Blocking JavaScript Without Breaking the Web: An Empirical Investigation

Abdul Haddi Amjad
Virginia Tech
hadiamjad@vt.edu

Zubair Shafiq
University of California, Davis
zubair@ucdavis.edu

Muhammad Ali Gulzar
Virginia Tech
gulzar@cs.vt.edu

ABSTRACT

Modern websites heavily rely on JavaScript (JS) to implement legitimate functionality as well as privacy-invasive advertising and tracking. Browser extensions such as NoScript block any script not loaded by a trusted list of endpoints, thus hoping to block privacy-invasive scripts while avoiding breaking legitimate website functionality. In this paper, we investigate whether blocking JS on the web is feasible without breaking legitimate functionality. To this end, we conduct a large-scale measurement study of JS blocking on 100K websites. We evaluate the effectiveness of different JS blocking strategies in tracking prevention and functionality breakage. Our evaluation relies on quantitative analysis of network requests and resource loads as well as manual qualitative analysis of visual breakage. First, we show that while blocking all scripts is quite effective at reducing tracking, it significantly degrades functionality on approximately two-thirds of the tested websites. Second, we show that selective blocking of a subset of scripts based on a curated list achieves a better trade-off. However, there remain approximately 15% "mixed" scripts, which essentially merge tracking and legitimate functionality and thus cannot be blocked without causing website breakage. Finally, we show that fine-grained blocking of a subset of JS methods, instead of scripts, reduces major breakage by 3.8× while providing the same level of tracking prevention. Our work highlights the promise and open challenges in fine-grained JS blocking for tracking prevention without breaking the web.

KEYWORDS

privacy, web, software engineering

1 INTRODUCTION

JavaScript is often used to provide rich user experiences on the web. The volume of JavaScript on the web has steadily increased over the years. The median web page load today ships 500+ kilobytes of JavaScript [73]. While some of it is used to implement various libraries and frameworks (*e.g.,* jQuery, React), almost half of it is third-party scripts that implement advertising and tracking services. The research community is concerned about the negative impact of JavaScript on performance [28, 47, 72], security [32, 36, 54, 76], and privacy [34, 38, 44, 56, 57].

Due to these concerns, there is a small but active community of web users who want to use the web without JavaScript. In fact, all major browsers now provide a native way for users to block all JavaScript [12]. Moreover, users can employ browser extensions such as NoScript [15] that block all scripts – except those from a trusted source. HTML5 now also supports the `noscript` element that allows web developers to gracefully support such browsers that do not support scripting [13].

While blanket JavaScript blocking does alleviate these concerns, it inevitably breaks the legitimate website functionality. The privacy community has developed content-blocking tools that selectively block tracking resources (*e.g.,* scripts) on a webpage. Privacy-enhancing content blockers, such as uBlock Origin [11], block network requests to known trackers by matching request URLs with manually curated filter lists [5, 7].

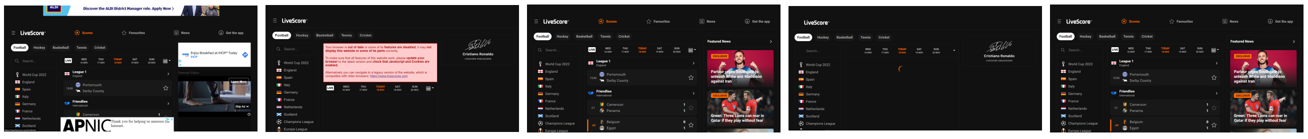
Since these privacy-enhancing content blockers are now used by more than one-third of web users [2, 53], there are strong financial incentives for web developers to evade content blockers. The typical evasion strategy is to manipulate the URLs, *e.g.,* change the URL path or hostname such that filter lists are no longer effective [25, 39]. This has led to an arms race where filter lists must be promptly updated in response to such evasion attempts [30, 49, 68]. Filter list curators have also made a concerted effort to selectively block the underlying scripts from downloading or execution that are responsible for initiating tracking requests. In response, a new evasion strategy has emerged where web developers attempt to mix tracking and functional code in the same script (*e.g.,* JS bundling [30]). Privacy-enhancing content blockers risk breaking a webpage if they block such scripts or compromise user privacy if they do not.

Privacy-enhancing content blockers aim to eliminate tracking while preserving website functionality. However, if they are forced to choose — *e.g.,* when tracking and functional code is mixed —they always prioritize functionality preservation. This is because most users tend to disable privacy-enhancing content blockers if they break legitimate website functionality. Recent research [26, 68] has shown that many websites now mix functional and tracking code that renders privacy-enhancing content blocking useless.

In this paper, we conduct a first-of-its-kind empirical investigation of JS blocking. To this end, we quantitatively and qualitatively evaluate the impact of different granularities of JS blocking on 100K websites. Our goal is to assess whether it is feasible to eliminate tracking effectively while preserving website functionality at different granularities of JS code *i.e.,* script and method. Beyond blanket JS blocking, we first investigate selective blocking of tracking scripts as well as mixed scripts. We further expand our investigation to the effectiveness of method-level blocking.

Our large-scale automated analysis of 100K websites reaffirms that blanket JS blocking indeed eliminates tracking, but it also breaks website functionality on approximately two-thirds of the tested websites. We then show that selective blocking of tracking scripts mitigates tracking without degrading website functionality,

Proceedings on Privacy Enhancing Technologies 2023(3), 391–404
© 2023 Copyright held by the owner/author(s).
https://doi.org/10.56553/popets-2023-0087



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

Figure 1: The snapshot of `livescore.com` with (a) control-setting (no content blocker), (b) NoScript (default setting), (c) uBlock Origin (default setting), (d) mixed script blocked (`_app-*.js`), and (e) JS method blocked (method in `_app-*.js`).

but there remains a significant fraction of scripts that mix tracking and functional behavior. Specifically, we find that 14.6% of the scripts exhibit both tracking and functional (*i.e.,* mixed) behavior. We then adapt Spectra-based fault localization (SBFL), a popular faulty code localization technique, to further localize tracking to the constituent methods of these mixed scripts. We find that method-level blocking of tracking methods significantly reduces website breakage while providing the same level of tracking prevention.

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

We summarize our key findings and contributions below:

- We find that method-level JS blocking is able to prevent tracking on par with script-level JS blocking while improving functionality preservation by 3.8× major breakage and 1.5× minor breakage.
- By comparing two web crawls conducted one year apart, we find a 14% increase in the number of websites that employ mixed scripts on 100K websites.
- Even at the method-level granularity, there remain 6% mixed methods that combine tracking and functionality and require even deeper program analysis for effective blocking without breaking functionality.
- The data set crawled for this study offers a full-scale view of JS code integration on today's websites, presenting a detailed lineage of tracking, functional, and mixed JS code units across 100K websites.

Data Availability: Our source code and data is available at https://zenodo.org/record/6526537.

2 MOTIVATION

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

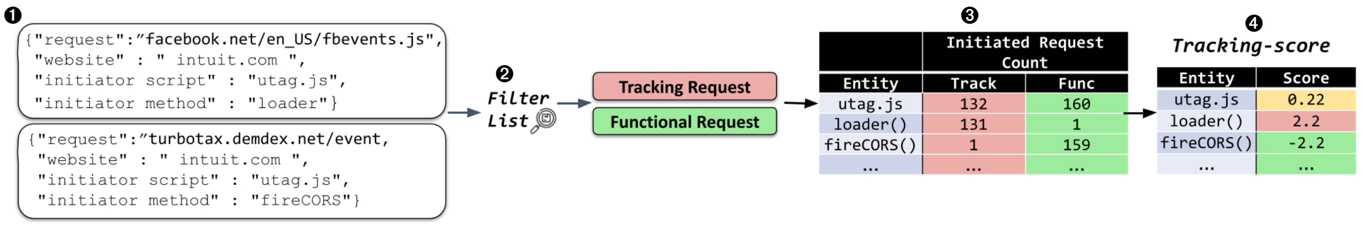
No JS blocking. Let's take the example of `livescore.com`, a top-10 ranked sports website [14]. We first load the homepage of `livescore.com` in a stock Chrome browser without any JavaScript intervention. Loading this webpage results in 294 network requests in 11 seconds, including 83 requests to fetch scripts and 175 requests initiated by these scripts. For motivation, consider two of these scripts that initiate network requests to *known*[1] tracking endpoints: `gtm.js` served by `googletagmanager.com` and `_app-*.js` served by `livescore.com`. `gtm.js` sends network requests to `googleadservices.com` and `google-analytics.com`. `_app-*.js` sends network requests to `doubleclick.net`. Upon careful inspection, we find that `_app-*.js` also sends a network request to `livescore.com/api/announcements/` that includes known tracking cookies such as `_gads` [29, 60]. While both scripts are responsible for network requests to tracking endpoints, `_app-*.js` is a mixed script that seems to implement both legitimate website functionality (*e.g.,* add media, populate game statistics) and tracking. Figure 1 (a) shows the homepage of `livescore.com` in the control configuration (without any blocking).

