

The phishing landscape through short urls. In *Proceedings of the 8th Annual Collaboration, Electronic Messaging, Anti-Abuse and Spam Conference*, 2011.

[166] G. Chrome. The Privacy Sandbox. `https://www.chromium.org/Home/chromium-privacy/privacy-sandbox`.

[167] A. Chudnov and D. A. Naumann. Inlined information flow monitoring for javascript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 629–643, 2015.

[168] C. Cimpanu. 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.

[169] CNET. Brave's AI blocks ads better than today's browser plug-ins, company says. `https://www.cnet.com/news/braves-ai-blocks-ads-better-than-todays-browser-plug-ins-company-says/`.

[170] R. Cointepas. CNAME Cloaking, the dangerous disguise of third-party trackers. https://medium.com/nextdns/cname-cloaking-the-dangerous-disguise-of-third-party-trackers-195205dc522a, 2010.

[171] L. F. Cranor, S. Egelman, S. Sheng, A. M. McDonald, and A. Chowdhury. P3P Deployment on Websites. In *Electronic Commerce Research and Applications*, 2008.

[172] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *USENIX Security Symposium*, 2011.

[173] H. Dang, Y. Huang, and E. Chang. Evading Classifiers by Morphing in the Dark. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[174] H. Dao, J. Mazel, and K. Fukuda. Characterizing CNAME Cloaking-Based

Tracking on the Web. In *Network Traffic Measurement and Analysis Conference*, 2020.

[175] A. Das, G. Acar, N. Borisov, and A. Pradeep. The Web's Sixth Sense: A study of scripts accessing smartphone sensors. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1515–1532, 2018.

[176] A. Datta, J. Lu, and M. C. Tschantz. The effectiveness of privacy enhancing technologies against fingerprinting. *arXiv preprint arXiv:1812.03920*, 2018.

[177] W. Davis. BlueCava Touts Device Fingerprinting. `https://web.archive.org/web/20150928090154/https://www.mediapost.com/publications/article/166916/bluecava-touts-device-fingerprinting.html`, 2012.

[178] S. Dey, N. Roy, W. Xu, R. R. Choudhury, and SrihariNelakuditi. AccelPrint: Imperfections of accelerometers make smartphones trackable. In *Proceeding of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, 2014.

[179] Y. Dimova, G. Acar, L. Olejnik, W. Joosen, and T. Van Goethem. The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion. *Proceedings on Privacy Enhancing Technologies*, 2021(1):109–126, 2021.

[180] P. Dolanjski. Mozilla Firefox The Path to Enhanced Tracking Protection. `https://blog.mozilla.org/futurereleases/2018/10/23/the-path-to-enhanced-tracking-protection`.

[181] P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*. Springer, 2010.

[182] A. Edelstein. Protections Against Fingerprinting and Cryptocurrency Mining Available in Firefox Nightly and Beta. `https://blog.mozilla.org/futurereleases/2019/04/09/protections-against-fingerprinting-and-cryptocurrency-mining-available-in-firefox-nightly-and-beta/`, 2019.

[183] S. Englehardt. The Hidden Perils of Cookie Syncing. `https://freedom-to-tinker.com/2014/08/07/the-hidden-perils-of-cookie-syncing/`, 2014.

[184] S. Englehardt, J. Han, and A. Narayanan. I never signed up for this! privacy implications of email tracking. *Proceedings on Privacy Enhancing Technologies*, 2018(1):109–126, 2018.

[185] S. Englehardt, J. Han, and A. Narayanan. I never signed up for this! Privacy implications of email tracking. In *PETS*, 2018.

[186] S. Englehardt and A. Narayanan. Online Tracking: A 1-million-site Measurement and Analysis. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[187] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten. Cookies That Give You Away:The Surveillance Implications of Web Tracking . In *World Wide Web (WWW) Conference*, 2015.

