Detecting Filter List Evasion With Event-Loop-Turn Granularity JavaScript Signatures

Quan Chen
North Carolina State University
qchen10@ncsu.edu

Peter Snyder
Brave Software
pes@brave.com

Ben Livshits
Brave Software
ben@brave.com

Alexandros Kapravelos
North Carolina State University
akaprav@ncsu.edu

2021 IEEE Symposium on Security and Privacy (SP) | 978-1-7281-8934-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SP40001.2021.00007

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

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

This work makes the following contributions to the problem of measuring and improving content blocking on the web: First, we design and implement a novel system to build per-event-loop-turn signatures of JavaScript behavior through deep instrumentation of the Blink and V8 runtimes. Second, we apply these signatures to measure how much privacy-and-security harming code is missed by current content blockers, by using EasyList and EasyPrivacy as ground truth and finding scripts that have the same privacy and security harming patterns. We build 1,995,444 signatures of privacy-and-security relevant behaviors from 11,212 unique scripts blocked by filter lists, and find 3,589 unique scripts hosting known harmful code, but missed by filter lists, affecting 12.48% of websites measured. Third, we provide a taxonomy of ways scripts avoid detection and quantify the occurrence of each. Finally, we present defenses against these evasions, in the form of filter list additions where possible, and through a proposed, signature based system in other cases.

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

I. Introduction

Previous research has documented the many ways content blocking tools improve privacy, security, performance, and user experience online (e.g., [25], [13], [24], [28]). These tools are the current stage in a long arms race between communities that maintain privacy tools, and online trackers who wish to evade them.

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

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

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

Despite these well known and simple circumventions, the privacy and research community lacks even an understanding of the scale of the problem, let alone useful, practical defenses. Put differently, researchers and activists know they *might* be losing the battle against trackers and online attackers, but lack measurements to determine if this is true, and if so, by how much. Furthermore, the privacy community lacks a way of providing practical (i.e., web-compatible) privacy improvements that are robust regardless of how the attackers choose to deliver their code.

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

Addressing this mismatch requires a solution that is able to

DOI 10.1109/SP40001.2021.00007

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

In this work, we address the above challenges through the design and implementation of a system for building signatures of privacy-and-security harming functionality implemented in JavaScript. Our system extracts script behaviors that occur in one JavaScript event loop turn [26], and builds signatures of these behaviors from scripts known to be abusing user privacy. We base our ground truth of known-bad behaviors on scripts blocked by the popular crowdsourced filter lists (i.e., EL and EP), and generate signatures to identify patterns in how currently-blocked scripts interact with the DOM, JavaScript APIs (e.g., the Date API, cookies, storage APIs, etc), and initiate network requests. We then use these signatures of known-bad behaviors to identify the same code being delivered from other URLs, bundled with other code, or inlined in a site.

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

In total we build 1,995,444 high-confidence signatures of privacy-and-security harming behaviors (defined in Section III) from 11,212 scripts blocked by EasyList and EasyPrivacy. We then use our browser instrumentation and collected signatures to identify 3,589 new scripts containing identically-performing privacy-and-security harming behavior, served from 1,965 domains and affecting 12.48% of websites. Further, we use these signatures, along with code analysis techniques from existing research, to categorize the *method* trackers use to evade filter lists. Finally, we use our instrumentation and signatures to generate new filter list rules for 720 URLs that are moved instances of known tracking code, which contribute to 65.79% of all instances of filter list evasion identified by our approach, and describe how our tooling and findings could be used to build defenses against the rest of the 34.21% instances of filter list evasions.

A. Contributions

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

1) The **design and implementation** of a system for generating signatures of JavaScript behavior. These signatures are robust to popular obfuscation and JavaScript bundling tools and rely on extensive instrumentation of the Blink and V8 systems.
2) A **web-scale measurement of filter list evasion**, generated by measuring how often privacy-sensitive behaviors of scripts labeled by EasyList and EasyPrivacy are repeated by other scripts in the Alexa 100K.
3) A **quantified taxonomy of filter list evasion techniques** generated by how often scripts evade filter lists by changing URLs, inlining, or script bundling.
4) 586 new **filter list rules** for identifying scripts that are known to be privacy-or-security related, but evade existing filter lists by changing URLs.

B. Research Artifacts and Data

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

Finally, we share a set of AdBlock Plus compatible filter list additions to block cases of websites moving existing scripts to new URLs (i.e., the subset of the larger problem that *can* be defended against by existing tools) [5]. We note many of these filter list additions have already been accepted by existing filter list maintainers, and note those cases.

II. Problem Area

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

A. Current Content Blocking Focuses on URLs

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

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

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

B. URL-Targeting Systems Are Circumventable

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

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

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

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

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

[1]One exception is uBlock Origin, which, when installed in Firefox, uses non-standard API's[17] to allow some filtering of inline script contents. However, because this technique is rare, and also trivially circumvented, we do not consider it further in this work.