Blanket JS blocking. The naive way is to block all JS on `livescore.com` at the page load time. This capability is available in all major browsers [12]. While this approach blocks all the aforementioned tracking requests, it also completely breaks the website functionality. `livescore.com` becomes unusable and in fact notifies the user[2] that JS needs to be enabled for the website to display correctly. NoScript [15] also blocks all JS on `livescore.com`, including `gtm.js` served by `googletagmanager.com` and `_app-*.js` served by `livescore.com`. This again completely breaks the website functionality. Figure 1 (b) shows the homepage of `livescore.com` when NoScript [15] is used.

Selective JS blocking. We next use a tracker blocking tool, called uBlock Origin [11], on `livescore.com`. Note that these tracker blocking tools do not specifically target JS. Instead, they use a curated filter list to block network requests to known tracking endpoints that may incidentally include network requests to fetch JS. Thus, compared to blanket JS blocking, uBlock Origin aims to block all network requests to known tracking endpoints while allowing other network requests. After loading `livescore.com` with uBlock Origin installed, we observe that `gtm.js` is blocked, thus eliminating all subsequent tracking network requests from `gtm.js`. However, instead of blocking `_app-*.js`, uBlock Origin blocks the network request to `doubleclick.net` while it allows the network request `livescore.com/api/announcements/` containing tracking cookies. Figure 1 (c) shows the homepage of `livescore.com` when uBlock Origin [11] is used. Although there is no website breakage, uBlock Origin has essentially decided not to block `_app-*.js` to avoid website breakage even though it results in tracking requests. As we elaborate later, trackers have been increasingly putting tracker blocking tools in such a bind.

[1]See, for example, Disconnect tracking protection list [4]

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



```
{"request":"facebook.net/en_US/fbevents.js",
 "website" : " intuit.com ",
 "initiator script" : "utag.js",
 "initiator method" : "loader"}
```

```
{"request":"turbotax.demdex.net/event,
 "website" : " intuit.com ",
 "initiator script" : "utag.js",
 "initiator method" : "fireCORS"}
```

	Initiated Request Count	
Entity	Track	Func
utag.js	132	160
loader()	131	1
fireCORS()	1	159
...	...	...

Entity	Score
utag.js	0.22
loader()	2.2
fireCORS()	-2.2
...	...

Figure 2: Steps for localizing tracking and functional JS code using Spectra-based fault localization. ❶ shows the two network requests on `intuit.com`**. Filter lists are used to label requests in ❷. Spectra-based fault localization is used to classify resources based on participation, as shown in ❸ and ❹.**

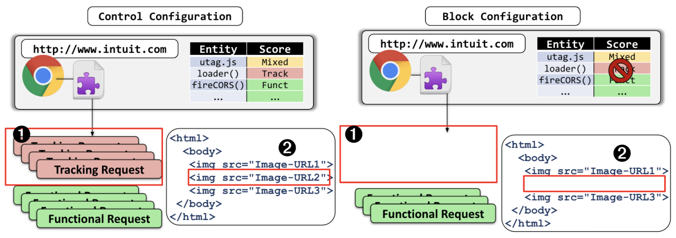


Figure 3: Illustration of the breakage metrics for automated JS blocking. Request count (❶) and HTML of website (❷) are compared with control configuration.

Tracking and Mixed JS blocking. To understand why uBlock Origin chose not to block `_app-*.js`, we next use uBlock Origin but also configure it to block `_app-*.js`. As shown in Figure 1 (d), this leads to a major functionality breakage on `livescore.com`; the navigation button, game statistics, and the featured news section are not rendered correctly. Put simply, there is a no-win situation when it comes to `_app-*.js`. Blocking it results in website breakage, and not blocking it results in tracking.

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

Listing 1: JS method `u` **that initiates tracking requests in script** `_app-*.js`**. We replace this method name with** `donotExecuteMe`**.**

Method-level JS blocking. Recent work [26, 68] has applied dynamic analysis to identify tracking methods in mixed scripts manually. Our analysis of network requests initiated by `_app-*.js` shows that the tracking requests were initiated by the method shown in Listing 1. As shown in Figure [14] (e), when this method in `_app-*.js` is blocked (*e.g.,* it is renamed such that all calls to this method are invalidated), the entire webpage renders completely while all tracking requests are also blocked. It is noteworthy that manually refactoring mixed scripts is not feasible at scale. Therefore, only a handful of mixed scripts have been refactored in prior work [18].

3 METHODOLOGY

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

3.1 Phase I: Automated JS Blocking Analysis

Figure 2 shows our automated JS blocking analysis pipeline comprising a JS collection step and JS code localization step. Figure 3 shows our JS blocking impact analysis step.

3.1.1 JavaScript Corpus Collection. We crawl landing pages of 100K randomly sampled websites from Tranco top-million list [63] using a custom-built Chrome extension. We spend 20 seconds on a page, exceeding the median `onLoad` time by 13.5 seconds on average. This allows us to capture the vast majority of the content fetched, which is consistent with over 90% of all webpages [20]. Nonetheless, we measure the impact of increasing the crawl time to 90 seconds on 200 web pages randomly sampled from 100K. We notice average differences of 2% and 5.2% in tracking and functional requests, respectively, causing an insignificant impact on our findings. Thus, we set the crawl time to 20 seconds.

For each webpage, our crawler outputs a JSON file that maps each network request to its initiator script and method (step ❶). We then label each network request and its initiator code (*e.g.,* JS script and methods) as tracking or functional using filter lists [5, 7] (step ❷). We use EasyList [5] and EasyPrivacy [7] that are used by existing content blockers such as uBlock Origin [11], Brave [3], and Adblock Plus [1]. These filter lists only do binary classification and tend to classify mixed resources as functional to avoid website breakage. This is an inherent limitation of filter lists that our work aims to highlight in the context of JavaScript blocking.

3.1.2 Localizing Tracking and Functional JS Code. Next, we classify each script and method using spectra-based "fault" localization (SBFL) [22, 43]. SBFL requires a set of failing and passing test cases. For every test, it simply collects the list of code units that participated in the test execution. Based on the test output, it labels the participating code units as either passing or failing. Finally, it compares the participation of code units in passing and failing tests and assigns a *score* to them.

We adapt SBFL to localize tracking code units (*i.e.,* scripts, methods). Instead of test cases, we analyze each network request and the methods and scripts in the call stack trace of the network request. For example, Figure 2-❶ shows two network requests on `intuit.com`. We use filter lists (step ❷) to classify a request (and its call stack) as tracking (*i.e.,* failed test case) and functional (*i.e.,* passed test case). We then calculate "tracking score" (Eq 1) for each code unit (*i.e.,* script or method) based on its participation in the call stack trace of tracking and functional requests, as shown in step ❸. The script `utag.js` initiates 132 tracking requests and

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

Table 1: Six different JS blocking configurations. ✗ represents an unblocked entity, and ✔ represents a blocked entity.

160 functional requests. In this script, method `loader` initiates 131 tracking requests and 1 functional request. Method `fireCORS` initiated 159 functional and 1 tracking request. Figure 2 demonstrates the calculation of the tracking score on the webpage in step ❹.

$$tracking\ score = \log\left(\frac{number\ of\ tracking\ requests}{number\ of\ functional\ requests}\right) \quad (1)$$

We classify code units that participate 100× times more in tracking than functional (*i.e.,* tracking score of > 2) as tracking. We classify code units that participate 100× times more in functional than tracking (*i.e.,* tracking score of < -2) as functional. This threshold is determined experimentally in prior work [26]. The code units that fall in neither category are classified as mixed. The localization step results in a list of tracking, functional, and mixed JS methods and scripts. In this example, script `utag.js` is classified mixed, method `fireCORS()` is functional, and method `loader()` is tracking.