[188] A. Fass, M. Backes, and B. Stock. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[189] A. Fass, M. Backes, and B. Stock. Jstap: A static pre-filter for malicious javascript detection. In *Proceedings of the 32nd Annual Conference on Computer Security Applications (ACSAC)*, 2019.

[190] D. Fifield and S. Egelman. Fingerprinting web users through font metrics. In *Financial Cryptography and Data Security*, pages 107–124. Springer, 2015.

[191] Forbes. No More Third Party Cookies, No Problemo. `https://www.forbes.com/sites/augustinefou/2020/08/31/no-more-third-party-cookies---good-or-bad-news/`.

[192] I. Fouad, N. Bielova, A. Legout, and N. Sarafijanovic-Djukic. Missed by Fil-

ter Lists: Detecting Unknown Third-Party Trackers with Invisible Pixels. In *Proceedings on Privacy Enhancing Technologies (PETS)*, 2020.

[193] G. A. Fowler. Think you're anonymous online? A third of popular websites are 'fingerprinting' you. `https://www.washingtonpost.com/technology/2019/10/31/think-youre-anonymous-online-third-popular-websites-are-fingerprinting-you/`, 2019.

[194] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. In *Journal of Computer and System Sciences*, 1997.

[195] B. Fulgham. Protecting Against HSTS Abuse. `https://webkit.org/blog/8146/protecting-against-hsts-abuse/`, 2018.

[196] H. Gao, J. Hu, C. Wilson, Z. Li, Y. Chen, and B. Y. Zhao. Detecting and characterizing social spam campaigns. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, 2010.

[197] K. Garimella, O. Kostakis, and M. Mathioudakis. Ad-blocking: A Study on Performance, Privacy and Counter-measures. In *WebSci*, 2017.

[198] A. Gervais, A. Filios, V. Lenders, and S. Capkun. Quantifying Web Adblocker Privacy. In *ESORICS*, 2017.

[199] A. Gomez-Boix, P. Laperdrix, and B. Baudry. Hiding in the Crowd: an Analysis of the Effectiveness of Browser Fingerprinting at Large Scale. In *The Web Conference*, 2018.

[200] I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[201] D. Goßen, I. H. Jonker, and I. E. Poll. Design and implementation of a stealthy openwpm web scraper. 2020.

[202] M. Graham. robots.txt meant for search engines don't work well for web

archives. https://blog.archive.org/2017/04/17/robots-txt-meant-for-search-engines-dont-work-well-for-web-archives/, 2017.

[203] D. Gugelmann, M. Happe, B. Ager, and V. Lenders. An Automated Approach for Complementing Ad Blockers' Blacklists. In *Privacy Enhancing Technologies Symposium (PETS)*, 2015.

[204] N. Hansen, L. D. Carli, and D. Davidson. Assessing Adaptive Attacks Against Trained JavaScript Classifiers. In *16th EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2020.

[205] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. Jsflow: Tracking information flow in javascript and its apis. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1663–1671, 2014.

[206] G. Hong, Z. Yang, S. Yang, L. Zhang, Y. Nan, Z. Zhang, M. Yang, Y. Zhang, Z. Qian, and H. Duan. How you get shot in the back: A systematical study about cryptojacking in the real world. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1701–1713. ACM, 2018.

[207] S. Hou, Y. Fan, Y. Zhang, Y. Ye, J. Lei, W. Wan, J. Wang, Q. Xiong, and F. Shao. αcyber: Enhancing robustness of android malware detection system against adversarial attacks on heterogeneous graph based model. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 609–618, 2019.

[208] S. Ihm and V. S. Pai. Towards Understanding Modern Web Traffic. In *ACM Internet Measurement Conference (IMC)*, 2011.

[209] M. Ikram, H. J. Asghar, M. A. Kaafar, A. Mahanti, and B. Krishnamurthy. Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning . In *Privacy Enhancing Technologies Symposium (PETS)*, 2017.

[210] M. Ikram, R. Masood, G. Tyson, M. A. Kaafar, N. Loizon, and R. Ensafi.