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

[3]http://browserify.org/

[4]https://parceljs.org/

C. Problem - Detection Mismatch

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

We propose an approach that aligns the content blocking decisions with the behaviors which are to be blocked. The rest of this paper presents such a system, one that makes blocking decisions based on patterns of JavaScript behavior, and not delivery URLs. Doing so provides both a way to measuring how often evasions currently occur, and the basis of a system for providing better, more robust privacy protections.

III. Methodology

This section presents the design of a system for building signatures of the privacy-and-security relevant behavior of JavaScript code, per event loop turn [26], when executed in a web page. The web has a single-threaded execution model, and our system considers the sum of behaviors each script engages in during each event loop turn, from the time the script begins executing, until the time the script yields control.

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

A. Difficulties in Building JavaScript Signatures

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

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

unpredictable effects of writing to `innerHTML`, or writing text inside a `<script>` element, among many others.

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

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

B. Signature Generation

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

This subsection proceeds by giving a high-level overview of each step, enough to evaluate its correctness and boundaries, but excluding some low-level details we expect not to be useful for the reader. However, we are releasing all of the code of this project to allow for reproducibility of our results and further research [5].

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

To solve this problem, we used and extended PageGraph,[8] a system for representing the execution of a page as a directed graph. PageGraph uses nodes to represent elements in a website's environment (e.g., DOM nodes, JavaScript units, fetched resources, etc.) and edges to describe the interaction between page elements. For example, an edge from a script element to a DOM node might encode the script setting an attribute on that DOM node, while an edge from a DOM node to a network resource might encode an image being fetched because of the `src` attribute on an `<img>` node. Figure 1 provides a simplified example of a graph generated by PageGraph.

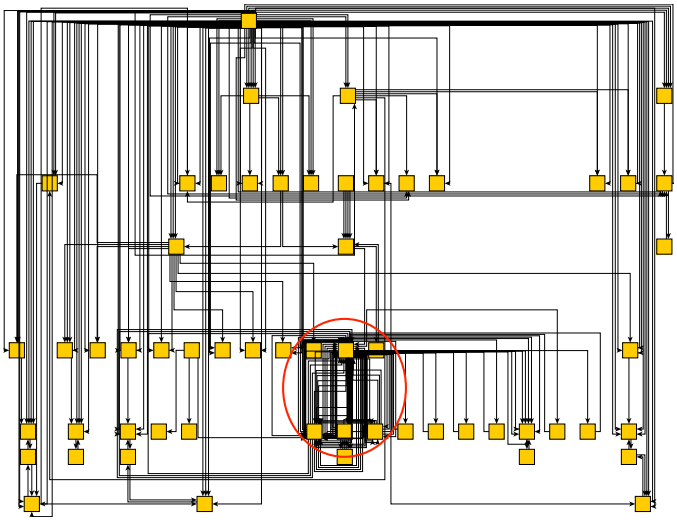

Fig. 1. Simplified rendering of execution graph for https://theoatmeal.com. The highlighted section notes the subgraph attributed to Google Analytics tracking code.

All edges and nodes in the generated graphs are fully ordered, so that the order that events occurred in can be replayed after the fact. Edges and nodes are richly annotated and describe, for example, the type of DOM node being created (along with parents and siblings it inserted alongside), the URL being fetched by a network request, or which internal V8 script id[9] a code unit in the graph represents.

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

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

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

3) Extracting Event-Loop Signatures: Next, we use the PageGraph generated graph representation of page execu-

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

[7]Section 2.C. of [19] includes more discussion of the difficulties of JavaScript attribution

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

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

Instrumented Event	Privacy?	Deterministic?
HTML Nodes		
node creation	no	yes
node insertion	no	yes
node modification	no	yes
node deletion	no	yes
remote frame activities	no	no
Network Activity		
request start	yes	some[1]
request complete	no	some[1]
request error	no	some[1]
API Calls		
timer registrations	no	yes
timer callbacks	no	no
JavaScript builtins	no	some[2]
storage access	yes	yes[3]
other Web APIs	no[4]	some

[1] Non-async scripts, and sync AJAX, occur in a deterministic order.
[2] Most builtins occur in deterministic order (e.g. Date API), though there are exceptions (e.g. setTimeout callbacks).
[3] `document.cookie`, `localStorage`, `sessionStorage`, and `IndexedDB`
[4] While many Web API can have privacy effects (e.g. WebRTC, browser fingerprinting, etc.) we do not consider such cases in this work, and focus only on the subset of privacy-sensitive behaviors relating to storage and network events.

TABLE I
PARTIAL LISTING OF EVENTS INCLUDED IN OUR SIGNATURES, ALONG WITH WHETHER WE TREAT THOSE EVENTS AS PRIVACY RELEVANT, AND WHETHER THEY OCCUR IN A DETERMINISTIC ORDER, GIVEN THE SAME JAVASCRIPT CODE.