3.1.3 *JS Blocking Impact Analysis.* To measure the impact of blocking JS code units, our custom-built Chrome extension loads every page from the 100K websites and blocks the associated tracking JS script or method from the list of labeled methods and scripts. It blocks the JS scripts from loading in the browser, similar to existing content blockers. To block a script method, it simply replaces the method name with `doNotExecuteMe` to redirect its invocations, as shown in Listing 1. Renaming the method name may cause a MethodNotFound exception that terminates the tracking thread in a webpage's JS execution as intended.

We conduct this experiment on the same 100K webpages in six parallel configurations shown in Table 1. These configurations are illustrated in the `livescore.com` case study and inspired by unique JS blocking strategies that are mostly in practice or proposed by prior work. Control configuration (`CTRL`) is used to localize JS code units (scripts and methods) using the aforementioned SBFL technique and for breakage comparison in the later subsection. In `ALL`, all scripts (tracking, mixed, and functional) are blocked to evaluate blanket JS blocking. This configuration represents NoScript, which blocks all scripts by default. In `TS`, tracking scripts are blocked to evaluate selective JS blocking. This configuration represents the majority of content blockers such as uBlock Origin [11], Brave [3], and Adblock Plus [1] that use EasyList [5] and EasyPrivacy [7]. In `MS`, mixed scripts are blocked to see its adverse consequence on functionality. In `TMS`, tracking and mixed scripts are blocked to evaluate tracking and mixed JS blocking. `TMS` is the optimum choice for content blockers in tracking prevention, but it risks functionality breakage, as shown in Section 2. Finally, we compare the results of `TMS` with `TM`, where we block tracking methods (all located in tracking and mixed scripts) to evaluate method-level JS blocking.

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

Table 2: Top JS methods found on the maximum number of websites in control configuration.

In `CTRL` configuration, we have websites that do not crash. However, website crashes and breakages may still occur in the blocking configurations due to blocking. Website breakage is a subjective metric that requires a visual inspection, which is not feasible on 100K webpages. Therefore, we discuss two metrics that are correlated with website breakage [49].

Tracking and Functional request count. Network requests fetch critical functional resources like scripts, images, and other media as well as JS scripts and images that perform tracking activity. We use the number of tracking and functional requests as a measure of tracking and functional activity on a webpage. We compare these numbers with the control configuration (`CTRL`) to get the missing requests, as shown in Figure 3-❶. This metric helps in collecting non-visual breakage clues. For example, we do not see any visual breakage on website *poshmark.ca* after blocking mixed script *sdk.js?hash=**. Instead, we observe two missing requests, one that sets the cookie and the other functional request that redirects the login button.

HTML of websites. We scan the HTML tags with `src` attributes on a webpage to estimate visible functional deterioration. These HTML tags include `<img>`, `<video>`, and `<iframe>`. Each tag has a source, `src`, attribute that specifies the URL of a resource file. We compare the missing tags in our experiments with the control configuration (`CTRL`), as shown in Figure 3-❷. Note that if the attribute of a missing URL belongs to the functional request in the control configuration (`CTRL`), then it is classified as functional breakage.

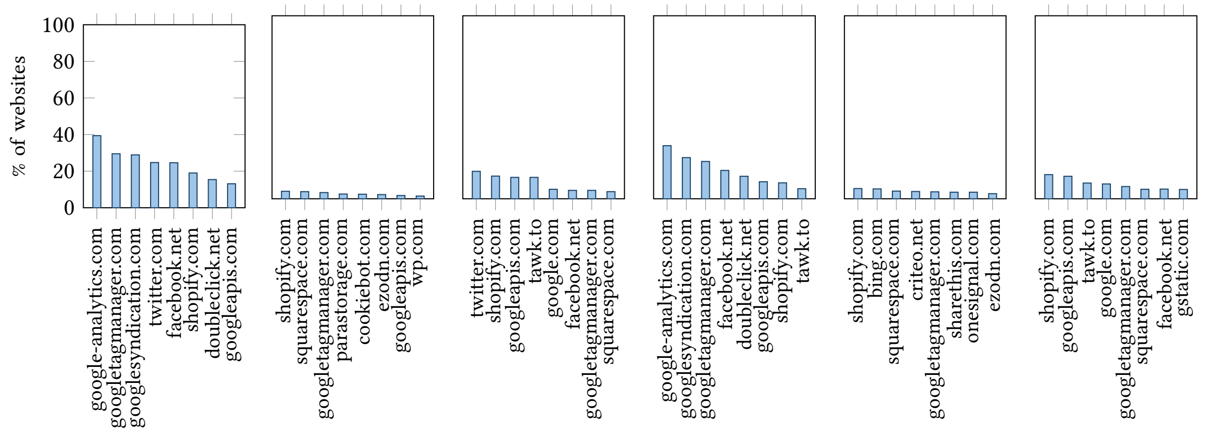
3.2 Phase II: Manual Inspection of JS Blocking

3.2.1 *Data Sampling.* Manually inspecting 100k websites is time-consuming and practically infeasible. We randomly sample 500 websites from the top 100K websites used in Phase I. We exclude duplicate websites and websites with the same second-level domains (SLD), but different top-level domains (TLD) *e.g.,*`google.com.uk` and `google.com`. We excluded a total of 117 websites and manually inspected 383 websites, which is a statistically significant sample size for 100K websites with ± 5% margin of error [16].

3.2.2 *Manual Inspection.* Two testers independently inspected 383 websites. Inspecting six configurations for each website manually and in parallel is prohibitively expensive. Therefore, we choose the three most important configurations *i.e.,* `CTRL` (for comparison), `TMS` (tracking and mixed JS blocking), and `TM` (method-level JS blocking). To assist inspection, our study platform launches three independent instances of Chrome (`CTRL`, `TMS`, and `TM` from Table 1) displayed adjacent to each other. Each tester spent at least 5 minutes inspecting

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

Table 3: Characteristics of the crawled dataset across six blocking configurations.



(a) **Script domains** in CTRL

(b) **Script domains in** ALL

(c) **Script domains in** TS

(d) **Script domains in** MS

(e) **Script domains in** TMS

(f) **Script domains in** TM

Figure 4: The top domains of request-initiating scripts across six blocking configurations. X-axis shows the top domains of the request-initiating scripts, and Y-axis shows the % of websites.

the three windows, scrolling each page end to end, and clicking on different webpage components. The two testers spent a total of 85 hours manually inspecting the websites and documenting their findings according to the following rubric. They report visual and functional differences in the following four categories and use a 3-level breakage scale (*i.e.,* no breakage, minor breakage, and major breakage). Any disagreements were discussed and resolved by consensus.

- **Navigation.** Website navigation contains lists of links to internal webpages. It typically consists of a menu or navigation bar that contains links to various sections of the website, such as the homepage, products or services, about us, and contact. Minor breakage involves non-functional navigation links, abnormal styling layouts, or missing icons. These issues can be frustrating for users and may make it difficult to navigate the website. Major breakage involves more serious issues, such as the navigation button not being operational or the navigation bar not appearing at all. This type of breakage can significantly impact the website's usability.
- **Single sign-on (SSO).** Website SSO allows users to sign in using credentials from services such as Google and Facebook. Minor breakage typically involves issues such as non-functional SSO services, unresponsive login buttons, or missing login options. For example, if the Google SSO service is not functioning, users may be unable to sign in to the website using their Google account. Major breakage involves more serious issues, such as the missing SSO service or the failure of all SSO options. This type of breakage can significantly impact the website's usability.
- **Appearance.** This category includes the appearances of media elements, the scrolling behavior of websites, and the HTML element. We exclude advertisements when inspecting appearance-based breakage. Minor breakage involves missing media resources, unstyled HTML, or jittery/unsmooth page scrolling experience. Major breakage involves all the media resources missing altogether or an unscrollable page.
- **Additional functionality.** Anything that does not fall into the mentioned categories is added to this category, such as dark mode, website settings, and chatbot. Minor breakage entails abnormal behavior or non-responsive feature. Major breakage includes page crashes and missing components.