The Chain of Implicit Trust: An Analysis of the Web Third-party Resources Loading. In *The Web Conference (WWW)*, 2019.

[211] U. Iqbal. Bug reports to filter lists. `https://github.com/uiowa-irl/FP-Inspector\#bug-reports-to-filter-lists`, 2020.

[212] U. Iqbal. Bug reports to mozilla firefox. `https://bugzilla.mozilla.org/buglist.cgi?email1=umar-iqbal&classification=Components&resolution=---&query_format=advanced&emailreporter1=1&emailtype1=substring`, 2020.

[213] U. Iqbal. JavaScript API instrumentation in WebKit. `https://bugs.webkit.org/show_bug.cgi?id=215208`, `https://bugs.webkit.org/show_bug.cgi?id=213319`, 2020.

[214] U. Iqbal, S. Englehardt, and Z. Shafiq. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In *Proceedings of the IEEE Symposium on Security & Privacy*, 2021.

[215] U. Iqbal, Z. Shafiq, and Z. Qian. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *Proceedings of the 2017 Internet Measurement Conference (IMC)*, 2017.

[216] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *Proceedings of the IEEE Symposium on Security & Privacy*, 2020.

[217] U. Iqbal, C. Wolfe, C. Nguyen, S. Englehardt, and Z. Shafiq. Khaleesi: Breaking the advertising & tracking redirect chains. In *Under review.*

[218] B. Johansen. Tracking visitors with adblockers. `https://www.bjornjohansen.com/tracking-visitors-with-adblockers`.

[219] John Ross Quinlan. Induction of decision trees. 1986.

[220] John Wilander. Bounce Tracking Protection. https://github.com/privacycg/proposals/issues/6.

[221] John Wilander. Clear-Site-Data For Cross-Site Tracking. https://github.com/privacycg/storage-partitioning/issues/11.

[222] John Wilander. Intelligent Tracking Prevention 2.0. https://webkit.org/blog/8311/intelligent-tracking-prevention-2-0/.

[223] John Wilander. Intelligent Tracking Prevention 2.2. https://webkit.org/blog/8828/intelligent-tracking-prevention-2-2/.

[224] John Wilander. Safari ITP Classifier. https://bugs.webkit.org/show_bug.cgi?id=168347.

[225] A. J. Kaizer and M. Gupta. Towards Automatic identification of JavaScript-oriented Machine-Based Tracking. In *IWSPA*, 2016.

[226] A. Kantchelian, J. D. Tygar, and A. Joseph. Evasion and hardening of tree ensemble classifiers. In *International Conference on Machine Learning*, 2016.

[227] A. H. Kargaran, M. S. Akhondzadeh, M. R. Heidarpour, M. H. Manshaei, K. Salamatian, and M. N. Sattary. On detecting hidden third-party web trackers with a wide dependency chain graph: A representation learning approach. *arXiv preprint arXiv:2004.14826*, 2020.

[228] R. Karim, F. Tip, A. Sochurkova, and K. Sen. Platform-independent dynamic taint analysis for javascript. *IEEE Transactions on Software Engineering*, 2018.

[229] G. Kontaxis, M. Polychronakis, A. D. Keromytis, and E. P. Markatos. Privacy-Preserving Social Plugins. In *Usenix Security Symposium*, 2012.

[230] M. Koop, E. Tews, and S. Katzenbeisser. In-Depth Evaluation of Redirect Tracking and Link Usage. In *Proceedings on Privacy Enhancing Technologies (PETS)*, 2020.

[231] B. Krishnamurthy and C. E. Wills. Privacy Diffusion on the Web: A Longitudinal Perspective. In *World Wide Web (WWW) Conference*, 2009.

[232] P. Laperdrix. Browser Fingerprinting: An Introduction and the Challenges Ahead. `https://blog.torproject.org/browser-fingerprinting-introduction-and-challenges-ahead`, 2019.

