Blocking Tracking JavaScript at the Function Granularity

Abdul Haddi Amjad
hadiamjad@vt.edu
Virginia Tech
Blacksburg, USA

Shaoor Munir
smunir@ucdavis.edu
University of California
Davis, USA

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

Muhammad Ali Gulzar
gulzar@cs.vt.edu
Virginia Tech
Blacksburg, USA

Abstract

Modern websites extensively rely on JavaScript to implement both functionality and tracking. Existing privacy-enhancing content blocking tools struggle against mixed scripts, which simultaneously implement both functionality and tracking. Blocking such scripts would break functionality, and not blocking them would allow tracking. We propose NoT.js, a fine-grained JavaScript blocking tool that operates at the function-level granularity. NoT.js's strengths lie in analyzing the dynamic execution context, including the call stack and calling context of each JavaScript function, and then encoding this context to build a rich graph representation. NoT.js trains a supervised machine learning classifier on a webpage's graph representation to first detect tracking at the function-level and then automatically generates surrogate scripts that preserve functionality while removing tracking. Our evaluation of NoT.js on the top-10K websites demonstrates that it achieves high precision (94%) and recall (98%) in detecting tracking functions, outperforming the state-of-the-art while being robust against off-the-shelf JavaScript obfuscation. Fine-grained detection of tracking functions allows NoT.js to automatically generate surrogate scripts, which our evaluation shows that successfully remove tracking functions without causing major breakage. Our deployment of NoT.js shows that mixed scripts are present on 62.3% of the top-10K websites, with 70.6% of the mixed scripts being third-party that engage in tracking activities such as cookie ghostwriting.

CCS Concepts

• **Security and privacy → Intrusion/anomaly detection and malware mitigation**; • **Information systems → Online advertising**; • **Software and its engineering → Software defect analysis**;

Keywords

privacy, web, software engineering, code refactoring

ACM Reference Format:
Abdul Haddi Amjad, Shaoor Munir, Zubair Shafiq, and Muhammad Ali Gulzar. 2024. Blocking Tracking JavaScript at the Function Granularity. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670329

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670329

1 Introduction

Modern websites extensively rely on third-party JavaScript (JS) to implement tracking (*e.g.,* advertising, analytics) and non-tracking (*e.g.,* functional content) [14, 58, 71]. In fact, 81% of all tracking requests are triggered by JS [54]. Privacy-enhancing content blockers aim to block tracking JS without breaking the legitimate website functionality. As the arms race has escalated with trackers attempting to evade blocking, the state-of-the-art content blocking approaches face the following key challenges in blocking tracking JS [13, 59, 67, 81, 82].

First, the state-of-the-art content blocking approaches do not capture the context needed to effectively detect tracking behavior implemented in a script. While existing approaches (*e.g.,* [53, 59, 81]) capture the script that initiates a network request, they do not capture the sequence of functions that led to the network request. Such code-level interactions leading to network requests can provide rich clues to a web page's intent. The lack of sufficient consideration of the execution context associated with tracking behavior fundamentally limits the effectiveness of existing approaches.

Second, the state-of-the-art content blocking approaches struggle when tracking and non-tracking resources are lumped together [53, 80]. When both tracking and non-tracking resources are served from the same network location, filter lists [8, 9] or even ML approaches [67, 81] struggle. This issue is further exacerbated by the use of URL encryption [11, 88]. This means that URL-based approaches are unable to discern between tracking and non-tracking resources. Similarly, JS signatures [59] are ineffective when tracking and non-tracking code is combined in the same script. This is fundamentally a granularity issue – *i.e.,* specific functions within mixed scripts are responsible for tracking [53].

Third, the state-of-the-art content blocking approaches rely on the laborious process of manually curated filter lists. Filter lists are used to block tracking network requests [8, 9] and stop the execution of tracking code [6, 41]. TrackerSift [53] relies on manually curated filter lists to detect mixed tracking scripts. uBlock Origin employs manually refactored replacements (aka scriptlets) to handle mixed tracking scripts [39, 40]. The reliance on manual curation fundamentally hinders scalability.

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

We evaluate the effectiveness, robustness, and usability of NoT.js on the top-10K Tranco websites [69]. Our evaluation shows that NoT.js accurately detects tracking JS functions with 94% precision and 98% recall, outperforming the state-of-the-art by up to 40%. NoT.js's contributions in incorporating dynamic execution context account for 29% improvement in F1-score. Against a number of JS obfuscation techniques, such as control flow flattening, dead code injection, functionality map, and bundling, NoT.js remains fairly robust – its F1-score decreases by only 4%. NoT.js's automatically generated surrogate scripts block 84% of the tracking JS function calls without causing any breakage on 92% of the websites.

We deploy NoT.js to study the tracking functions in mixed scripts, discovering that 62% of the top-10K websites have at least one mixed script. We find that the tracking functions are served in the mixed scripts from more than eight thousand unique domains, including those belonging to tag management services, advertisers, and content delivery networks (CDNs). Notably, among these mixed scripts, a significant 70.6% are third-party scripts that engage in tracking activities such as cookie ghostwriting [74, 79].

Our key contributions are summarized as follows:

(1) We propose NoT.js, a machine learning-based approach to detect and block tracking at the JS function-level granularity. We show that NoT.js outperforms the state-of-the-art in terms of accuracy and is robust against evasion.
(2) We implement NoT.js as a browser extension and show that it can be used to automatically generate surrogate scripts by neutralizing tracking function calls. We show that these surrogate scripts can be injected into a website to reliably mitigate tracking at its origin without breaking website functionality.
(3) We deploy NoT.js on the top-10K websites to measure the prevalence of tracking functions in the mixed scripts. We show that these mixed scripts are commonly served by tag management services, advertisers, and content delivery networks (CDNs). A majority of these mixed scripts are third-party, actively engaged in tracking activities such as cookie ghostwriting.

Data Availability: Our code and data are available at https://github.com/hadiamjad/Not.js.

Extended Version: The extended version of the paper is available on arXiv (https://arxiv.org/pdf/2405.18385).

2 Background & Related Work

Existing countermeasures against web tracking. Privacy enhancing content blockers, such as uBlock Origin [13] and Brave [57], primarily rely on manually curated filter lists [8–10, 12, 57] to block tracking network requests at the client side. These filter lists contain URL- or domain-based rules to determine whether a network request should be allowed or blocked. Prior research shows that trackers are able to evade filter lists by changing their network location, such as the URL or domain [4, 70, 73, 87]. These evasions necessitate manual updates to the filter lists to accommodate the new network locations, leading to a perpetual arms race between the maintainers of filter lists and trackers [1–3, 5, 66, 84, 89].

To mitigate this issue, recent approaches, such as AdGraph [67], WebGraph [81], and PageGraph [82], use machine learning to automatically generate filter list rules, aiming at the identification of tracking network requests. These approaches adopt a graph-based representation of the webpage execution to classify network requests. Both WebGraph and PageGraph enhance AdGraph by incorporating additional features into their graph representation. For instance, WebGraph additionally captures storage accesses (*e.g.,* cookie read/write), exhibiting superior performance against URL/domain-based evasion. While these approaches detect and block tracking network requests, their instrumentation is limited to the interactions between the initiator script and the tracking request, making it ineffective in pinpointing the origin of tracking activity within mixed resources.