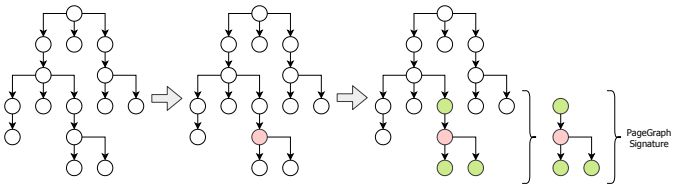

Fig. 2. PageGraph signature generation. The red node represents a script unit that executed privacy-related activity and the green nodes are the ones affected by the script unit during one event loop turn. The extracted signature is a subgraph of the overall PageGraph.

tion, along with the enumeration of deterministic behaviors, to determine the behaviors of each JavaScript unit during each event loop turn (along with deterministically occurring "downstream" effects in the page). Specifically, to obtain the signature of JavaScript behaviors that happened during one event-loop turn, our approach extracts the subgraph depicting the activities of each JavaScript unit, for each event loop turn, that occurred during page execution (see Figure 2). However, for the sake of efficiency, we do not generate signatures of script activity that do not affect privacy. Put differently, each signature is a subgraph of the entire PageGraph generated graph, and *encodes at least one privacy-relevant event*. Hereafter we refer to the extracted signatures, which depict the per-event-loop behaviors of JavaScript, as *event-loop signatures*.

For the purposes of this work, we consider privacy-and-security related events to consist solely of (i) storage events,[10] because of their common use in tracking, and (ii) network events,[11] since identifiers need to be exfiltrated at some point for tracking to occur. We note that there are other types of events that could be considered here, such as browser fingerprinting-related APIs, but reserve those for future work.

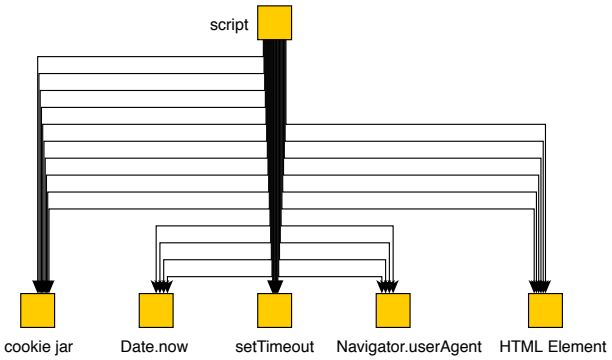

Fig. 3. Simplified signature of Google Analytics tracking code. Edges are ordered by occurrence during execution, and nodes depict Web API and DOM elements interacted with by Google Analytics.

As an example, Figure 3 shows a (simplified) subgraph of the larger graph from Figure 1, depicting what the Google Analytics script did during a single event loop turn: accessing cookies several times (storage events), reading the browser user-agent string, creating and modifying an `<img>` element (and thus sending out network requests), etc.

At a high level, to extract event-loop signatures from a PageGraph generated graph, we determined which JavaScript operations occurred during the same event-loop turn by looking for edges with sequential ids in the graph, all attached to or descending from a single script unit. As a page executes, control switches between different script units (or other actions on the page); when one script yields its turn on the event loop, and another script begins executing, the new edges in a graph will no longer be attached to the first script, but to the newly executing one. Event-loop turns are therefore encoded in the graph as subgraphs with sequential edges, all related to the same script node. We discuss some limitations of this approach, and why we nevertheless preferred it to possible alternatives in SectionVI-E.

More formally, we build signatures of privacy-and-security affecting JavaScript behavior using the following algorithm:[12].

(i) Extract all edges in the graph representing a privacy-effecting JavaScript operation (as noted in Table I).
(ii) Attribute each of these edges to the JavaScript unit responsible. If no script is responsible (e.g., a network request was induced by the parser), abort.

[10] i.e. cookies, localStorage, sessionStorage, IndexedDB
[11] both direct from script (e.g., AJAX, fetch) and indirect (e.g., `<img>`)
[12] This description omits some implementation specific details and post-processing techniques that are not fundamental to the approach. They are fully documented and described in our shared source code [5]

(iii) Extract the maximum subgraph containing the relevant edge and responsible JavaScript code unit comprising all sequentially occurring nodes and edges. This is achieved by looking for edges that neighbor the subgraph, and which occurred immediately before (or after) the earliest (or latest) occurring event in the subgraph. If an edge is found, add it and the attached nodes to the subgraph.
(iv) Repeat step 3 until no more edges can be added to the subgraph.

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

C. Privacy Behavior Ground Truth

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

We used EasyList and EasyPrivacy to build a ground truth determination of privacy-harming JavaScript behaviors. If a script was identified by an EasyList or EasyPrivacy filter rule for blocking, and was not excepted by another rule, then we considered all the signatures generated from that code as privacy-harming, and thus should be blocked. This measure builds on the intuition that filter rules block known bad behavior, but miss a great deal of additional unwanted behavior (for the reasons described in Section II). Put differently, this approach models filter lists as targeting behaviors in code units (and that they target URLs as an implementation restriction), and implicitly assumes that filter lists have high precision but low (or, possibly just lower) recall in identifying privacy-and-security harming behaviors.