3.3 Dataset

This section summarizes the characteristics of dataset crawled across six blocking configurations. Table 3 lists the total network requests and script-initiated requests in six configurations over 100K websites and the JS scripts and methods that initiate those requests. In control configuration (CTRL), out of 5.45 million requests, 22% of the requests are tracking, leaving the remaining 78% as functional. 34% of the total requests are initiated by JS scripts. In script-initiated requests, 52% are tracking, and the remaining 48%

are functional. These script-initiated requests are initiated by 366K JS methods inside 256K scripts.

Figure 4 shows the top domains of the scripts that initiate network requests. In control configuration (CTRL), 39% of websites initiate requests from the script served by google-analytics.com, 30% of websites initiate requests from the script served by googletagmanager.com, and 29% of websites initiate requests from the script served by googlesyndication.com.

Our baseline JS blocking configuration is ALL in which all tracking, mixed, and functional scripts are blocked. Note that a small number of scripts may still load in All if such scripts were previously not observed during the localization step in Section 3.1.2. When tracking JS scripts are blocked (TS configuration), the majority of tracking script domains disappear, including google-analytics.com. We observe a relatively lower occurrence of script domains in TMS than TM because TMS blocks all tracking and mixed scripts that include all tracking methods and some functional methods. Whereas in TM, only tracking methods are blocked. For example, due to the mixed nature of scripts from facebook.net, scripts from facebook.net appear in TM, but not in TMS.

```
wd = function(a, b, c, d) {
    var e = O.XMLHttpRequest;
    if (e) return 1;
    var g = new e;
    if (("withCredentials" in g)) return 1;
    a = a.replace(/^http:/, "https:");
    g.open("POST", a, 0);
    g.withCredentials = 0;
    g.setRequestHeader("Content-Type", "text/plain");
    g.onreadystatechange = function() {
      if (4 == g.readyState) {
        if (d && "text/plain" === g.getResponseHeader("
     Content-Type")) try {
          Ea(d, g.responseText, c)
        }
        catch (ca) {
          ge("xhr",
            "rsp"), c()
        } else c();
        g = null}};
    g.send(b);
    return 0}
  ...
ta = function(a) {
    var b = M.createElement("img");
    b.width = 1;
    b.height = 1;
    b.src = a;
    return b}
```

Listing 2: Methods wd and ta in analytics.js served by google-analytics.com are present on 38% and 25% of 100K websites, respectively.

Table 2 shows the top five request-initiating JS methods across 100k websites. Method wd in script analytics.js is served by google-analytics.com. It appears in 38% of the 100K websites where it sets up a request and its header using XMLHttpRequest [21] API, shown in Listing 2. Method ta in script analytics.js is served by google-analytics.com. It appears in 25% of the websites where it adds the <img> tag with a specific source given as a parameter to the function, shown in Listing 2. Both of these methods are classified as tracking in the localization step in Section 3.1.2.

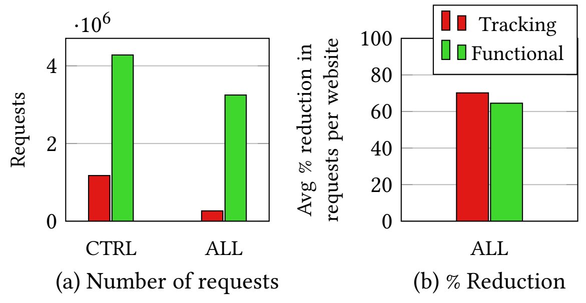


Figure 5: (a) compares the request count of control configuration with blanket JS blocking (ALL). (b) shows average % reduction in request per website for blanket JS blocking (ALL).

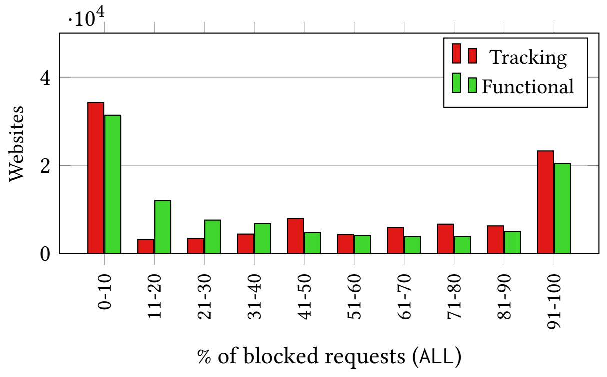


Figure 6: The % of blocked request in blanket JS blocking configuration (ALL). Low % of blocked functional requests and high % of blocked tracking requests are desirable.

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

Table 4: Missing HTML tags whose URLs are classified as functional in blanket JS blocking (ALL).

4 RESULTS

This section presents the results of our empirical investigation of different types of JS blocking listed in Table 1.

4.1 Phase I: Large-scale JS Blocking Analysis

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

(1) How resilient is website functionality against blanket JS blocking (ALL)?
(2) How effective is selective script-level JS blocking in tracking prevention and functionality preservation (TS and MS)?

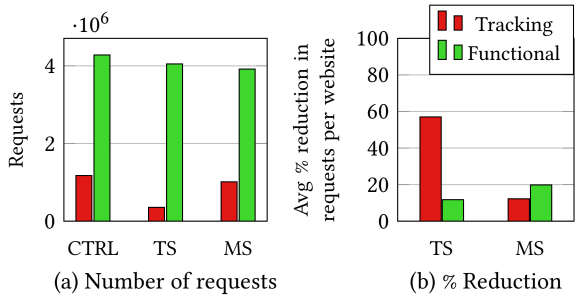


Figure 7: (a) compares the request count of control configuration with selective JS blocking (TS and MS). (b) shows average % reduction in request per website for selective JS blocking (TS and MS).

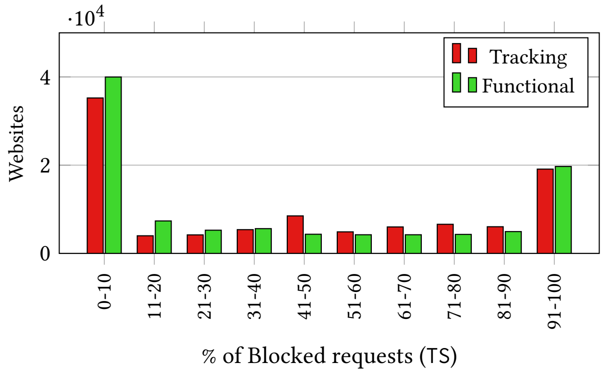


Figure 8: The % of blocked request in selective JS blocking configuration (TS). Low % of blocked functional requests and higher % of blocked tracking requests are desirable.

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

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

4.1.1 RQ1: Blanket JS Blocking. We first study the naive approach to JS blocking by blocking all JS scripts (ALL configuration in Table 1). Specifically, we block all 256K scripts on 100K webpages and compare the breakage metrics (*i.e.,* network request count and HTML resource count) with the control (CTRL). Given blanket JS blocking, we expect a sharp drop in the number of tracking or functional requests. Figure 5 (a) shows that 22% of functional requests and 76% of tracking requests remain after blocking all JS scripts (ALL). Note that a few requests are initiated by the scripts previously not captured in the localization step in Section 3.1.2 and hence, were not blocked in blanket JS blocking (ALL) configuration. Figure 5 (b) presents the average percentage of reduction in request count per webpage. On average, per webpage, the tracking and functional request count decrease by 70% and 65%, respectively. This shows that webpages today can retain one-third of functionality even with extreme blocking strategies. Another observation is that the tracking reduction per webpage is higher than functional reduction, which means that many webpages often sacrifice tracking but attempt to retain functionality.

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

Table 5: Missing HTML tags whose URLs are classified as functional in selective JS blocking (TS and MS) .

To map this behavior per webpage, we find the number of webpages with different levels of request reduction for both tracking and function. Figure 6 illustrates the result. We find that the majority of the webpages (57%) have either less than 10% request reduction or more than 90% request reduction in both tracking and functional. This result shows both (1) high resilience against tracking reduction and functional breakage due to anti-content blocking strategies such as loading resources by changing network endpoints [25, 50], and also (2) low resilience where blocked scripts are critical for a functioning webpage [26, 68]. Further inspection of HTML DOM elements reveal that 191K functional HTML tag sources are missing from 100K webpages when ALL scripts are blocked, reflecting severe functionality loss. Table 4 shows the breakdown of the category of these missing sources. In ALL configuration, 71K functional `<img>` tags, 21K functional `<iframe>` tags, and 100K functional `<script>` tags are missing.