Script-level restrictions against web tracking. Trackers/advertisers evade content blockers by utilizing a common network location to serve both tracking and non-tracking resources. For instance, trackers have started using Content Delivery Networks (CDNs) or engage in CNAME cloaking [61, 63, 64] to serve both tracking and non-tracking requests from a common network location. Content blockers are thus presented with a dilemma: either block the network request, potentially disrupt legitimate website functionality, or permit the network request, consequently letting go of tracking/advertising. To address this issue, a few content blockers such as uBlock Origin have introduced support for "scriptlets", which are custom JS snippets injected at runtime to substitute the code that initiates tracking/advertising network requests [40]. This scriptlet strategy is effective in countering tracking, even when originating from the same network location as non-tracking resources, as it eliminates tracking at its origin — well before a tracking network request is initiated. However, akin to filter lists, scriptlets are manually crafted and, therefore, challenging to scale across the entire web. At present, Brave Browser and uBlock Origin are capable of blocking a mere 27 scripts[1], while popular filter lists comprise over 6,000 exception rules designed to enable functionality-critical tracking scripts [85]. Consequently, tens of thousands of privacy-invasive scripts remain unblocked.

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

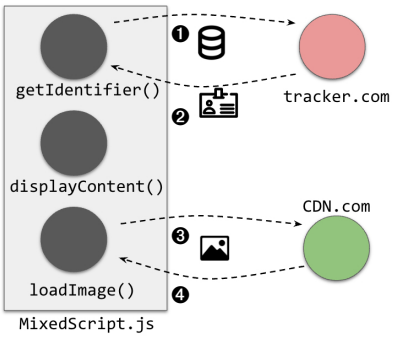


Figure 1: Marker ❶ signifies a tracking request to `tracker.com`, returning a user ID (❷) for activity monitoring, while ❸ indicates a non-tracking CDN request retrieving an image (❹).

Static or dynamic program analysis approaches have also been proposed to detect tracking/advertising JS code. Ikram *et al.* [65] employed machine learning to analyze syntactical and structural aspects of JS code, aiming to classify tracking scripts. However, their static analysis approach remains vulnerable to basic JS obfuscation techniques. Chen *et al.* [59] devised event-loop-based signatures based on dynamic code execution to detect tracking scripts. They found that some trackers bundle tracking and non-tracking code within a single script, posing a similar challenge when tracking and non-tracking resources are served from the same network location.

The increasing prevalence of mixed scripts, particularly those facilitated through bundling, poses a fundamental challenge to privacy-enhancing content blocking [7]. According to the Web Almanac, bundling is already a common practice on top-ranked websites; specifically, 17% of the top-1K websites employ the Webpack JS bundler [14]. Furthermore, there has been a 5× increase in the downloads of prominent bundlers such as Webpack over the past five years [44]. Amjad *et al.* showed that the prevalence of mixed scripts on top-100K websites increased from 12.8% in 2021 [53] to 14.6% in 2022 [54].

Function-level restrictions against web tracking. Prior research has proposed a function-level characterization of JS code. Smith *et al.* proposed SugarCoat [83] to systematically generate substitutes for scripts involved in tracking activities. SugarCoat relies on existing filter lists to identify tracking scripts for rewriting. It is ineffective against false negatives present in these filter lists. It uses PageGraph [82] to pinpoint the code locations where scripts access the privacy-sensitive data, such as `document.cookie` and `localStorage`. Subsequently, these code locations, which can include JS functions, are replaced by benign mock implementations that are manually generated by developers. However, to date, only six mock implementations [2] are created for the designated APIs through developer assistance. This reliance on manual effort from developers limits SugarCoat's applicability to a large number of tracking scripts.

Amjad *et al.* [53, 54] proposed TrackerSift, an approach to untangle mixed JS code down to the granularity of individual functions. Their approach is able to separate out 98% of all requests generated by tracking and non-tracking resources. Their empirical analysis determines that targeting specific JS functions reduces breakage by 3.8× as compared to blocking entire scripts. Nevertheless, TrackerSift's efficacy solely relies on existing filter lists for the differentiation of tracking and non-tracking resources, thus limiting its usefulness.

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

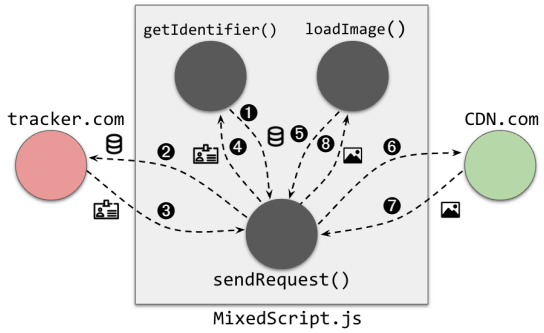


Figure 2: Marker ❶ signifies a call from `getIdentifier` to `sendRequest`, starting a tracking request to tracker ❷, and the identifier received at ❸ is returned to the originating function at ❹. Concurrently, ❺ represents a call from `loadImage` to `sendRequest`, triggering a non-tracking request ❻ to a CDN, with the image displayed at ❼ and returned to `loadImage` at ❽.

3 Threat Model

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

Definitions. We use the term `initiator function` to describe a JS function that directly initiates a network request. This function is always at the top of the call stack when we analyze a network request. In Figure 1, both `getIdentifier` and `loadImage` are examples of initiator functions. Next, we use the term `gateway function` to describe a specific type of initiator function that only initiates network requests and performs no other task. A gateway function essentially initiates network requests on behalf of other functions. In Figure 2, `sendRequest` is an example of a gateway function. Finally, we use the term `"neutralize"` to remove tracking in a JS by replacing a tracking function call with a mock function call.

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

Distinct tracking and non-tracking JS functions in the same script. Figure 1 illustrates a script containing both tracking and non-tracking JS functions, each with separate roles. In this example, `getIdentifier` gathers user data and sends a tracking request to `tracker.com` ❶, which returns a user identifier ❷. This identifier is used to track user activity on the webpage. The same script also includes functions like `displayContent` and `loadImage`, essential for the webpage to function properly. For instance, `loadImage` sends a non-tracking request ❸ to `CDN.com` to load an image ❹. Blocking the script to stop `getIdentifier` would also disable essential functions like `loadImage`, harming the webpage's functionality. Therefore, an ideal approach would neutralize tracking function calls to `getIdentifier` while leaving the rest of the script untouched.

Use of gateway functions to initiate tracking and non-tracking requests. Figure 2 illustrates how a gateway function, `sendRequest`,

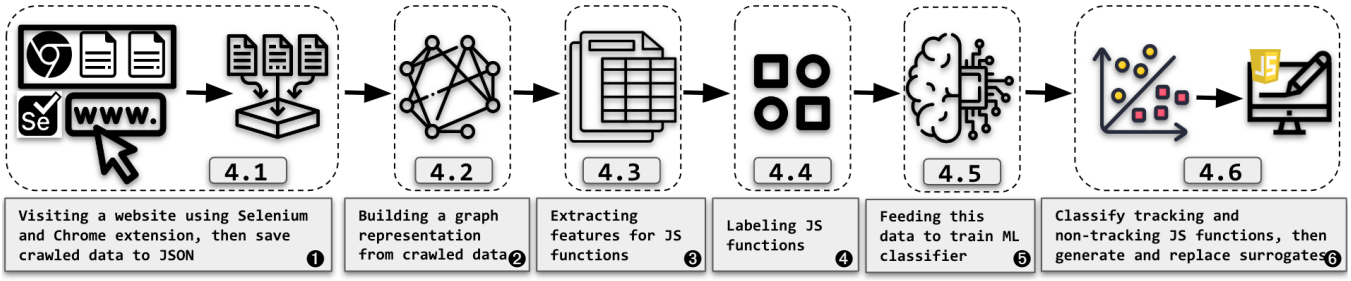


Figure 3: NoT.js pipeline: ❶ Crawl websites, save data; ❷ Create JS function graph; ❸ Extract features and ❹ label it; ❺ Train classifier; ❻ Classify tracking/non-tracking JS functions, and create surrogates.