[233] P. Laperdrix, G. Avoine, B. Baudry, and N. Nikiforakis. Morellian Analysis for Browsers: Making Web Authentication Stronger with Canvas Fingerprinting. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2019.

[234] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine. Browser fingerprinting: A survey. *arXiv preprint arXiv:1905.01051*, 2019.

[235] P. Laperdrix, W. Rudametkin, and B. Baudry. Beauty and the Beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy*, 2016.

[236] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. 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.

[237] H. Le, F. Fallace, and P. Barlet-Ros. Towards accurate detection of obfuscated web tracking. In *IEEE International Workshop on Measurement and Networking (M&N)*, 2017.

[238] S. Le Page, G. Jourdan, G. V. Bochmann, J. Flood, and I. Onut. Using url shorteners to compare phishing and malware attacks. In *2018 APWG Symposium on Electronic Crime Research (eCrime)*, 2018.

[239] S. Lekies, B. Stock, and M. Johns. 25 million flows later: Large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, pages 1193–1204, 2013.

[240] T. Lendacky. The Platform for Privacy Preferences (P3P). https://www-archive.mozilla.org/projects/p3p/.

[241] A. Lerner, T. Kohno, and F. Roesner. Rewriting History: Changing the Archived Web from the Present. In *ACM Conference on Computer and Communications Security (CCS)*, 2017.

[242] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner. Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. In *USENIX Security Symposium*, 2016.

[243] B. Li, P. Vadrevu, K. H. Lee, and R. Perdisci. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *25th Annual Network and Distributed System Security Symposium*, 2018.

[244] X. Li, L. Wang, and E. Sung. AdaBoost with SVM-based component classifiers. In *Engineering Applications of Artificial Intelligence*, 2007.

[245] I. Lunden. Relx acquires ThreatMetrix for 817M to ramp up in risk-based authentication. `https://techcrunch.com/2018/01/29/relx-threatmetrix-risk-authentication-lexisnexis/`, 2018.

[246] F. Maggi, A. Frossi, S. Zanero, G. Stringhini, B. Stone-Gross, C. Kruegel, and G. Vigna. Two years of short urls internet measurement: Security threats and countermeasures. In *Proceedings of the 22nd International Conference on World Wide Web*, 2013.

[247] M. Malloy, M. McNamara, A. Cahn, and P. Barford. Ad Blockers: Global Prevalence and Impact. In *ACM Internet Measurement Conference (IMC)*, 2016.

[248] J. Marshall. The Rise of the Anti-Ad Blockers. `https://www.wsj.com/articles/the-rise-of-the-anti-ad-blockers-1465805039`, 2016.

[249] V. Mavroudis, S. Hao, Y. Fratantonio, F. Maggi, C. Kruegel, and G. Vigna. On

the privacy and security of the ultrasound ecosystem. *Proceedings on Privacy Enhancing Technologies*, 2017(2):95–112, 2017.

[250] J. R. Mayer. “any person... a pamphleteer”: Internet anonymity in the age of web 2.0. 2009.

[251] J. R. Mayer and J. C. Mitchell. Third-Party Web Tracking: Policy and Technology. In *IEEE Symposium on Security and Privacy*, 2012.

[252] MDN. Storage access policy: Block cookies from trackers. `https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Storage_access_policy`.

[253] MDN. Redirect Tracking Protection. `https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Redirect_Tracking_Protection`, 2020.

[254] H. Mekky, R. Zhi-Li, Zhang, S. Saha, and A. Nucci. Detecting malicious http redirections using trees of user browsing activity. In *IEEE International Conference on Computer Communications (INFOCOM)*, 2014.

[255] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl. Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools. In *IEEE European Symposium on Security and Privacy*, 2017.

[256] Microsoft Edge Team. Introducing tracking prevention, now available in Microsoft Edge preview builds. `https://blogs.windows.com/msedgedev/2019/06/27/tracking-prevention-microsoft-edge-preview/`.