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

4.1.2 RQ2: Effectiveness of Selective JS Blocking. Since Blanket JS blocking is ineffective, we study the effectiveness of selective JS blocking by blocking tracking scripts (TS configuration in Table 1). Later, we block mixed scripts (MS configuration in Table 1) to see its adverse effects on functionality.

Blocking Tracking Scripts. In this experiment, we block 93K tracking scripts (TS) from 256K JS scripts across 100K live webpages and investigate its impact on tracking mitigation and functional breakage. Figure 7 (a) reports that 95% of functional requests persist, whereas 30% of tracking requests manage to survive. Figure 7 (b) shows an average reduction in requests per webpage. In the case of TS, we observe a 57% reduction in tracking requests and an 11% reduction in functional requests per webpage on average. Measurement with HTML tag metric in Table 5 shows that blocking tracking JS scripts (TS) results in 46K missing functional sources across 100K webpages. In TS configuration, 13K functional `<img>` tags, 12K functional `<iframe>` tags, and 22K functional `<script>` tags are missing.

Blocking Mixed Scripts. In this experiment, we block only mixed JS scripts (MS). We expect a decrease in both functionality and tracking, as mixed scripts represent both. Figure 7 (a) visualizes these results. Overall, we see 86% of tracking and 92% of functional requests. This observation is consistent with other HTML tag metric in Table 5. In MS configuration, 20K functional `<img>` tags, 15K functional `<iframe>` tags, and 37K functional `<script>` tags are

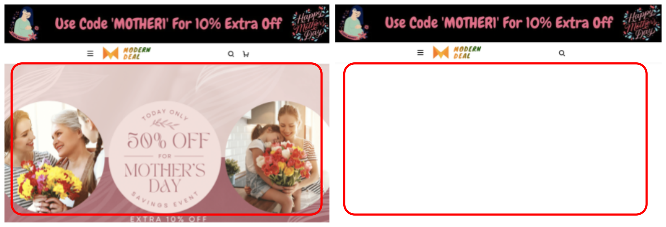

Use Code 'MOTHER1' For 10% Extra Off
Use Code 'MOTHER1' For 10% Extra Off
TODAY ONLY
50% OFF
FOR
MOTHER'S DAY
SAVINGS EVENT
EXTRA 10% OFF

Figure 9: Visual impact of blocking mixed JS script. The left side shows a normal website, whereas the right side shows a breakage due to blocking.

missing. Figure 9 show visual breakage on `pressl.co` due to blocking mixed JS scripts that eliminate tracking at the cost of critical functional breakage.

We further ask *Do all webpages react similarly when tracking scripts are blocked?* Our goal is to unfold the resilience of different webpages with blocked tracking scripts (TS). Figure 8 measures the distribution of webpages across different levels of functional breakage and tracking mitigation from blocking tracking scripts. 39K webpages experience less than 10% functional deterioration, and 35K webpages experience less than 10% tracking mitigation. The left of the bar chart represents webpages that heavily employ mixed scripts, making JS script blocking ineffective. 19K webpages are only left with greater than 90% functionality deterioration and tracking mitigation, representing the class of webpages relying less on mixing scripts and thus are susceptible to JS script blocking. Although JS script blocking is effective on a few webpages, it does not apply to a significant proportion of webpages that employ mixed scripts. Therefore, we must address the tracking behavior concealed in mixed scripts.

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

4.1.3 RQ3: Prevalence of Mixed Scripts. A trivial way for web developers and trackers to bypass filter lists is by mixing functional behavior with tracking in a single script. Privacy-enhancing content blockers, such as uBlock Origin, cannot afford to break the webpage and have no choice but to allow such scripts to load in the browser. To gather concrete evidence on the prevalence of this practice, we first conduct a longitudinal experiment on the frequency of mixed JS scripts over the past two years (2021 and 2022) on 100K webpages. In 2021, we crawled 100K webpages and classified the collected JS code using the SBFL-inspired approach from Section 3. We repeat the same experiment in 2022 on the same 100K webpages.

Figure 10 shows the result of the experiment. The x-axis represents the percentage of scripts that are mixed, ranging from 0 to 100 in 10 bins each of size 10. The y-axis represents the number of webpages in each bin. In 2021, 15% of webpages out of 100K have between 11% to 20% of scripts that were mixed. This number increases to 18% in 2022. Overall, in 2021, out of 220K JS scripts, 28K are mixed JS scripts, making it 12.8%, whereas, in 2022, 37.5K out of 256K JS scripts are mixed, making it 14.6%. There is 14%

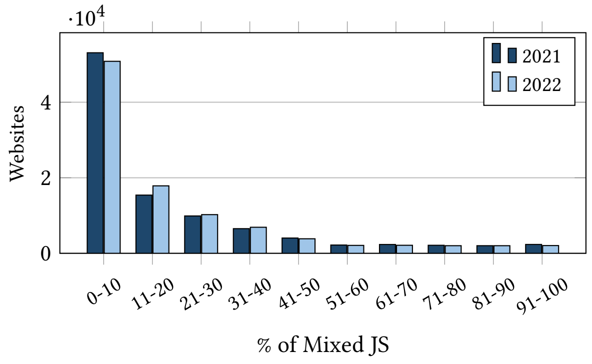


Figure 10: Comparison of % mixed JS scripts when tracking score is in [-2,2] for web corpus collected in 2021 and 2022.

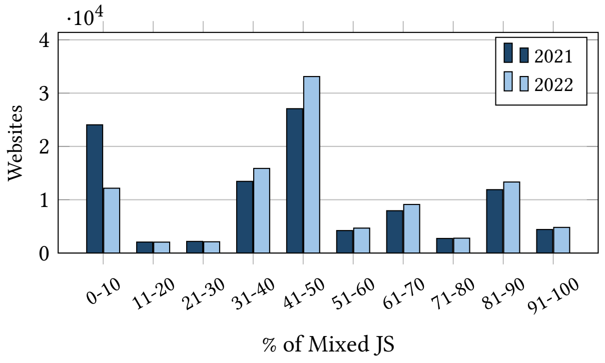


Figure 11: Comparison of % mixed JS scripts without any threshold on tracking scores for web corpus in 2021 and 2022.

increase in the number of websites employing mixed scripts over 100K websites, as compared to last year. For example, on the website `kixie.com`, we observe a new mixed JS script `20564323.js` in 2022, initiating HubSpot analytics code along with the functional code that redirects the `Try Kixie Free` button. We also find that the change in total script count corroborates the general belief that JS scripts across the web have increased marginally since 2021 [73].

While investigating selective JS blocking, we also find deterioration in the functionality when only tracking scripts are blocked (TS). Naturally, we ask *why does blocking tracking scripts (TS) result in functional deterioration?* We suspect that such an issue may arise due to the narrow threshold on SBFL's tracking score. JS code units (*i.e.,* scripts, methods) with > 2 score are annotated as purely tracking. Functional behavior in tracking scripts can also exist due to the dynamic nature of webpages. Between the tracking score measurement and blocking experiments, the script may have changed, or the webpage deliberately refactors the script slightly for reasons such as JS obfuscation [61, 67] or minification [59]. For better threshold selection, we must answer *what are the consequences of widening the tracking score threshold?* We conduct a brief sensitivity analysis on the tracking score's threshold. Figure 11 shows the new distribution when the threshold is set to maximum. We find that 46% of the webpages have more than 50% of their scripts mixed with at least one tracking or functional request, further reducing the applicability of JS script blocking and showing the extent of

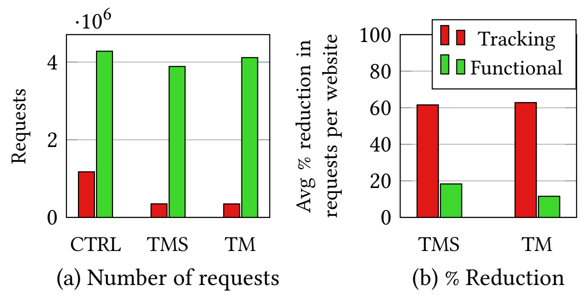


Figure 12: (a) compares the request count of control configuration with tracking and mixed (TMS) and method-level JS blocking (TM). (b) shows average % reduction in request per website for tracking and mixed (TMS) and method-level JS blocking (TM).

this problem. Our investigation in RQ3 highlights the following trade-off. We either sacrifice functionality when blocking mixed JS scripts or let go of privacy. If functional preservation is critical, we forego opportunities to block numerous tracking activities.