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

Table 1: List of webpage's activities captured by NoT.js.

is used to handle both tracking and non-tracking network requests. The single script includes functions for both tracking and non-tracking tasks. Specifically, `getIdentifier` gathers user data for `tracker.com` to obtain a user identifier. Conversely, `loadImage` fetches images for the webpage from `CDN.com`. In contrast with the previous scenario, these functions do not initiate requests directly and delegate this task to the gateway `sendRequest` function. For instance, `getIdentifier` calls `sendRequest` ❶ to send request to `tracker.com` ❷. In return, `tracker.com` provides a user ID ❸, which is sent back to `getIdentifier` ❹. Similarly, `loadImage` calls `sendRequest` to fetch images from `CDN.com`, which are then displayed on the webpage in ❺ - ❼. The gateway functions further abstract tracking and non-tracking in JS code and necessitate careful analysis of the execution context of a JS function to determine the most effective strategy to block the execution of tracking JS code. Since the tracking function `getIdentifier` is not always at the top of the call stack (Figure 2), simply neutralizing the initiator function `sendRequest()` is not effective. Therefore, an ideal approach needs to incorporate a complete execution call stack and calling context to identify tracking JS functions. In this case, only the function calls to `sendRequest` invoked from `getIdentifier` should be neutralized, which preserves the functionality when `sendRequest` is invoked from `loadImage`.

4 NoT.js Design and Implementation

In this section, we describe the design and implementation details of NoT.js. Figure 3 provides an overview of NoT.js's pipeline, which starts with automated website crawling using the Selenium web driver and collecting data using a Chrome browser instrumented with a custom-built extension ❶. Using the collected data, NoT.js generates a graph representation ❷ of each webpage that encodes JS dynamic execution context of a comprehensive list of webpage's activities like network requests, DOM modifications, storage access, and a subset of other Web APIs (listed in Table 1), that are commonly used by trackers [55]. Using this graph representation, NoT.js extracts unique structural and contextual features of tracking activity ❸ and labels it ❹, which are then utilized to train a random forest classifier capable of accurately identifying tracking entities ❺. Finally, the classification results are utilized to generate surrogate scripts by neutralizing tracking function calls. These surrogate scripts can be used as replacements by content blockers at runtime ❻.

4.1 NoT.js's Chrome Instrumentation

NoT.js first collects the training data to train a fine-grained, high-accuracy classifier. It automates the website crawling and data collection process using selenium [38] and a custom-built Chrome extension. We choose Chrome extension interface [19] to capture web activities due to its ease of use and lightweight nature compared to instrumenting the JS engine, such as V8 in Chrome [42]. NoT.js's Chrome extension captures the JS dynamic execution context for each activity on the webpage, along with other relevant meta-data such as network requests and response payloads. NoT.js's dynamic execution context for each web activity comprises of:

- The **call stack** [3] outlines the sequence of function calls, including script URL, method name, and line and column numbers where each function is invoked.
- The **scope chain** [4] for each function call within the stack that includes the number of arguments and local and global variables.

The intuition behind capturing dynamic execution context is to gain a deeper understanding of the web activity. Figure 4 illustrates the two primary components of the Chrome extension, namely the background script and content script, that work together to facilitate data collection.

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

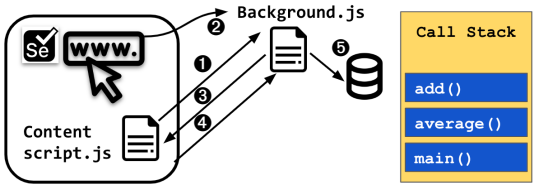


Figure 4: An illustration of NoT.js's chrome-extension, showing the communication sequence between background and content scripts.

Background script. The background script [16] is an essential component of the Chrome extension that captures webpage activity using the Chrome DevTools Protocol APIs [18]. Specifically, the network API [50] monitors traffic and provides valuable information about HTTP requests and responses, including headers, bodies, call stack, scope chain, and timestamps. Additionally, the DOM API [49] captures changes in the Document Object Model and provides read and write operations along with the call stack and the scope chain. However, the DevToolProtocol does not expose all Web APIs, cookies, or storage APIs available through the content script, as discussed next.

Content script. The content script [21] runs within the context of the webpage and can interact with its functionality. It is responsible for (1) collecting the JS dynamic execution context, *i.e.,* call stack and scope chain for the webpage's activity, and (2) exposing Web APIs, cookies, and storage APIs that are not available in the background script by overriding functions. Listing 1 shows a snippet from `content.js` that overrides the `sendBeacon` function of `Navigator` object. The overridden function collects two types of information: the call stack of JS execution and the scope chain of each function call in the call stack. The call stack is collected using the `console.trace()` [20] function, which logs the stack trace. The scope chain is collected using Debugger API [23], specifically via the `Debugger.paused` event that provides `callFrames.scopeChain` parameter.

```
// storing the original sendBeacon function
const sendBeac = Navigator.prototype.sendBeacon;
// overriding sendBeacon function
Navigator.prototype.sendBeacon = function(url, data) {
    // collect stack trace and scope chain
    console.trace();
    Debugger.paused.callFrames.scopeChain;
    // call back the original function
    sendBeac();}
```

Listing 1: Overriding `sendBeacon` function using content script.

Figure 4 illustrates the sequence of the data collection process with NoT.js's chrome extension. First, `content script.js` sets up communication with `background.js`, as shown in step ❶. On a network activity or DOM modifications (❷), `background.js` triggers a message for `content.js` to capture the corresponding JS execution call stack and scope chain, as shown in ❸. Finally, the collected data is sent to `background.js` for storage, as shown in ❹ and ❺. Additionally, the `content.js` uses the same communication channel to log cookies, storage, and APIs data in a storage. **Data collection.** We conduct an automated crawl of the landing pages of the top 10K websites in Tranco top-million list [69] as of July 2023,

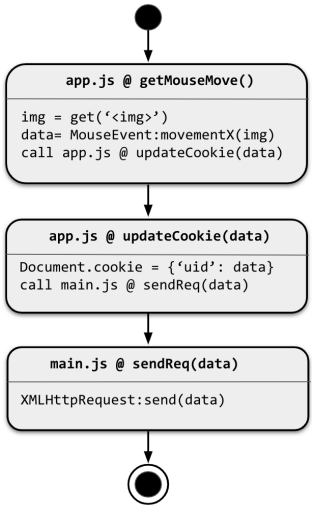


Figure 5: `getMouseMove()` collects data, calls `updateCookie(data)` to modify cookie, and triggers `sendReq(data)` to send the request.

using Selenium with Chrome 114.0.5735.133 and a purpose-built extension. We perform this crawl from the United States. On average, it took approximately 10 seconds for a webpage to fully load (until the `onLoad` event is fired) [32], and an additional 30 seconds before moving on to the next website. The crawling process is stateless, as we cleared all cookies and local browser states between consecutive crawls. This helps ensure that the collected data is reproducible and accurately reflects the current state of the webpage without being biased by previous webpage visits.