[257] K. Mowery and H. Shacham. Pixel perfect: Fingerprinting canvas in html5. *Proceedings of W2SP*, 2012.

[258] M. H. Mughees, Z. Qian, and Z. Shafiq. Detecting Anti Ad-blockers in the Wild . In *Privacy Enhancing Technologies Symposium (PETS)*, 2017.

[259] NetIQ. Device Fingerprinting for Low Friction Authentication.

`https://www.microfocus.com/media/white-paper/device_fingerprinting_for_low_friction_authentication_wp.pdf`.

[260] Nick Nikiforakis and Wouter Joosen and Benjamin Livshits. PriVaricator: Deceiving Fingerprinters with Little White Lies. In *WWW*, 2015.

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

[262] N. Nikiforakis, W. Joosen, and B. Livshits. Privaricator: Deceiving fingerprinters with little white lies. In *Proceedings of the 24th International Conference on World Wide Web*, pages 820–830. International World Wide Web Conferences Steering Committee, 2015.

[263] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *Security and Privacy (S&P)*. IEEE, 2013.

[264] R. Nithyanand, S. Khattak, M. Javed, N. Vallina-Rodriguez, M. Falahrastegar, J. E. Powles, E. D. Cristofaro, H. Haddadi, and S. J. Murdoch. Adblocking and Counter-Blocking: A Slice of the Arms Race. In *USENIX Workshop on Free and Open Communications on the Internet*, 2016.

[265] M. Nottingham. Unsanctioned Web Tracking. `https://www.w3.org/2001/tag/doc/unsanctioned-tracking/`, 2015.

[266] L. Olejnik, G. Acar, C. Castelluccia, and C. Diaz. The leaking battery: A privacy analysis of the HTML5 Battery Status API. In *Cryptology ePrint Archive: Report 2015/616*, 2015.

[267] L. Olejnik, S. Englehardt, and A. Narayanan. Battery Status Not Included:Assessing Privacy in Web Standards. In *International Workshop on Privacy Engineering*, 2017.

[268] L. Olejnik, M.-D. Tran, and C. Castelluccia. Selling off privacy at auction. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

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

[270] P. Papadopoulos, N. Kourtellis, and E. P. Markatos. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *Proceedings of the World Wide Web (WWW) Conference*, 2019.

[271] P. Papadopoulos, N. Kourtellis, and E. P. Markatos. Cookie Synchronization: Everything You Always Wanted to Know But Were Afraid to Ask. In *The Web Conference*, 2019.

[272] P. Papadopoulos, P. Snyder, D. Athanasakis, and B. Livshits. Keeping out the Masses: Understanding the Popularity and Implications of Internet Paywalls. In *The Web Conference*, 2020.

[273] D. Plohmann, K. Yakdan, M. Klatt, J. Bader, and E. Gerhards-Padilla. A Comprehensive Measurement Study of Domain Generating Malware. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[274] E. Pujol, O. Hohlfeld, and A. Feldmann. Annoyed Users: Ads and Ad-Block Usage in the Wild. In *ACM Internet Measurement Conference (IMC)*, 2015.

[275] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[276] M. Z. Rafique, T. V. Goethem, W. Joosen, C. Huygens, and N. Nikiforakis. It's Free for a Reason: Exploring the Ecosystem of Free Live Streaming Services. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[277] S. Ramaswamy. Building a better web for everyone. `https://www.blog.google/topics/journalism-news/building-better-web-everyone/`, 2017.

[278] T. Ren, A. Wittman, L. D. Carli, and D. Davidson. An analysis of first-party cookie exfiltration due to cname redirections. In *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb)*, 2021.

[279] F. Roesner, T. Kohno, and D. Wetherall. Detecting and Defending Against Third-Party Tracking on the Web . In *USENIX Symposium on Networked Systems Design and Implementation (NDSI)*, 2012.