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

D. Determining Privacy-Harming Signatures

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

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

This minimal signature size was determined by starting with an initial signature size of 5 edges and 4 nodes, and then doing a manual evaluation of 25 randomly sampled matches between signatures (i.e., cases where the same signature was generated by a blocked and not-blocked script). We had our domain-expert then examine each of the 25 randomly sampled domains to determine whether the code units actually included the same code and functionality. If the expert encountered a false positive (i.e., the same signature was generated by code that was by best judgement unrelated) the minimum graph size was increased, and 25 new matches were sampled. This process of manual evaluation was repeated until the expert did not find any false positives in the sampled matches, resulting in a minimum graph size of 13 edges and 4 nodes.

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

IV. Results

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

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

TABLE II
STATISTICS REGARDING OUR CRAWL OF THE ALEXA 100K, TO BOTH BUILD SIGNATURES OF KNOWN TRACKING CODE, AND TO USE THOSE SIGNATURES TO IDENTIFY NEW TRACKING CODE.

A. Initial Web Crawl Data

We began by using our PageGraph-enhanced browser to crawl the Alexa 100K, which we treated as representative of the web as a whole. We automated our crawl using a puppeteer-based tool, along with extensions to PageGraph to support the DevTools interface [13].

For each website in the Alexa 100K, our automated crawler visited the domain's landing page and rested for 60 seconds to allow for sufficient time for scripts on the page to execute. We then retrieved the PageGraph generated graph-representation of each page's execution, encoded as a GraphML-format XML file.

Table II presents the results of this crawl. From the Alexa 100K, we got a successful response from the server from 88,035 domains, and were able to generate the graph representation for 87,941. We attribute not being able successfully crawl 11,965 domains to a variety of factors, including bot detection scripts [18], certain sites being only accessible from some IPs [36], [6], and regular changes in website availability among relatively unpopular domains. This number of unreachable domains is similar to those found by other automated crawl studies [32], [21]. A further 4,286 domains could not be measured because they used browser features that PageGraph currently does not correctly attribute (most significantly, module scripts).

B. Signature Extraction Results

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

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

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

1) Contained at least one privacy-or-security relevant event (defined in Section III-B3)
2) Occurred at least once in a script blocked by EasyList or EasyPrivacy (i.e., a *blocked script*)
3) Occurred at least once in a script *not* blocked by EasyList and EasyPrivacy (i.e., an *evaded script*).
4) Have a minimum size of 13 edges and 4 nodes (see Section III-D)

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

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

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

Figure 4 shows the distribution of the number of unique signatures in our ground truth set that were found in scripts on each visited domain in our crawl of the Alexa top 100K (56,390 domains have at least one script where signatures from our ground truth set were found), as well as the distribution of the number of unique ground truth signatures in each script unit where such signatures were found. As we did in Table III, for comparison here we also plot the same statistics for the small signatures.

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

TABLE III
THE NUMBER OF SCRIPTS WHOSE BEHAVIORS MATCH SIGNATURES FROM OUR GROUND TRUTH SET, BOTH IN TOTAL AND BROKEN DOWN BY WHETHER THEY ARE BLOCKED BY EL/EP. FOR COMPARISON WE ALSO SHOW THE SAME STATISTICS FOR THE DISCARDED SMALL SIGNATURES.

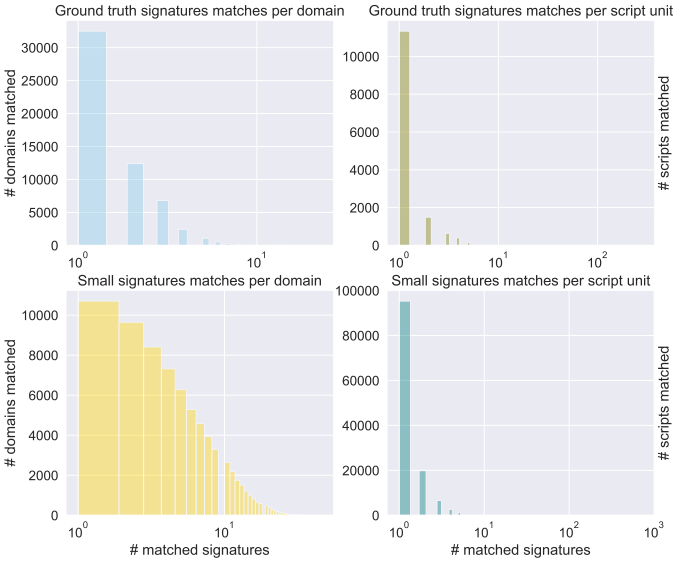

Fig. 4. Distribution of the number of signatures per domain and the number of such signatures in each matched script unit for our ground truth dataset and for the small signatures dataset.