4.2 Graph Representation

NoT.js constructs a graph representation from the collected data of each webpage. NoT.js's graph representation leverages JS function-level features and JS execution context of each of the webpage's activity, as listed in Table 1. The graph's unique structure offers several advantages over traditional techniques. NoT.js allows for semantic reasoning with its dynamic execution context and enables traceability by providing a complete history of how a particular webpages activity is executed. This information is essential in differentiating the intention of the same graph node (*e.g.,* JS function) when participating in different execution scenarios. While its static representation remains unchanged, the JS execution context around such nodes at runtime may alter their semantics. By building a graph representation first, NoT.js facilitates the extraction of structural and contextually rich features, which it utilizes to identify privacy-invasive JS functions that are otherwise tightly interleaved with non-tracking code. NoT.js's graph representation is the first-of-its-kind [67, 81, 82] to incorporate the JS execution call stack and calling contexts (scope chain) fully.

Nodes. NoT.js categorizes a webpage's activity into five types of nodes: JS functions, DOM elements, network, storage, and web APIs. The JS function node represents a function call that has attributes, including its parent script's URL, name (except for anonymous functions), scope chain, line, and column number. The scope chain refers to the variables, functions, objects, and closures that a JS function can access at the time of invocation. Closures are a special

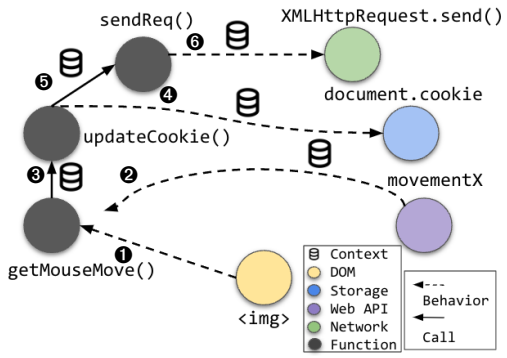


Figure 6: A simplified NoT.js's graph on mouse movement tracking.

type of function that can access variables in its enclosing function's scope chain, even after the function has returned. One function can generate multiple nodes depending on its calling sequence (call stack) and context (scope chain). This is particularly useful for gateway functions that participate in both tracking and non-tracking activity. For the threat model in Figure 2, NoT.js creates two separate nodes for the `sendRequest` function: one for its invocation within `getIdentifier` and another for `loadImage`. The DOM element node has attributes such as element name, class, or id and a value if available. The network node represents all the network requests that are sent by a webpage. The storage node represents all client-side data storing mechanisms, with attributes that differentiate between mechanisms such as cookies [22] and local storage [28]. The web API node type represents Web APIs, as listed in Table 1, with attributes that differentiate between APIs, such as charging time and discharging time. NoT.js's graph representation is the first to include function and web API node types.

Edges. Edges in the graph generated by NoT.js depict runtime dependencies between nodes. We extract two types of edges: call and behavioral edges. Call edges represent the sequence of function calls in a JS execution call stack. These edges connect function nodes to other function nodes to represent the dynamic caller-callee relationship. Whenever a function is called, a directed edge is created from the caller function node to the callee function node. These edges add valuable information about the sequence of function calls, which existing graph representations do not capture [67, 81]. Such edges enhance NoT.js's capabilities in modeling the semantics of the JS script by making its graph context-aware. On the other hand, behavioral edges are used to represent interactions between the JS functions, DOM, network, storage, and API nodes. If a function initiates a network request, a behavioral edge is created from the corresponding JS function node to the network node. Similarly, if a function triggers a storage or Web API call, a behavioral edge is created between the JS function node and the corresponding storage or Web API node, respectively. The direction of the edge is dependent on the context of the API call. For instance, if the JS function is storing data, there is an edge from the JS function to the storage or API node, and vice versa.

Graph construction. Figure 5 illustrates a sequence diagram for a webpage activity that tracks user movement on an HTML element, stores it in a cookie, and sends data to an external server. Each box in the illustration represents an entry of a JS execution call

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

Table 2: A small subset of features extracted from Figure 6.

stack. The `getMouseMove` function captures the mouse pointer's movement on the specified HTML element. This is achieved through `MouseEvent` API, which provides access to `movementX` property. This property represents the difference in the X coordinate of the mouse pointer between the current event and the previous mouse move event. Once the data (context) is collected, it is passed on to the `updateCookie` function, which updates the cookie value using `document.cookie` property. The data (context) is then passed to the `sendReq` function, which sends an HTTP request to an external server with the data.

Figure 6 shows a NoT.js's graph representation. The graph begins with a behavioral edge (dotted-line) that obtains mouse movement data associated with the specific HTML element, represented in ❶ and ❷. The call edge (solid-line) ❸ between `getMouseMove` and `updateCookie` represents a function call that also passes the mouse movement data (scope). The behavioral edge in ❹ represents the `document.cookie` call to update the storage node. The call edge ❺ between `updateCookie` and `sendReq` represents a function call that also passes the mouse movement data (scope). Finally, the behavioral edge ❻ between the function and the network node represents the HTTP request sent to an external server.

4.3 Feature Extraction

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

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

Contextual features. NoT.js includes JS dynamic execution context in the generated graph using contextual-based features. We extract three types of contextual features. First, we count the number of local variables, global variables, and closure variables. Second, we count the number of arguments passed to the function. Third, we include the number of network requests, DOM modifications, and API (*e.g.,* cookies and storage APIs) calls made by the function. These features play a vital role in understanding JS function behavior, especially during the execution of tracking activity. A

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

Table 3: The breakdown of the data employed to train, test, and validate the NoT.js classifier.

function's calling context has been used previously to construct the dynamic invariants of a program, which helps in verifying a given program behavior [72, 75, 93]. Such invariants are often implemented as assertions to detect unintended program behavior. Similarly, NoT.js's contextual features can help the downstream training process learn invariants about a JS function under which the JS function participates in tracking and the invariant under which it participates in non-tracking activity. Our graph representation is unique in its ability to extract contextual-based features that cannot be obtained by prior approaches [67, 81, 82].

Table 2 presents a subset of the features, including structural and contextual attributes, obtained from the graph representation of NoT.js shown in Figure 6. The `number of requests sent` feature is triggered for all functions that appear in the call stack when tracking requests are initiated with the `sendReq` function acting as the gateway. Contextual features encompass the number of storage (getter and setter) and web API (getter and setter) operations, as well as function arguments, providing insights into each function's contextual behavior. Structural attributes, specifically ascendant and descendant relationships in storage and web API nodes, enhance comprehension of hierarchical structures and code dependencies. For a detailed catalog of all features, please refer to Table 12 in the extended version [52].

4.4 Labeling

We adopt a prior approach [54] to label NoT.js's graph representation. We label network requests as either tracking or non-tracking based on whether their URL matches the rules in EasyList [8] and EasyPrivacy [9], which are widely employed by content blockers to identify tracking activity. While the filter lists have false negatives, they are highly accurate—a network request labeled as tracking by filter lists is actually tracking—as experts vet each rule in the filter list manually. Next, we analyze the request call stack and label a JS function as tracking if it exclusively participates in the stack trace of tracking requests. Otherwise, if a JS function participates in non-tracking requests (or a combination of tracking and non-tracking requests), we label it as a non-tracking JS function. This approach establishes a conservative ground truth where the functions with mixed behavior are labeled as non-tracking. However, these mixed functions comprise only 3.9% of our ground truth, as discussed in more detail in Section 7. In addition, we exclude functions that trigger web, storage, or cookie API calls but are not present in either a tracking or non-tracking request call stack. This is because we lack evidence from filter lists to label them.