[280] E. Rudd, A. Rozsa, M. Gunther, and T. Boult. A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions. *IEEE Communications Surveys & Tutorials*, 19(2):1145–1172, 2017.

[281] I. Sanchez-Rola, D. Balzarotti, C. Kruegel, G. Vigna, and I. Santos. Dirty Clicks: A Study of the Usability and SecurityImplications of Click-related Behaviors on the Web. In *The Web Conference (WWW)*, 2020.

[282] I. Sanchez-Rola, I. Santos, and D. Balzarotti. Clock Around the Clock: Time-Based Device Fingerprinting. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[283] J. Schuh. Building a more private web. `https://www.blog.google/products/chrome/building-a-more-private-web`, 2019.

[284] M. Schwarz, F. Lackner, and D. Gruss. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In *NDSS*, 2019.

[285] R. Scully. Identity Resolution vs Device Graphs: Clarifying the Differences. `https://amperity.com/blog/identity-resolution-vs-device-graphs-clarifying-differences/`.

[286] Shitong Zhu and Umar Iqbal and Zhongjie Wang and Zhiyun Qian and Zubair Shafiq and Weiteng Chen. ShadowBlock: A Lightweight and Stealthy Adblocking Browser. In *The Web Conference (WWW)*, 2019.

[287] A. Shuba, A. Markopoulou, and Z. Shafiq. NoMoAds: Effective and Efficient

Cross-App Mobile Ad-Blocking. In *Privacy Enhancing Technologies Symposium (PETS)*, 2018.

[288] S. Siby, U. Iqbal, S. Englehardt, Z. Shafiq, and C. Troncoso. Webgraph: Capturing advertising and tracking information flows for ro-bust blocking. In *To appear in the USENIX Security Symposium*, 2022.

[289] S. Sivakorn, J. Polakis, and A. D. Keromytis. I'm not a human: Breaking the Google reCAPTCHA. In *Black Hat Asia*, 2016.

[290] A. Sjösten, P. Snyder, A. Pastor, P. Papadopoulos, and B. Livshits. Filter List Generation for Underserved Regions. In *WWW*, 2020.

[291] P. Skolka, C.-A. Staicu, and M. Pradel. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *World Wide Web (WWW) Conference*, 2019.

[292] G. Sloane. Ad Blocker's Successful Assault on Facebook Enters Its Second Month. `http://adage.com/article/digital/blockrace-adblock/311103/`, 2017.

[293] P. Snyder, L. Ansari, C. Taylor, and C. Kanich. Browser feature usage on the modern web. In *Proceedings of the 2016 Internet Measurement Conference*, pages 97–110. ACM, 2016.

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

[295] P. Snyder, A. Vastel, and B. Livshits. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. In *ACM SIGMETRICS*, 2020.

[296] B. Software. PageGraph. `https://github.com/brave/brave-browser/wiki/PageGraph`.

[297] O. Starov and N. Nikiforakis. Xhound: Quantifying the fingerprintability of browser extensions. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 941–956. IEEE, 2017.

[298] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 655–670, San Diego, CA, 2014.

[299] G. Storey, D. Reisman, J. Mayer, and A. Narayanan. The Future of Ad Blocking: An Analytical Framework and New Techniques. In *arXiv:1705.08568*, 2017.

[300] F. Tramer, P. Dupre, G. Rusak, G. Pellegrino, and D. Boneh. AdVersarial: Perceptual Ad Blocking meets Adversarial Machine Learning. In *ACM Conference on Computer and Communications Security (CCS)*, 2019.

[301] F. Tramèr, P. Dupré, G. Rusak, G. Pellegrino, and D. Boneh. Adversarial: Perceptual ad blocking meets adversarial machine learning. In *CCS*, 2019.

[302] A. Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106, 2004.

[303] Valentino Rizzo. Machine Learning Approaches for Automatic Detection of Web Fingerprinting. Master's thesis, Politecnico di Torino, Corso di laurea magistrale in Ingegneria Informatica (Computer Engineering), 2018.