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

C. Impact on Browsing

Next, we attempt to quantify the practical impact on privacy and security from filter list evasion. Here the focus is not on the number of parties or scripts engaging in filter list evasion, but on the number of websites users encounter on which filter list evasion occurs. We determined this by looking for the

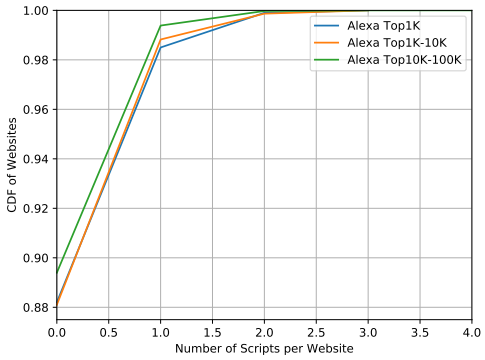

Fig. 5. Total number of evaded scripts per website, for "popular" (Alexa top 1K), "medium" (Alexa top 1K - 10K), and "unpopular" (Alexa top 10K - 100K) websites.

number of sites in the Alexa 100K (in the subset we were able to record correctly) that included at least one script matching a signature from a blocked script, but which was not blocked.

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

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

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

Figure 5 summarizes this measurement as a CDF of how many instances of filter list evasion occur on sites in each group. As the figure shows, filter list evasion occurred roughly evenly on sites in each group; we did not observe any strong relationship between site popularity and how frequently filter lists were evaded.

D. Popularity of Evading Parties

Finally, we measure the relationship, in regards to domain popularity (i.e., the delta in their ranking), between the sites

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

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

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

Figure 6 shows the distribution of all the ranking deltas, calculated as described above (since we cannot approximate the relative popularity for the 538 pairs where both domains are outside of Alexa top 1M, we excluded them from Figure 6). Note that this distribution is very symmetric: about as many of the domain pairs have negative delta as those that have positive delta, and the distribution on both sides of the x-axis closely mirrors each other. We believe this is mostly due Alexa favoring first-party domains when calculating popularity-metrics; according to Alexa's documentation, multiple requests to the same URL by the same user counts as only one page view for that URL on that day [7]. Thus, if on a single day the user visits multiple sites that contain tracking scripts loaded from the same tracker domain, then Alexa counts those as only one visit to that tracker domain. As a result, domains that host tracking scripts tend to occupy the middle range of the Alexa ranking, and their tracking scripts are equally likely to be hosted on websites both before and after them in the Alexa rankings (web developers often choose host third-party tracking scripts on domains that they control, while at the same time minifying/obfuscating, or otherwise bundling the scripts, in order to evade filter list blocking, and we provide a taxonomy of such evasion attempts in Section V).

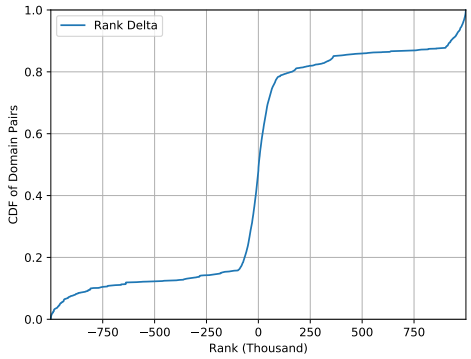

Fig. 6. Distribution of the delta in Alexa ranking of domains hosting EL/EP blocked scripts vs. evaded scripts that matched the same signature. A negative delta means the script is moved from a popular domain to a less popular domain. The x-axis of domain rank delta is in thousands.

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

TABLE IV
MOST POPULAR DOMAINS HOSTING RESOURCES THAT WERE BLOCKED BY FILTER LISTS. THE FIRST COLUMN RECORDS THE HOSTING DOMAIN, THE NEXT COLUMN THE NUMBER OF DOMAINS LOADING RESOURCES FROM THE HOSTING DOMAIN, THE THIRD COLUMN THE NUMBER OF UNIQUE URLS REQUESTED FROM THE DOMAIN, AND THE FINAL COLUMN THE COUNT OF (NON-UNIQUE) BLOCKED, HARMFUL SCRIPTS LOADED FROM THE DOMAIN.

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

Incidentally, we also measured what domains hosted scripts most often blocked by filter lists, and which domains hosted

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

TABLE V
MOST POPULAR DOMAINS HOSTING SCRIPTS THAT EVADED FILTER LISTS, BUT MATCHED KNOWN HARMFUL SCRIPTS. THE FIRST COLUMN RECORDS THE HOSTING DOMAIN, THE SECOND COLUMN THE NUMBER OF DOMAINS THAT REFERENCED THE HOSTING DOMAIN, THE THIRD COLUMN THE NUMBER OF UNIQUE, EVADING URLS ON THE HOSTING DOMAIN, AND THE FINAL COLUMN THE COUNT OF (NON-UNIQUE) NON-BLOCKED, HARMFUL SCRIPTS LOADED FROM THE DOMAIN.

scripts that contained known-harmful behavior, but evaded detection. Tables IV and V record the results of these measurements. We find that Google properties are the most frequently blocked resources on the web (Table IV), both for tracking and advertising resources, followed by the `addthis.com` widget for social sharing (that also conducts tracking operations).