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

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

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

Figure 12 summarizes these results. Both baseline tracking and mixed JS blocking (TMS) and method-level JS blocking (TM) reduce the tracking requests by 71% and block on average 62% of the tracking requests per page. The two configurations cover most of the tracking requests among themselves, and blocking them will yield the same result. More surprisingly, we see an improvement in total functionality retention when blocking method-level (TM) *i.e.,* a 6% total improvement, whereas the average functional request breakage per page decreases by 7%. On evaluating HTML, JS method-level blocking(TM) retains approximately 2X more functional HTML tag sources, such as images and scripts, than blocking tracking and mixed JS scripts (TMS), as shown in Table 6. For example, in Figure 14, we visually inspect `deeretnanews.com` to find functional media breakage in TMS configuration that loads normally in TM configuration.

We further investigate *how much functional breakage does each webpage face with method-level blocking* (TM) *compared to the baseline* TMS*?* Figure 13 sheds more light on the functional request count

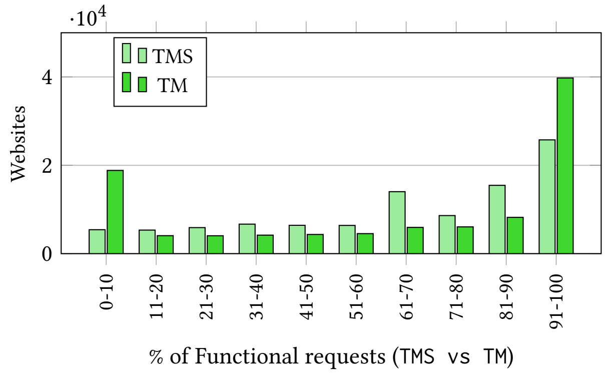


Figure 13: The % of functional requests in tracking and mixed (TMS) JS blocking and method-level JS blocking (TM). A higher % of functional requests is desirable.

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

Table 6: Missing HTML tags whose URLs are classified as functional in tracking and mixed (TMS) and method-level (TM) JS blocking.

between two blocking granularities. With method-level JS blocking (TM), 40% webpages have less than 10% functional breakage (preserved more than 90% functional requests). In comparison, tracking and mixed JS blocking (TMS) leads to around 25% of webpages in this category.

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

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

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

In Phase II, we perform a qualitative study to validate our quantitative findings with an in-depth visual inspection of sampled websites,

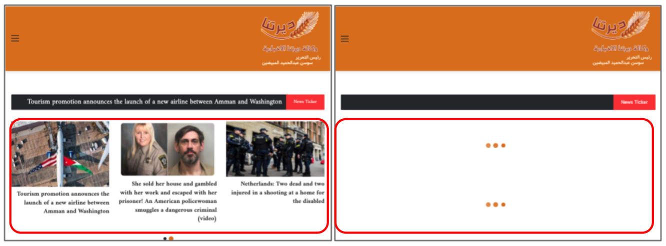


Figure 14: Image compares the functional breakage in tracking and mixed JS blocking (right) as compared to method-level JS blocking (left), which loads the website `deeretnanews.com` normally.

as described in Section 3.2. We seek to answer the following research questions:

(5) Does our manual inspection validate that method-level JS blocking is more effective than JS blocking?
(6) Is method-level JS blocking the most effective in minimizing breakage while preventing tracking?
(7) Can webpages withstand the removal of tracking methods?

4.2.1 RQ5: Validating the effectiveness of method-level JS blocking. Figure 15 and Figure 16 summarizes the results of investigating true functional breakage on 383 websites, measured according to four established metrics (*i.e.,* navigation, SSO, appearance, and others) and three levels of breakage. The X-axis represents the percentage of websites with functional breakage. Overall, there is an evident decline in the number of broken websites, for both major and minor breakage, when JS method-level blocking is used instead of tracking and mixed JS blocking. These results validate the findings of quantitative analysis in RQ4. In tracking and mixed JS blocking (`TMS`), 68 websites have minor breakage and 118 websites have major breakage, whereas, in method-level JS blocking, 45 websites have minor breakage, and 29 websites have major breakage. Most of the breakages were observed in additional feature categories, comprising broken widgets (*e.g.,* chatbots and feedback) and malfunctioning home buttons.

`Washingtonpost.com` (ranked 9th in news and media publisher category in USA [19]) is one of the 383 sampled websites. It suffers a crash (a major breakage) in tracking and mixed scripts JS blocking (`TMS`). On the contrary, the website is completely functional and tracking-free at method-level JS blocking (`TM`). Similarly, on `tenki.jp` (ranked 4th in the streaming and online TV category in Japan [17]), manual inspection reveals a missing Twitter widget and a Twitter button in tracking and mixed scripts JS blocking (`TMS`). These breakages are documented as minor breakages. However, in method-level JS blocking (`TM`), all tracking advertisements are blocked, and both the button and widget appear correctly and are functional, similar to the control experiment (`CTRL`). The website `ndtv.com` (rank 5th in the news and media category in India [48]) renders multiple advertisements in the control experiment (`CTRL`). Website completely crashes in tracking and mixed scripts JS blocking (`TMS`), whereas, in method-level JS blocking, it renders normally without any advertisement.

We also argue that minor improvements can make a difference in many websites. For example, website `gamestop.com` (rank 9th in the gaming category in USA [10]) shows 37.5% breakage in tracking and mixed scripts JS blocking (`TMS`) whereas shows only 12.5% breakage at method-level JS blocking(`TM`). At `TMS`, we see unexpected white spaces on the top of the website, a minor breakage in the appearance category. The webpage's home button also causes the website to crash, a major breakage recorded in additional functionality. However, in `TM`, we only see an unexpected white space on the website, a minor breakage in the appearance category. These results also affirm that the breakage metrics (network request and media resources) used in Phase I are effective measures of breakage.

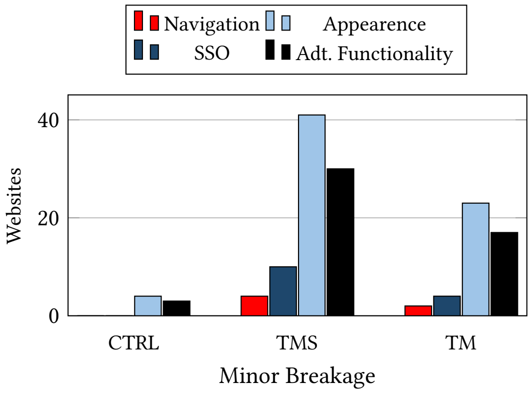


Figure 15: Comparison of "minor" breakage in tracking and mixed JS blocking (`TMS`) vs method-level JS blocking (`TM`) among 383 sampled websites.

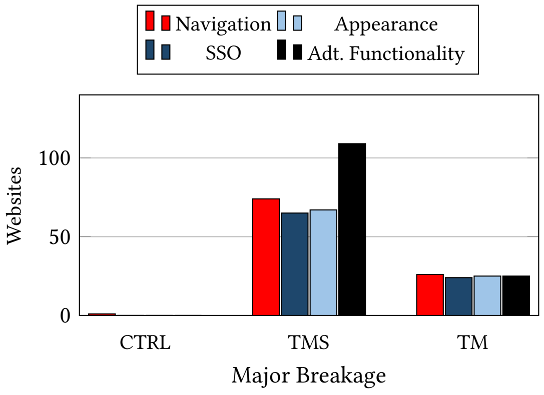


Figure 16: Comparison of "major" breakage in tracking and mixed JS blocking (`TMS`) vs method-level JS blocking (`TM`) among 383 sampled websites.

4.2.2 RQ6: Is method-level blocking most effective in reducing breakage and eliminating tracking? Although method-level JS blocking (`TM`) performs significantly better than tracking and mixed JS blocking (`TMS`), there are cases where we observe little or no improvement. This is mainly because of 6% methods still show mixed behavior *i.e.,* include tracking and functional code. `Elpais.com` (currently ranks 2nd in the news and media publisher category in Spain [8]) fails to load a single resource in tracking and mixed scripts JS blocking (`TMS`). However, in method-level JS blocking (`TM`), it causes the navigation bar to be unresponsive, a minor breakage due to the mixed method `e.loadInternal` in script `provider.hlsjs.js`.