[304] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy. Fp-Scanner: The Privacy Implications of Browser Fingerprint Inconsistencies. In *USENIX Security*, 2018.

[305] A. Vastel, P. Laperdrix, W. Rudametkin, and R. Rouvoy. Fp-stalker: Tracking browser fingerprint evolutions. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 728–741. IEEE, 2018.

[306] VentureBeat. Researchers use AI to combat and quantify browser fingerprinting. `https://venturebeat.com/2020/08/17/researchers-use-ai-to-combat-and-quantify-browser-fingerprinting/`.

[307] R. J. Walls, E. D. Kilmer, N. Lageman, and P. D. McDanie. Measuring the Impact and Perception of Acceptable Advertisements. In *ACM Internet Measurement Conference (IMC)*, 2015.

[308] W. Wang, Y. Zheng, X. Xing, Y. Kwon, X. Zhang, and P. Eugster. WebRanz: Web Page Randomization For Better Advertisement Delivery and Web-Bot Prevention. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[309] X. Wang, J. Eaton, C.-J. Hsieh, and F. Wu. Attack graph convolutional networks by adding fake nodes. *arXiv preprint arXiv:1810.10751*, 2018.

[310] J. Wilander. Apple Safari Intelligent Tracking Prevention. `https://webkit.org/blog/8311/intelligent-tracking-prevention-2-0/`, 2018.

[311] J. Wilander. Intelligent Tracking Prevention 2.3. `https://webkit.org/blog/9521/intelligent-tracking-prevention-2-3/`, 2019.

[312] J. Wilander. Bounce Tracking. `https://github.com/privacycg/proposals/issues/6`, 2020.

[313] J. Wilander. CNAME Cloaking and Bounce Tracking Defense. `https://webkit.org/blog/11338/cname-cloaking-and-bounce-tracking-defense/`, 2020.

[314] B. Williams. Ping pong with Facebook. `https://adblockplus.org/blog/ping-pong-with-facebook`, 2018.

[315] M. Wood. Today's Firefox Blocks Third-Party Tracking Cookies and Cryptomining by Default. `https://blog.mozilla.org/blog/2019/09/03/todays-firefox-blocks-third-party-tracking-cookies-and-cryptomining-by-default/`, 2019.

[316] Q. Wu, Q. Liu, Y. Zhang, P. Liu, and G. Wen. A Machine Learning Approach for Detecting Third-Party Trackers on the Web. In *ESORICS*, 2016.

[317] Y. Yang and J. O. Pedersen. A Comparative Study on Feature Selection in Text Categorization. In *International Conference on Machine Learning*, 1997.

[318] Z. Yu, S. Macbeth, K. Modi, and J. M. Pujol. Tracking the Trackers. In *World Wide Web (WWW) Conference*, 2016.

[319] Z. Zaifeng. Who is Stealing My Power III: An Adnetwork Company Case Study, 2018. `http://blog.netlab.360.com/who-is-stealing-my-power-iii-an-adnetwork-company-case-study-en/`.

[320] ZDNet. A quarter of the Alexa Top 10K websites are using browser fingerprinting scripts. `https://www.zdnet.com/article/a-quarter-of-the-alexa-top-10k-websites-are-using-browser-fingerprinting-scripts/`.

[321] S. Zhu, X. Hu, Z. Qian, Z. Shafiq, and H. Yin. Measuring and disrupting anti-adblockers using differential execution analysis. NDSS, 2018.

[322] S. Zhu, Z. Wang, X. Chen, S. Li, U. Iqbal, Z. Qian, K. S. Chan, S. V. Krishnamurthy, and Z. Shafiq. A4: Evading learning-based adblockers. *arXiv preprint arXiv:2001.10999*, 2020.

[323] D. Zügner, A. Akbarnejad, and S. Günnemann. Adversarial attacks on neural networks for graph data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2847–2856, 2018.