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

V. EVASION TAXONOMY

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

TABLE VI
TAXONOMY AND QUANTIFICATION OF OBSERVED FILTER LIST EVASION TECHNIQUES IN THE ALEXA 100K.

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

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

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

For each taxonomy label, we perform code analysis and comparison techniques using Esprima,[14] a popular and open-source JavaScript parsing tool. We use Esprima to generate ASTs for each JavaScript file to look for structural similarities between code units. By comparing the AST node types between scripts we are resilient to code modifications that do not affect the structure of the program, like renaming variables or adding/changing comments. We consider signatures from scripts *not blocked* by EasyList and EasyPrivacy, but matching a signature generated by a script blocked by EasyList and EasyPrivacy to determine the relationship of the non-blocked script to the blocked scripts.

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

A. Moving Code

The simplest filter list evasion strategy we observed is moving tracking code from a URL identified by filter lists to a URL unknown to filter lists. This may take the form of just copying the code to a new domain but leaving the path fixed,[15] leaving the script contents constant but changing the path,[16] or some combination of both. We also include in this category cases where code was moved to a new URL and minified or otherwise transformed without modifying the code's AST.

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

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

1) Classification Methodology: We detect cases of "moving code" evasion by looking for cases where code with the identical AST appears at both blocked and not-blocked URLs. For each script that generated a signature that was *not* blocked by EasyList or EasyPrivacy (i.e, an evading script), we first generated the AST of the script, and then generated a hash from the ordered node types in the AST. We then compared this AST hash with the AST hash of each blocked script that also produced the same signature. For any not-blocked

[14]https://esprima.org/
[15]https://tracker.com/track.js → https://example.org/track.js
[16]https://tracker.com/track.js → https://tracker.com/abdcjd.js

script whose AST hash matched one of the blocked scripts, we labeled that as a case of evasion by "moving code". We observed 720 unique script units (7,924 instances) that evade filter lists using this technique in in the Alexa 100K.

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

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

B. Inlining Code

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

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

1) Classification Methodology: Inlining is the most straightforward evasion type in our taxonomy scheme. Since PageGraph records the source location of each JavaScript unit that executes during the loading of a page, we can easily determine which scripts were delivered as inline code. We then compare the AST hashes (whose calculation method we described in Section V-A1) of all inline scripts to all blocked scripts that generated identical signatures. We labeled all cases where the hashed-AST of an inline script matched the hashed-AST of a script blocked by EasyList or EasyPrivacy, and both scripts generated identical signatures, as cases of evasion by "inlining". We observed 498 cases of sites moving blocked, privacy harming behaviors inline.

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

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

C. Combining Code

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

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

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

2) Case Study: Insights JavaScript SDK: Insights is a JavaScript tracking, analytics and telemetry package from Microsoft,[17] that allows application developers to track and record visitors. It includes functionality identified by EasyPrivacy as privacy-harming, and is blocked by the filter rule `||msecnd.net/scripts/a/ai.0.js`.

In order to evade EasyPrivacy, some sites copy the Microsoft Insights code from the Microsoft provided URL, and included it, among many other libraries, in a single JavaScript file. This process is sometimes called "bundling" or "packing". As one example, the website https://lindex.com includes the

[17] https://docs.microsoft.com/en-us/azure/azure-monitor/overview

Insights library, along with the popular Vue.js and Mustache libraries, in a single URL,[18] packaged together using the popular WebPack[19] library.

D. Included Library

Finally, filter lists are unable to protect against JavaScript code including common privacy-harming libraries. Such libraries are rarely, if ever, included by the site directly, but are instead downstream dependencies by the libraries directly included on the website. These cases are common, as JavaScript build systems emphasize small, reusable libraries. Downstream libraries are difficult for filter lists to target because there is no URL filter list maintainers can block; instead, filter list maintainers can only target the diverse and many bespoke JavaScript applications that include the libraries.

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

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

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

Other libraries that include the Adobe Visitor API code though are missed by filter lists, and thus are undefended against. For example, the site ferguson.com indirectly loads the Visitor API code on its site, through the site's "setup" code.[20] There many other similar examples of downstream, privacy-and-security harming libraries included by diverse JavaScript applications, following this same pattern.

[18]https://lindex.com/web-assets/js/vendors.8035c13832ab6bb90a46.js

[19]https://webpack.js.org/

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

VI. Discussion

A. Comparison to Hash-Based Detection

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

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

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

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

B. Countermeasures

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

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

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

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

2) Event-Loop Turns Without Network: Instances of code being inlined, or privacy-or-security affecting code being combined with other code, are more difficult to defend against, and require runtime modifications. These have *not* been implemented as part of this work, but we discuss possible approaches for doing so here.[21] We note that the single-threaded model of the browser means that signature-matching state only needs to be maintained per JavaScript content, to track the behavior of the currently executing script; state does not need to be maintained per code unit.