4.5 Classification

We use a random forest classifier, which is an ensemble learning method. It constructs multiple decision trees and combines their predictions to obtain a final prediction. Prior work [67, 81] also use a random forest classifier since it outperformed other comparable models. To assess our classifier's performance, we divide our dataset into training, validation, and testing sets. Before splitting, we deduplicate our dataset by considering only one instance for each script URL and function name, ensuring no contamination between the training and testing phases. The numbers in Table 3 represent unique functions. Our dataset initially contained 3 million total functions, which was reduced to 1.8 million unique functions after deduplication. Specifically, we use 60% of the dataset for training the classifier, 20% for evaluating its accuracy during hyperparameter tuning, and the remaining 20% for final evaluation of the model. To optimize hyperparameters, we use the validation set to configure the depth of each decision tree in the forest to 20 and the number of trees used in the forest to 1000.

4.6 Surrogate Generation and Replacement

NoT.js generates surrogate scripts to replace mixed scripts in future page loads. NoT.js's surrogate generation and replacement is website-specific, which helps address the variations from obfuscation and minification techniques deployed by different websites. Our approach for surrogate generation involves neutralizing tracking function calls by substituting them with a mock function call within the script. Once surrogate scripts are generated for a specific website, we create a filter rule that substitutes the original script with the surrogate script at runtime.

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

Algorithm 1: Response Replacement Algorithm for Chrome Extension Manifest version 2.

Input: $scriptURL$ — the script URL to be requested
Output: Returns $surrogate$ if available, otherwise $None$ to indicate presence or absence of a replacement.

```
1: Procedure message, URL, responseCode
2:     if message == "Fetch.requestPaused" then
3:         response ← getResponse();
4:         if responseCode == 200 then
5:             if isInNOTJSFilter(URL) then
6:                 surrogate ← getSurrogateResponse();
7:                 if surrogate ≠ None then
8:                     response ← surrogate;
9:     Fetch.continueRequest(response);
```

Leveraging this information, we neutralize tracking function calls by substituting them with a mock call designed to consistently return an empty response. It is important to note that surrogate generation is an offline step.

To illustrate this process, consider the function call `track(o[0], o[1])` on `adobe.com` in Listing 2 that operates in the context of Adobe Analytics and is classified as tracking by NoT.js. NoT.js records the exact line (line 7) and column where this function

call begins. The column corresponds to the first element "t" of the function name "track" in this example. In the first step, we verify that the function call exists at the recorded line and column numbers based on its function name while skipping this step for anonymous functions. We replace the original function call with the mock call that returns an empty response. As we discuss later in Section 5.5, this simple approach effectively neutralizes a vast majority of tracking function calls while upholding the syntactic integrity of the script.

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

Listing 2: The example demonstrating the neutralization of tracking function calls during surrogate generation.

Surrogate replacement. NoT.js replaces the original mixed scripts with the generated surrogates at runtime during future page loads. NoT.js first identifies the target scripts on a webpage using regular expressions (regex) of generated surrogate scripts. For example, we create a regex rule to identify scripts associated with a domain like `adobe.com/*analytics.js`. Once identified, NoT.js replaces the target script with the corresponding surrogate. This replacement mechanism is supported in different Chrome extension environments like manifest versions 2 (V2) and 3 (V3). In manifest V2, Chrome extensions employ the Fetch API [25], a conventional method for intercepting and modifying responses at runtime. Algorithm 1 shows the steps for surrogate replacement. When a network request is made, the NoT.js's browser extension intercepts it, verifies that the response status is `OK` (status code 200), and subsequently modifies the response content. This approach allows us to replace a mixed script with the corresponding surrogate script. It is worth noting that this approach is similar to scriptlet replacement in content blocking tools such as uBlock Origin [40] and AdGuard [6]. Manifest V3 introduces the Declarative Net Request API [24] that does not allow direct response modification capabilities like V2. Instead, the original request is blocked and redirected to an alternate URL in manifest V3, effectively replacing the response content with the content retrieved from the redirected URL [30].

5 Evaluation

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

5.1 Accuracy Analysis

We train NoT.js's random forest classifier on approximately 1.1 million JS functions in the training set, do hyper-parameter tuning on 361 thousand JS functions in the validation set, and then evaluate its accuracy on 361 thousand JS functions in the testing set. Table 3 presents a breakdown of the data split between tracking and non-tracking functions. Table 4 shows that NoT.js achieves a precision of 94.3% and a recall of 98.0%. The overall F1-score for NoT.js is 96.2%, indicating its effectiveness in accurately distinguishing between tracking and non-tracking JS functions. A confusion matrix (Table 8) of this analysis is attached in the appendix. Additionally, we perform 5-fold cross-validation to assess NoT.js's accuracy. We split the dataset into five equally sized folds, where the model is trained on four folds and evaluated on the remaining fold, repeated five times so that each fold serves as the testing set once. In cross-validation, NoT.js achieves a comparable precision of 94.6 and a recall of 96.2, with an overall 95.3±0.3% F1-score, averaged over 5-folds.

Model	Section	Precision	Recall	F1 Score
NoT.js	Standard - 5.1	94.3%	98.0%	96.2%
NoT.js	Obfuscation - 5.3	93.5%	90.4%	91.9%
NoT.js	Coverage - 5.3	88.4%	95.7%	91.9%
WebGraph	Comparison - 5.4	49.3%	66.4%	56.5%
SugarCoat	Comparison - 5.4	23.0%	22.6%	22.8%

Table 4: NoT.js's precision, recall, F1-score in standard settings, enhanced coverage, obfuscation robustness, and comparison with existing tools.

Error analysis. While NoT.js has a relatively low false positive rate (3.5%) and false negative rate (1.9%), we investigate the reasons and contexts of its errors. We conduct a manual evaluation by randomly sampling 50 instances where NoT.js incorrectly identifies JS functions as tracking, despite our ground-truth data missing them. The causes for these errors can be categorized into three primary categories:

The first category includes functions in mixed scripts that cannot be blocked by filter lists to prevent website functionality breakage. For example, the function `t` in `js.cookie.min.js` on `kakaku.com` sets the tracking cookie and is classified as tracking by NoT.js, while its other functions primarily serve the history feature on the website. Consequently, being a mixed script, it cannot be blocked by filter list authors (acknowledged in this GitHub issue [45]), yet it can be handled by NoT.js.

The second category includes functions that are actually involved in tracking but missed by filter lists [51]. For example, the function `b` in the script `htlbid-advertising.min.js` on `wkrn.com` manages ad slots and their configurations. We conduct an in-depth manual evaluation of the entire script, discovering that most of its functions are classified as tracking by NoT.js. Following this, we report this issue [46] to filter list authors, leading to its inclusion in the filter rules. In total, NoT.js identifies ten such cases out of a sample of fifty in the aforementioned two categories that are either missed or cannot be handled by EasyList/EasyPrivacy. We report these cases to filter list authors. The filter list authors confirm (via Github issues [45–48]) that these mixed scripts are not blocked by filter lists despite being known to implement tracking due to breakage concerns. This highlights NoT.js's potential to provide finer-grained tracker blocking as compared to EasyList/EasyPrivacy.