4.2.3 RQ7: Can webpages sustain simply removing the tracking JS method? On 100K webpages, we have found that webpages in their vanilla form have 1.32 severe errors on average. Severe error refers to three main compile-time errors in JavaScript: syntax errors, runtime errors, and logical errors. Errors are common in JS and do not always impact functionality. Compared to other software, webpages can withstand many runtime issues, such as network error, JS script not found, and JS script syntax errors that can arise from diverse host environments. In our experiments, we block JS tracking method by simply renaming the method, which may lead to MethodNotFound error. Replacing a method name and redirecting its invocation may generate additional errors. However, such errors do not affect the website's functionality, as they only terminate the tracking-inducing thread in the JS process.

5 DISCUSSION

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

JS blocking at finer granularity. While blocking JS tracking methods is beneficial, we still observe that 5.5% webpages with some levels of tracking activity and functionality breakage. These webpages contain method(s) that (1) implement both tracking and functionality or (2) are used by tracking and functional code for downstream activity (*e.g.,* initiating a network request). We foresee better separation at a finer granularity. In the future, we propose applying dynamic program slicing [24, 46, 75] to separate tracking statements from functional statements. For inseparable code, we propose dynamic invariant detection [35, 51] to construct program variable profiles for tracking and functional behaviors. Program invariants for tracking can be used as an automated guard to prohibit tracking execution.

Dynamic nature of JS. We find that a number of scripts use dynamic features such as `eval()` and anonymous functions [59, 65]. A number of scripts also employ JS minification and obfuscation techniques that produce code that is uninterpretable manually [61, 67]. Such practices further motivate the use of advanced dynamic program analysis techniques for tracking code identification and removal.

JS dataflow analysis. In this work, we captured the stack trace of a tracking or functional network request and then annotate the script method at the top of the stack. By focusing on request-initiator code units, we may miss opportunities to trace back to the source of the tracking behavior inside the nested JS codebase. Finding such a location may offer better opportunities to preserve functionality as the request-initiator method or script may simply be a "gateway" for all network requests. In addition to the call stack, we can also leverage the dataflow graph of the JS codebase to perform a richer analysis of a webpage's execution. For example, in Listing 3, the stack trace inside the method `B` does not contain the parameter `C`. Since the method `B` depends on parameter `C`, the identification technique may not understand the entire context when `B()` is called. We recommend capturing such rich execution traces with calling contexts and a complete data flow graph to understand better the flow of information through the nested code and how it influences the execution behavior, tracking, or functional. We anticipate that such traces can help identify better locations (*e.g.,* non request-initiator methods) to alleviate tracking while preserving functionality.

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

Listing 3: Call stack does not show complete dataflow.

Performance impact of JS blocking. Although we do not consider performance in our analysis, our focus is to minimize tracking without comprising functionality. Recent works [27, 28] show that the removal of non-critical components of JS code can significantly reduce page load times. Similarly, removing the tracking JS code may reduce the performance overhead along with functionality preservation.

Other future research directions. We plan to conduct an investigation into more meaningful and semantics-aware tracking code identification. Our key observation is that finding a tracking code unit in webpages has striking similarities with fault localization. Even a simple faulty code localization method such as SBFL showed promising results towards functionality-preserving JS blocking. On the code refactoring front, our observation of 100K vanilla live websites reveals that today's webpages can withstand severe errors. Therefore, we expect that slightly unsafe code refactoring techniques to remove the tracking code may be promising in effectively preserving functionality while preventing tracking.

Future tracking code identification techniques can greatly benefit from recent advances in automated debugging and fault localization [41, 55]. For example, given filter list as a test oracle, we can adapt search-based debugging approaches to perform a systematic search on JS code and precisely isolate the tracking and functional code units [58]. Similarly, the completeness of static code dependency analysis (*e.g.,* reachability analysis) can complement the soundness of dynamic analysis (*e.g.,* call graph) to improve the precision of tracking code localization.

Code clone detection is an active area of research, with many advanced techniques available for traditional software [31]. Given annotated JS code units, code clone detection techniques can identify similar code on webpages to find the presence of tracking code. Once a JS code clone is correctly detected, we can leverage supervised learning [38, 66] to extract valuable features, both semantic and syntactic, for accurate tracking code localization. If such an accurate model is available, a JS blocker can detect tracking JS code units in real time and block them before loading the website.

Similar to training a classification model, one possible direction is to create a taxonomy of tracking code's signature, similar to the ones in malware detection [37, 40, 77], and find a match with a webpage's JS entity at page load. However, page load times are critical in the web domain, refraining from any computationally expensive operation. Using fingerprints to locate tracking code at page load is a lightweight process that can easily be performed at page load time without a noticeable slowdown.

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

6 LIMITATIONS

Internal validity. Our analysis in Section 4 relies on the correlation between a JS blocking strategy and the webpage's behavior in terms of network requests and resource loads. However, other confounding factors may impact the webpage's behavior. For instance, some webpages fetch different number and type of resources for each visit due to the inherent dynamism. For our experiments, requests monitored in one experiment may not be triggered in another experiment. Other factors include behavior change due to environment (*i.e.,* browser and host OS), visit time, and location. We minimize internal validity threats by keeping the environments consistent across different blocking configurations *i.e.,* same location, browser, and stateless crawls.

External validity. We conducted our experiments using the Chrome browser with a Chrome-based extension. Extensions on other browsers have different permissions and have access to a varying set of information about a webpage's behavior. While our choice of using the Chrome browser minimizes external validity threats, it is possible that our results may not fully generalize to JS method-level blocking on other browsers. Similarly, our annotation relies on previously observed tracking behavior captured in filter lists. Its effectiveness may be limited for unseen JS. To minimize this issue, we use the two most actively maintained filter lists for annotation.

Construct validity. We collect the website's data at page load time and do not capture other events triggered by the user interactions such as scrolling and clicking. This is a general limitation of dynamic analysis that can be mitigated by using a forced execution framework [45].

7 RELATED WORK

Smith et al. [68] and Amjad et al. [26] identify tracking code regions in the JS scripts of websites. Sugarcoat [68] dynamically captures the call graphs of web APIs and uses it to determine the call site in JS code that tries to access the user-sensitive information from the local storage, which is unanimously considered as tracking behavior. They replace these identified call sites with the surrogate JS code that mitigates the information access but preserves functionality. This process helps create surrogate scripts for exception rules in the filter list. SugarCoat requires excessive manual effort by a domain expert to identify the tracking call sites in the JS code. Due to this limitation, our empirical study could not validate SugarCoat's effectiveness. Amjad et al. [26] introduce a hierarchical approach to annotate web entities (domain, hostname, script, and method) to precisely isolate the code responsible for tracking behavior. They dynamically collect the call stack information for tracking behavior and isolate the entities based on their participation in invoking it. We adapt their approach for web corpus collection and extend it to enable real-time code blocking and capture additional information.

Modern websites extensively use third-party JS scripts that may access potentially sensitive information [44, 52, 69, 70]. Tran et al. [70] develop a principal-based tainting approach that dynamically analyzes the JS libraries to identify the underlying privacy violations. They tag each compiled JS library at run-time and observe its suspicious behavior with the author-defined principles *i.e.,* a set of permissions that should not be violated. Similarly, Staicu et al. [69] introduce an automated approach that collects taint specifications of JS libraries and identifies behaviors that lead to security vulnerabilities. These approaches work at the granularity of JS libraries, which, as we find, is insufficient for preserving functionality. Moreover, these works use taint analysis that incurs prohibitively high-performance overhead and can not efficiently work in the browser in real-time. Prior work's findings on the challenges from JS dynamism resonate with our findings. Jueckstock et al. [44] present a lightweight dynamic analysis tool using chrome V8 to identify untrustworthy JS scripts. It logs function calls and storage access during JS execution to identify suspicious code.

The limitations of identifying tracking code share similarities with prior research on fault localization. For example, spectra-based fault localization (SBFL) [33, 42, 62, 64, 74] leverages the statement coverage using the set of passing and failing test cases to localize the statement that is most likely to induce a test failure. Similarly, Bela et al. [71] and Laghari et al. [48] present an approach that uses the frequency of method occurrence in the call stack of failing test cases for localizing the faulty methods. A method that appears more in the call stack of failing test cases is more likely to be faulty. Abreu et al. [23] conducted an empirical study on the accuracy of these SBFL techniques and highlighted that these approaches are independent of the quality of the test oracle. Crowdsourced blocklists [5–7, 9] are the authoritative source of labels for requests and are adequate to detect tracking behavior.