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

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

C. Accuracy Improvements

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

D. Web Compatibility

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

E. Limitations

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

1) Automated Crawling: The most significant limitation is our reliance on automated-crawls to build signatures. While such automated crawls are useful for covering a large portion of the web, they have significant blind spots, including missing scripts only accessible after authentication on a site, or only after performing complex interactions on a page. Prior work has attempted to deal with this though paid research-subject volunteers [33], or other ways of approximating real world usage. Such efforts are beyond the scope of this project, but we note them here for completeness.

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

While our system could account for some of these cases through further crawling (to account for more possible code

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

paths) or generalizations in signature generation, we note this attack as a current limitation and area for future work.

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

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

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

VII. Related Work

A. Blocking trackers

The current line of defense that most users have against web tracking is via browser extensions [1], [15], [4], [2]. These extensions work by leveraging hand-crafted filter lists of HTML elements and URLs that are connected with advertisers and trackers [3]. There are also dynamic approaches for blocking, like Privacy Badger from EFF [10], which tracks images, scripts and advertising from third parties in the visited pages and blocks them if it detects any tracking techniques. The future of browser extensions as web tracking prevention tools is currently threaten by the transition to the newest version of WebExtensions Manifest v3 [8], which limits the capabilities of dynamically making decisions to block content.

Previous research has also focused on automated approaches to improve content blocking. Gugelmann *et al.*, built a classifier for identifying privacy-intrusive Web services in HTTP traffic [14]. NoMoAds leverages the network interface of a mobile device to extract features and uses a classifier to detect ad requests [30].

B. Instrumenting the browser

Extracting information from the browser is mandatory step into understanding web tracking. Previous approaches, like OpenWPM, have focused on leveraging a browser extension to monitor the events of a visited page [11]. In-band approaches like OpenWPM inject JS into the visited page in order to capture all events, which can affect their accuracy, as they are running at the same level as the monitored code. Recently, we have observed a shift in pushing more browser instrumentation out-of-band (in-browser) [19], [23], [20]. In this paper, we follow a similar out-of-band approach, where we build signatures of tracking scripts based on the dynamic code execution by instrumenting Blink and V8 in the Chromium browser.

C. Code Similarity

Code similarity is a well-established research field in the software engineering community [29]. From a security perspective, finding code similarities with malicious samples has been explored in the past. Revolver [22] performed large-scale clustering of JavaScript samples in order to find similarities in cases where the classification is different, automatically detecting this way evasive samples.

Ikram *et al.* [16], suggested the use of features from JavaScript programs using syntactic and structural models to build a classifier that detects scripts with tracking [16]. Instead of relying on syntactic and structural similarity, in our work we identify tracking scripts based on the tracking properties of their execution in the browser, defeating this way techniques like obfuscation [31] and manipulation of ASTs [12].

D. Other Content Blocking Strategies

Another approach to block content is via perceptual detection of advertisements [34], [27]. This approach is based on the identifying advertisements based on known visual patterns that they have, such as the AdChoices standard [9]. Although this is an exciting new avenue of blocking content on the web, there is already work that aims to create adversarial attacks against perceptual ad blocking [35].

VIII. Conclusion

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

In this work we make several contributions to begin solving this problem, by identifying malicious code using highly granular, event-loop turn level signatures of runtime JavaScript behavior, using a novel system of browser instrumentation and graph-based signature generation. We contribute not only the first Web-scale measurement of how much evasion is occurring on the Web, but also the ground work for practical defenses. To further contribute to the goal a privacy-and-security respecting Web, we also contribute the source code for our instrumentation and signature generation systems, the

raw data gathered during this work, and filter list rules that can help users of existing tools defend against a subset of the problem [5].

IX. Acknowledgements

We would like to thank our shepherd Ben Stock, and our anonymous reviewers for their insightful feedback and comments. This work was supported by the Office of Naval Research (ONR) under grant N00014-17-1-2541, by DARPA under agreement number FA8750-19-C-0003, and by the National Science Foundation (NSF) under grant CNS-1703375.