The last category includes functions, where 40 out of 50 instances represent an actual error by NoT.js. We use the number of arguments and local and global variables in the scope chain rather than examining the types or values passed to these variables. Therefore, functions with identical dynamic execution contexts—*i.e.,* the same

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

Table 5: Ablation analysis results of NoT.js's features in terms of precision, recall, and F1-score.

call stack and scope chain—for both tracking and non-tracking activities are labeled as non-tracking in the ground truth but classified as tracking by NoT.js. These instances are not technically misclassified by NoT.js, but rather point to the limitations intrinsic to dynamic execution context. Such multi-purpose functions can serve both tracking and non-tracking roles.

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

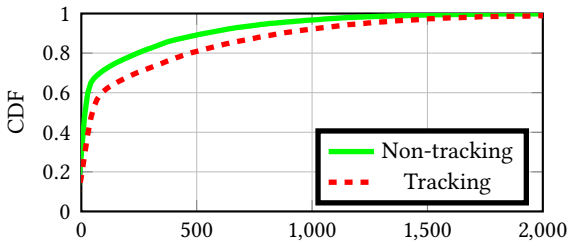
Listing 3: `_setField` function from the `Microsoft` script domain

For instance, in Listing 3, `_setField` function in `visitor*.js` script is erved by `microsoft.com`. If parameter `f` is set to false, the function participates in the tracking activity on the website. However, if parameter `f` is set to true, the function participates in the non-tracking activity on the website. In ground truth, this function is labeled as non-tracking because of the same number of arguments. A deeper examination of parameter types and values could address these issues in future models. In summary, 20% of the tracking functions in the sample of 50 represent mistakes in the filter list, while the remaining 80% constitute actual mistakes by NoT.js.

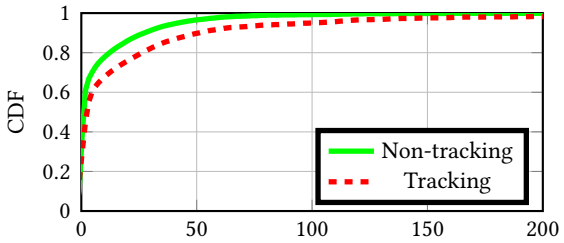
False negatives predominantly stem from code coverage, as not all properties of tracking functions are captured. For instance, a more comprehensive list of Web APIs could be employed to capture additional characteristics, assisting in more precise profiling of tracking functions and, consequently, fewer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

5.2 Feature Analysis

We analyze feature importance using information gain to better understand which features are potentially useful to NoT.js. The most important feature is the number of successor functions, a structural feature, followed by the number of local storage accesses, a contextual feature. The number of successor functions provides insights into the function's calling context. Successor functions show how connected a function is to other parts of the code. Figure 7a shows that tracking JS functions tend to have more successor functions than non-tracking functions. The average number of successor functions is 252 for tracking functions and 148 for non-tracking functions. The higher number of successor functions in tracking activities indicates more complex behavior, like gathering, processing, and transmitting personal data. Additionally, the number of storage accesses is the second most important feature in distinguishing tracking functions from non-tracking. Previous research recognizes that storage APIs are commonly employed by trackers [81]. Figure 7b shows that tracking JS functions have a higher frequency of



(a) Number of successor functions



(b) Number of storage access

Figure 7: CDF highlights NoT.js features: (a) successor functions, and (b) storage access, in tracking vs non-tracking contexts.

storage accesses than non-tracking functions. The average storage access is 15 in tracking functions and 8 in non-tracking functions.

Ablation analysis. Next, we evaluate the impact of various graph configurations and features. We summarize our findings in Table 5. First, we find that solely incorporating initiator functions in the graph while excluding non-initiator functions lowers the F1-score of NoT.js by 29.3%. This reduction is primarily due to the limitation that focusing solely on initiator functions omits the broader context essential to distinguish between tracking and non-tracking activities. Next, we include both initiator and non-initiator functions but exclude the execution context features. Under this configuration, although precision decreases, there is a notable increase in the recall. This shows that the model now possesses broader code coverage—it identifies a greater number of tracking functions, thereby elevating recall. However, this leads to an increase in false positives. We observe that a call to the same function in tracking and non-tracking contexts has the same representation in the graph; thus, the extracted features are not distinctive enough to accurately segregate tracking functions from non-tracking ones. Finally, we include all functions and contextual features. This setting surpasses the performance of all prior configurations owing to the enhanced code coverage and execution context. The integration of calling context in NoT.js results in the best precision and overall F1 score. This underscores the pivotal role of JS dynamic execution context with both initiator and non-initiator functions, emphasizing its importance in precisely identifying tracking JS functions.

5.3 Robustness

We evaluate the robustness of NoT.js in detecting tracking functions amid manipulation attempts such as JS code obfuscation and code modifications, as well as in the context of enhanced code coverage.

Code obfuscation. JS obfuscation is often employed to conceal the meaning of the code, making it harder to decipher for those who may try to reverse engineer or modify it [60, 94]. From our

dataset of 10K websites, we randomly selected 10%, which includes 15,939 unique scripts. We then obfuscate these scripts using the `obfuscater.io` [31], with the configuration shown in Listing 4.

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

Listing 4: Configuration used to obfuscate the scripts using obfuscator.io.

JS obfuscation poses several challenges for NoT.js. First, it modifies the names of JS functions using hexadecimal notation, making it difficult for NoT.js to identify and track the execution of these functions accurately. Second, it alters the JS execution call stack through the use of multiple techniques, such as control flow flattening [56, 90] and self-defending [77, 78]. Control flow flattening involves breaking down JS functions into smaller basic blocks and then rearranging these blocks to change the order of execution. This can make it difficult for NoT.js to understand the flow of the code and track the execution of different functions. Self-defending adds protection code to the program that can detect if it is being debugged or modified and attempt to avoid execution in these cases. The protection code can prevent NoT.js from gathering the necessary information by changing the execution call stack of JS activity.

Table 4 presents the classification results of NoT.js on the obfuscated data using the aforementioned configuration. Overall, obfuscating the JS code reduces the precision by only 0.8% and recall by 7.6%, leading to a 4.3% reduction in the overall F1 score. The marginal drop in recall can be attributed to the model's increased likelihood of missing additional tracking functions. This can be traced back to the non-operational "garbage" functions in the stack, which the model finds challenging to accurately identify. Given NoT.js's reliance on the dynamic execution context over static features, its precision remains unaffected by JS obfuscation. To further enhance NoT.js's robustness again script obfuscation, future work could incorporate adversarial training by explicitly adding obfuscated scripts in the training set [62].

Enhanced code coverage. To assess the impact of enhanced code coverage on NoT.js's performance, we conduct a more exhaustive crawl on a randomly selected subset, making up 10% of our original 10K website corpus. During this analysis, in addition to assessing the landing page, we explored five distinct internal pages of each website. We implement bot mitigation strategies that include simulating mouse movements at five unique offsets using the `move_by_offset(x, y)` function. Furthermore, we incrementally scroll through the webpage using the `window.scrollBy()` function. Upon completing the website crawl, we utilize the NoT.js to classify the involved JS functions. As summarized in Table 4, our results indicate a 5.9% decline in precision, which led to a 4.2% reduction in the F1-score. Consequently, the data reveals only a minor decrease in both precision and the overall F1-score for NoT.js, suggesting that it maintains a reliable level of accuracy even with expanded coverage.

5.4 Comparison with Existing Countermeasures

We compare the performance of NoT.js against two state-of-the-art baselines: WebGraph and SugarCoat, evaluating their precision and recall specifically in identifying tracking functions. Later, in section 5.5, we assess the impact on website functionality, focusing on website breakage.