Websites heavily rely on JS libraries containing significant dead code that is unused or unreachable, posing a noticeable impact on the website's performance. Kupoluyi et al. [47] highlight that popular websites have 70% unused functions, and their elimination can speed up the page load by 30%. Recent works [27, 28] have further explored non-critical regions in JS libraries and the performance overhead caused by them. Zaki et al. [28] propose on rule-based classification techniques to identify and replace non-critical regions in JS using pre-define code patterns, achieving a 50% reduction in page load time. Towards the same goal, Chaqfeh et al. [27] develop a tool that helps developers in eliminating JS elements by visually inspecting them and shows 90% improvement in Google's lighthouse performance score. Similarly, Vazquez et al. [72] proposed a technique to decompose bundles JS code in a website, reducing code size by 26%. Our findings in this study are equally beneficial to the research on improving website performance and energy consumption that specifically adopt functionality-preserving code debloating approaches.

8 CONCLUSION

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

ACKNOWLEDGMENTS

This work is partly supported by the National Science Foundation under grant numbers 2106420, 2103038, 2138139, 2103439, and 2051592. We want to thank the anonymous shepherd and reviewers for their constructive feedback that helped improve the work.

REFERENCES

[1] 2023. ABP anti-circumvention filter list. https://github.com/abp-filters/abp-filters-anti-cv.

[2] 2023. Ad blockers usage and demographic statistics in 2022. https://backlinko.com/ad-blockers-users

[3] 2023. Brave Browser. https://brave.com/.

[4] 2023. Disconnect Tracking protection lists. https://disconnect.me/trackerprotection.

[5] 2023. EasyList. https://easylist.to/easylist/easylist.txt.

[6] 2023. Easylist Cookie List. https://secure.fanboy.co.nz/fanboy-cookiemonster.txt.

[7] 2023. EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt.

[8] 2023. eplais. https://www.similarweb.com/website/elpais.com/.

[9] 2023. Fanboy's Social Blocking List. https://easylist.to/easylist/fanboy-social.txt.

[10] 2023. gamestop. https://www.similarweb.com/website/gamestop.com/.

[11] 2023. gorhill/uBlock: uBlock Origin - An efficient blocker for Chromium and Firefox. Fast and lean. https://github.com/gorhill/uBlock.

[12] 2023. How to enable or disable JavaScript in a browser? https://www.computerhope.com/issues/ch000891.htm.

[13] 2023. HTML5 The noscript element. https://www.w3.org/TR/2011/WD-html5-author-20110809/the-noscript-element.html.

[14] 2023. livescore. https://www.similarweb.com/top-websites/category/sports/soccer/.

[15] 2023. NoScript. https://noscript.net/.

[16] 2023. Sample size calculator. https://www.surveymonkey.com/mp/sample-size-calculator/.

[17] 2023. tenki. https://www.similarweb.com/website/tenki.jp/.

[18] 2023. uBlock Origin web accessible resources. https://github.com/gorhill/uBlock/tree/master/src/web_accessible_resources.

[19] 2023. washingtonpost. https://www.similarweb.com/website/washingtonpost.com/.

[20] 2023. webalmnac:onloac. https://perma.cc/W9ZW-DP2D.

[21] 2023. XMLHttpRequest. https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest.

[22] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software* 82, 11 (2009), 1780–1792.

[23] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques*.

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

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

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

[27] Moumena Chaqfeh, Russell Coke, Jacinta Hu, Waleed Hashmi, Lakshmi Subramanian, Talal Rahwan, and Yasir Zaki. 2022. JSAnalyzer: A Web Developer Tool for Simplifying Mobile Web Pages Through Non-Critical JavaScript Elimination. *ACM Transactions on the Web* (2022).

[28] Moumena Chaqfeh, Yasir Zaki, Jacinta Hu, and Lakshmi Subramanian. 2020. JSCleaner: De-Cluttering Mobile Webpages Through JavaScript Cleanup. In *Proceedings of The Web Conference 2020*.

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

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

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

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

[33] Higor A de Souza, Marcos L Chaim, and Fabio Kon. 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347* (2016).

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

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

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

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

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

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

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

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

[42] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*.

[43] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*.

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

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

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

[47] Jesutofunmi Kupoluyi, Moumena Chaqfeh, Matteo Varvello, Russell Coke, Waleed Hashmi, Lakshmi Subramanian, and Yasir Zaki. 2022. Muzeel: Assessing the Impact of JavaScript Dead Code Elimination on Mobile Web Performance. In *Proceedings of the 22nd ACM Internet Measurement Conference*.

[48] Gulsher Laghari, Alessandro Murgia, and Serge Demeyer. 2015. Localising Faults in Test Execution Traces. In *Proceedings of the 14th International Workshop on Principles of Software Evolution (IWPSE 2015)*.

[49] Hieu Le, Salma Elmalaki, Athina Markopoulou, and Zubair Shafiq. 2023. AutoFR: Automated Filter Rule Generation for Adblocking. *USENIX Security Symposium* (2023).

[50] Hieu Le, Athina Markopoulou, and Zubair Shafiq. 2021. CV-Inspector: Towards Automating Detection of Adblock Circumvention. In *Network and Distributed System Security Symposium (NDSS)*.

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

[52] Song Li. 2022. *Towards Making JavaScript Applications Secure and Private.* Ph. D. Dissertation. Johns Hopkins University.

[53] Matthew Malloy, Mark McNamara, Aaron Cahn, and Paul Barford. 2016. Ad Blockers: Global Prevalence and Impact. In *Proceedings of the 2016 Internet Measurement Conference (IMC '16)*.

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

[55] Wes Masri. 2015. Automated Fault Localization: Advances and Challenges. *Advances in Computers* 99 (2015), 103–156.

[56] Jonathan Mayer and John Mitchell. 2012. Third-party web tracking: Policy and technology. In *IEEE symposium on security and privacy*.

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

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

[59] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 569–580.

[60] Shaoor Munir, Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. 2022. CookieGraph: Measuring and Countering First-Party Tracking Cookies. *arXiv:2208.12370* (2022).

[61] Ray Ngan, Surya Konkimalla, and Zubair Shafiq. 2022. Nowhere to Hide: Detecting Obfuscated Fingerprinting Scripts. *arXiv preprint arXiv:2206.13599* (2022).

[62] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*.

[63] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2018. Tranco: A research-oriented top sites ranking hardened against manipulation. *https://tranco-list.eu/assets/tranco-ndss19.pdf* (2018).

[64] Qusay Idrees Sarhan and Árpád Beszédes. 2022. A survey of challenges in spectrum-based software fault localization. *IEEE Access* 10 (2022), 10618–10639.

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

[66] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. 2022. WebGraph: Capturing Advertising and Tracking Information Flows for Robust Blocking. In *USENIX Security Symposium*.

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

[68] Michael Smith, Pete Snyder, Benjamin Livshits, and Deian Stefan. 2021. SugarCoat: Programmatically Generating Privacy-Preserving, Web-Compatible Resource Replacements for Content Blocking. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2844–2857.

[69] Cristian-Alexandru Staicu, Martin Toldam Torp, Max Schäfer, Anders Møller, and Michael Pradel. 2020. Extracting Taint Specifications for JavaScript Libraries. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*.

[70] Minh Tran, Xinshu Dong, Zhenkai Liang, and Xuxian Jiang. 2012. Tracking the Trackers: Fast and Scalable Dynamic Analysis of Web Content for Privacy Violations. In *Applied Cryptography and Network Security*, Feng Bao, Pierangela Samarati, and Jianying Zhou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 418–435.

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

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

[73] Jeremy Wagner. 2022. JavaScript: 2022: Web Almanac by HTTP Archive. https://almanac.httparchive.org/en/2022/javascript

[74] Ratnadira Widyasari, Gede Artha Azriadi Prana, Stefanus A Haryono, Yuan Tian, Hafil Noer Zachiary, and David Lo. 2022. XAI4FL: Enhancing Spectrum-Based Fault Localization with Explainable Artificial Intelligence. In *2022 IEEE/ACM 30th International Conference on Program Comprehension (ICPC)*. IEEE, 499–510.

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

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

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