References

[1] Adblock Plus. https://adblockplus.org/.
[2] Disconnect. https://disconnect.me/.
[3] EasyList and EasyPrivacy filter lists. https://easylist.to/.
[4] Ghostery. https://www.ghostery.com/.
[5] Semantic signatures. https://github.com/semantic-signatures/semantic-signatures, 2020.
[6] Sadia Afroz, Michael Carl Tschantz, Shaarif Sajid, Shoaib Asif Qazi, Mobin Javed, and Vern Paxson. Exploring server-side blocking of regions. *arXiv preprint arXiv:1805.11606*, 2018.
[7] Alexa. How are Alexa's traffic rankings determined? https://support.alexa.com/hc/en-us/articles/200449744-How-are-Alexa-s-traffic-rankings-determined-.
[8] Chromium Blog. Trustworthy Chrome Extensions, by default. https://blog.chromium.org/2018/10/trustworthy-chrome-extensions-by-default.html.
[9] Digital Advertising Alliance (DAA). Self Regulatory Principles for Online Behavioral Advertising. https://digitaladvertisingalliance.org/sites/aboutads/files/DAA_files/seven-principles-07-01-09.pdf, 2009.
[10] EFF. Privacy Badger. https://www.eff.org/privacybadger.
[11] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2016.
[12] Aurore Fass, Michael Backes, and Ben Stock. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. 2019.
[13] Kiran Garimella, Orestis Kostakis, and Michael Mathioudakis. Ad-blocking: A study on performance, privacy and counter-measures. In *Proceedings of the 2017 ACM on Web Science Conference*, pages 259–262. ACM, 2017.
[14] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. An automated approach for complementing ad blockers' blacklists. *Proceedings of the Symposium on Privacy Enhancing Technologies (PETS)*, 2015.
[15] Raymond Hill. uBlock Origin. https://github.com/gorhill/uBlock.
[16] Muhammad Ikram, Hassan Jameel Asghar, Mohamed Ali Kaafar, Anirban Mahanti, and Balachandar Krishnamurthy. Towards seamless tracking-free web: Improved detection of trackers via one-class learning. *Proceedings of the Symposium on Privacy Enhancing Technologies (PETS)*, 2017.
[17] Mozilla Inc. webrequest.filterresponsedata() - mozilla — mdn. https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest/filterResponseData.
[21] Jordan Jueckstock, Shaown Sarker, Peter Snyder, Panagiotis Papadopoulos, Matteo Varvello, Benjamin Livshits, and Alexandros Kapravelos. The blind men and the internet: Multi-vantage point web measurements. *arXiv preprint arXiv:1905.08767*, 2019.
[18] Luca Invernizzi, Kurt Thomas, Alexandros Kapravelos, Oxana Comanescu, Jean-Michel Picod, and Elie Bursztein. Cloak of visibility: Detecting when machines browse a different web. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 743–758. IEEE, 2016.
[19] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. Adgraph: A graph-based approach to ad and tracker blocking. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2020.
[20] Jordan Jueckstock and Alexandros Kapravelos. VisibleV8: In-browser Monitoring of JavaScript in the Wild. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2019.
[22] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Chris Kruegel, and Giovanni Vigna. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *Proceedings of the USENIX Security Symposium*, 2013.
[23] Bo Li, Phani Vadrevu, Kyu Hyung Lee, and Roberto Perdisci. Jsgraph: Enabling reconstruction of web attacks via efficient tracking of live in-browser javascript executions. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2018.
[24] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing your enemy: understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 674–686. ACM, 2012.
[25] Ben Miroglio, David Zeber, Jofish Kaye, and Rebecca Weiss. The effect of ad blocking on user engagement with the web. In *Proceedings of the 2018 World Wide Web Conference*, pages 813–821. International World Wide Web Conferences Steering Committee, 2018.
[26] Mozilla. Event loop. https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Event_loop, 2020.
[27] Adblock Plus. Sentinel - the artificial intelligence ad detector. https://adblock.ai/.
[28] Enric Pujol, Oliver Hohlfeld, and Anja Feldmann. Annoyed users: Ads and ad-block usage in the wild. In *Proceedings of the 2015 Internet Measurement Conference*, pages 93–106. ACM, 2015.
[29] Chanchal Kumar Roy and James R Cordy. A Survey on Software Clone Detection Research. *Queen's School of Computing, Technical Report*, 2007.
[30] Anastasia Shuba, Athina Markopoulou, and Zubair Shafiq. Nomoads: Effective and efficient cross-app mobile ad-blocking. *Proceedings of the Symposium on Privacy Enhancing Technologies (PETS)*, 2018.
[31] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. Anything to Hide? Studying Minified and Obfuscated Code in the Web. In *Proceedings of the Web Conference (WWW)*, 2019.
[32] Peter Snyder, Lara Ansari, Cynthia Taylor, and Chris Kanich. Browser feature usage on the modern web. In *Proceedings of the 2016 Internet Measurement Conference*, pages 97–110. ACM, 2016.
[33] Peter Snyder, Cynthia Taylor, and Chris 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.
[34] Grant Storey, Dillon Reisman, Jonathan Mayer, and Arvind Narayanan. The future of ad blocking: An analytical framework and new techniques. *arXiv preprint arXiv:1705.08568*, 2017.
[35] Florian Tramèr, Pascal Dupré, Gili Rusak, Giancarlo Pellegrino, and Dan Boneh. AdVersarial: Defeating Perceptual Ad Blocking. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.
[36] Michael Carl Tschantz, Sadia Afroz, Shaarif Sajid, Shoaib Asif Qazi, Mobin Javed, and Vern Paxson. A bestiary of blocking: The motivations and modes behind website unavailability. In *8th USENIX Workshop on Free and Open Communications on the Internet (FOCI 18)*, 2018.