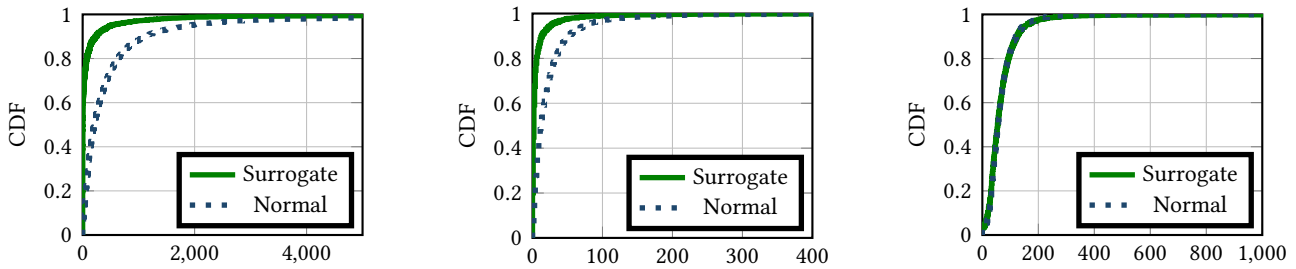
WebGraph comparison. WebGraph, by default, classifies tracking script URLs, presuming all functions in the script have the same label as the script's URL. In contrast, NoT.js uniquely classifies each JS function individually. The WebGraph and NoT.js are compared on the same dataset. We crawl each website once and then build two different graphs – one for WebGraph[5] and one for NoT.js, as the graph representations differ between the two approaches. The appendix 9.1 provides a detailed account of the key differences in graph representations between WebGraph and NoT.js.

Table 4 summarizes the results for NoT.js and WebGraph. NoT.js outperforms WebGraph, which classifies script-URL, in classifying tracking JS functions within mixed scripts with 45.0% higher precision, 31.6% higher recall, and an overall 39.7% better F1-score. WebGraph is unable to fully capture the communication edges between the scripts present in the call stack during a tracking activity. As a result, the dynamic execution context surrounding data transfer and the initiator script, which sends the data to the server, is not captured and modeled. Furthermore, tracking at the script-level granularity obscures important features that are associated with fine-grained JS functions, making it difficult to distinguish tracking and non-tracking activity in mixed scripts [54]. Although WebGraph claims a 92% accuracy rate in its paper, this is largely because it is tailored to identify tracking script URLs. Its limitations become evident when dealing with mixed scripts, underscoring the need for function-level granularity.

SugarCoat comparison. SugarCoat follows a three-step process. First, a developer curates a list of mixed scripts containing both tracking and non-tracking functions. Second, SugarCoat generates a behavioral graph for these target mixed scripts, pinpointing six privacy-relevant API accesses to specific locations within the JS source code by analyzing the collected data. Third, SugarCoat creates a surrogate of the target mixed scripts, redirecting the identified API accesses to manually crafted mock implementations [36]. SugarCoat requires significant manual effort, including identifying mixed scripts and crafting handwritten mock API implementations. In contrast, NoT.js relies on a detection model to automatically identify mixed scripts and does not rely on handwritten mock API implementations. Thus, it is unsurprising that SugarCoat has approximately two hundred surrogate scripts only [37].

Since SugarCoat lacks an automated detection model, for the sake of comparison, we assume that the functions modified in its publicly available surrogates [37] are classified as tracking functions. For example, in the SugarCoat repository, we find a script ads.google.com/aw/JsPrefetch?origin=lead_in_page [15]. Its

[5]We use Structural + Flow features for WebGraph since the Content features are not robust, as shown in the WebGraph paper.



(a) Number of tracking functions (b) Number of tracking requests (c) Number of non-tracking requests

Figure 8: CDF illustrates pre- and post-surrogate replacement metrics for websites: tracking functions, tracking requests, non-tracking requests.

function `prefetchJS()` is replacing the default `localStorage` API with the mock implementation, indicating that SugarCoat classified this function as tracking. In contrast, NoT.js utilizes its classification model for this purpose. Table 4 provides a summary of the results for NoT.js and SugarCoat. NoT.js outperforms SugarCoat in detecting tracking JS functions with 71.3% higher precision, 75.8% higher recall, and an overall 73.4% better F1-score. As SugarCoat is designed to address only mixed scripts, we precisely compute the precision and recall for only mixed scripts. Even in this favorable comparison scenario, SugarCoat achieves 27.5% precision, 35.1% recall, and an overall 30.9% F1-score in detecting tracking JS functions. This clearly shows that SugarCoat's manual detection and dependency on six manually crafted mock implementations for tracking functions makes it infeasible to apply at scale.

5.5 Surrogate Generation and Replacement

We assess the feasibility of NoT.js's real-time deployment through the evaluation of its surrogate generation and replacement strategy. For our evaluation, surrogates are generated for a randomly selected 50% of the 10K websites, after which we re-visit each site, substituting the original script with the surrogate prepared by NoT.js. We utilize the surrogate replacement strategy as outlined in Section 4, implemented as a Chrome extension.

Surrogate generation. As elaborated in Section 4, NoT.js locates the tracking function call within the script source code and substitutes it with a mock function call, effectively neutralizing it. During the surrogate generation, we calculate four parameters to assess the efficacy of the technique. First, we measure the average number of tracking function calls per webpage that can be successfully neutralized, which is 122.3. Second, we measure the average number of tracking function calls per webpage that cannot be neutralized due to the limitations of our instrumentation, which fails to capture the script source code. The average number of such tracking functions per webpage is 3.1. Third, we measure the average number of tracking function calls per webpage that cannot be neutralized due to script inlining—*i.e.,* embedded within the HTML of the page. The average number of such tracking functions per webpage is 4.7. Finally, we measure the average number of tracking function calls per webpage that cannot be neutralized owing to the dynamism in the JS [91, 92], rendering us unable to confirm the line number and column number at run-time. The average number of such tracking functions per webpage is 14.8. In summary, our surrogate generation technique successfully neutralizes an average of 84.4% of classified tracking functions per webpage.

Time and space analysis. The average time required to generate a surrogate is 1.31 seconds. The size of an average surrogate generated by NoT.js is 164 KB. For 50% of the 10K websites, a total space of 9.6 GB is required. However, users of NoT.js do not need to download all surrogate scripts at once. We discuss in more detail in Section 7 how NoT.js can be implemented more efficiently. For example, it can cache surrogate scripts for commonly visited websites and potentially pre-fetching popular scripts.

Surrogate replacement. After generating the surrogates, we evaluate both privacy and usability metrics across 50% of these websites, both before and after surrogate deployment. We calculate the average number of tracking functions, as well as tracking and non-tracking requests per website. Furthermore, we carry out a performance and user-centered manual breakage analysis to verify the usability of websites after surrogate deployment. Originally, the average number of tracking functions per website is 153.6, which drops to 38.5 after deploying the surrogates, representing an 80.1% reduction per website. Figure 8a displays the CDF that shows a significant decrease in tracking functions after surrogate deployment. Moreover, the average count of tracking requests per website is initially 28.0. This decreases to 8.4 post-deployment, representing a 76.9% reduction per website. Figure 8b displays the CDF that shows a significant decrease in tracking requests on the majority of websites following surrogate deployment. Finally, the average number of non-tracking requests per website is 74.3 and drops minimally to 71.5 after deploying the surrogates. Figure 8c displays the CDF, illustrating the number of non-tracking requests against the website distribution. The overlapping lines in the graph indicate negligible impact on website usability.

Performance analysis. We evaluate the performance overhead across 50% of these websites, both before and after surrogate deployment, utilizing Selenium [38] to extract standard page performance metrics for each visit. These metrics include JS memory usage and the timing of key page load events. Table 6 summarizes the results. Regarding JS memory usage, we observe a decrease of 32% in total and 25% in used JS heap memory. The total JS heap size represents the entire memory allocation for JS execution, while the used portion reflects the memory actively utilized by JS on the webpage. This decrease is primarily due to the neutralization of tracking function calls, which return an empty response instead of executing the original memory-consuming function.

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

Table 6: Performance analysis for the post-surrogate replacement.

A standard benchmark [27] for user-perceived web page performance measures the timeline of key events marking various stages in the browser's page load and rendering process. With NoT.js's surrogates, these events are completed on average 32 milliseconds earlier. These improvements are seen across several metrics: DOM content load time, which indicates the time from page load start to complete, HTML parsing, and initial DOM interactive time. DOM interactive time indicates the duration until the DOM is fully prepared for user interaction, and load event time indicates the total time to completely load all resources, such as images and CSS, signifying the full usability of a webpage. This overall improvement is expected, as our neutralized tracking function calls avoid invoking the original tracking functions, performing less work by simply returning an empty response.

User-centric breakage analysis. We conduct a qualitative manual analysis of NoT.js, following methodologies from previous studies [54, 74]. We select 50 webpages from the top-10K websites, specifically those hosting scripts mentioned in the exception rules of filter lists[6] and addressed with SugarCoat's six mock API implementations [36]. These are mainly mixed scripts; content blockers typically avoid blocking them to prevent website breakage, even though allowing them may compromise user privacy.

One of the authors evaluates all fifty webpages. We recruit ten additional independent evaluators for breakage assessments. Each independent evaluator evaluates five distinct websites from our sample. Each webpage is evaluated by at least two different evaluators.

The webpages are evaluated in four different configurations: (1) Control, which displays the default webpage without any blocking; (2) WebGraph, highlighting the limitations of advanced script-level blocking in mixed script scenarios; (3) SugarCoat, highlighting its effectiveness, though it faces scalability challenges, especially with limited mock API implementations [36] (4) NoT.js, showing a reduced impact on website functionality compared to WebGraph, while being on par with SugarCoat in terms of handling mixed scripts. We ensure high coverage for NoT.js by ensuring that 98% of the surrogate scripts are triggered at page load. This ultimately highlights that NoT.js significantly advances the handling of mixed scripts compared to current state-of-the-art approaches. We present WebGraph, SugarCoat, and NoT.js in a randomized sequence to each evaluator, revealing only the control configuration.

We ask evaluators to classify breakage into four categories: navigation (moving between pages), SSO (initiating and maintaining login state), appearance (visual consistency), and miscellaneous (such as chats, search, and shopping cart). Each evaluator labels breakage as either major or minor for each category: Minor breakage occurs when it is difficult but not impossible for the evaluator to use the functionality. Major breakage occurs when it is impossible to use the functionality on a webpage. The inter-evaluator agreement was 95.5%, suggesting substantial agreement. Conflicts are resolved by one of the authors revisiting those websites. Conflicts mainly stem from ambiguity between appearance and miscellaneous categories, such as the interchangeable reporting of a missing ad overlay in both.

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

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

Table 7: Qualitative manual analysis for 50 webpages using NoT.js, SugarCoat, and WebGraph, showing % of No, Minor, and Major breakages in navigation, SSO, appearance, and miscellaneous categories. 4% of minor appearance breakages with NoT.js are due to missing ad overlay, as shown in Figure 9 in the appendix.

Table 7 summarizes the results in each category. WebGraph causes major breakage on 6% and minor breakage on 10% of the webpages. In contrast, NoT.js, only causes minor breakage on 8% of the webpages, without any major breakage. As compared to WebGraph, NoT.js causes significantly less major breakage in the navigation category. For example, on the webpage `bbc.com`, the search bar in the navigation menu disappears with WebGraph, resulting in major breakage. This major breakage stems from the blocking of the mixed script hosted by `bbc.com`. NoT.js effectively mitigates tracking in the mixed script by generating a surrogate replacement. While NoT.js performs comparably to SugarCoat in terms of major breakage, it results in 4% more minor breakage in the appearance category. As shown in Figure 9, this minor breakage is due to a missing ad overlay left after successfully neutralizing the tracking function. Overall, NoT.js matches SugarCoat in terms of breakage while being more scalable due to its automated surrogate generation as compared to SugarCoat's reliance on just six manually crafted mock implementations.

6 In the Wild Deployment

In this section, we analyze the tracking JS functions detected by NoT.js on top-10K websites.

Prevalence of tracking functions. NoT.js classifies 32.1% of the 2,088K JS functions in our dataset as tracking. We find that these tracking functions are present in the scripts served by 8,587 unique domains. Among these tracking functions, 8.2% are anonymous functions, 18.3% are part of inline scripts, and 1.5% are part of eval scripts. Table 13 in the extended version of the paper [52] lists the top-5 most prevalent tracking functions across top-10k websites. For instance, the function `"Z.D"` from the script `analytics.js` hosted by `google-analytics.com` appears on 56% of the websites. This function, on average, invokes cookie setter 3.3 times and cookie getter 20 times. Similarly, function `"c"` from the script `fbevents.js`, hosted by `connect.face`

book.net, appears on 21.5% of the websites. Its typical calling context contains 3.9 closures and 6.6 get attribute (getAttribute) calls, along with 2.5 cookie accesses.

Characteristics of tracking functions in mixed scripts. We find that 13.4% of all scripts are mixed, aligning with previous studies [53, 54], while 62.3% of websites incorporate at least one mixed script. On average, a website contains around 2.42 mixed scripts. The overview of the domains (eTLD + 1) serving the most mixed scripts is provided in the appendix (Table 11 in the extended version of the paper [52]). Notably, 70.6% of the mixed scripts are served from third-party domains[7].

Our analysis of the top-100 mixed scripts reveals that they are commonly included in the first-party context, enabling the setting of ghost first-party cookies [79]. These scripts set 14,867 ghost first-party cookies, out of which 150 are found to be tracking after running CookieGraph [74], a tool designed to detect first-party tracking cookies. NoT.js classified 83% of JS functions in these mixed scripts as tracking functions that are either setting or getting these ghost first-party tracking cookies [74]. For example, NoT.js detected the tracking function "a" within the script launch-*.min.js which is accessing the ghost first-party tracking cookie mbox on adobe.com. As another example, NoT.js detected the tracking function "o" within the script opus.js which is accessing the ghost first-party tracking cookie A1S on yahoo.com.

To illustrate how NoT.js tackles our threat model, consider these two types of mixed scripts. The script named webpack-*.js is served by the domain cloudfront.net. NoT.js detects tracking and non-tracking functions within this mixed script. Specifically, the function "t" accesses local storage four times, sets it six times, and attaches event listeners to 244 different DOM elements, detected as tracking. In contrast, the "a" function in the same script avoids interactions with both local storage and the DOM, detected as non-tracking. Another scenario involves the script app.js from the domain acsbapp.com, which contains the function "_e". The behavior of this function depends on its dynamic execution context, detected by NoT.js. In a non-tracking calling context, the number of closures and local variables for the function are 3 and 1, respectively. However, in a tracking calling context, these values are 0, emphasizing the importance of context to distinguish between tracking and non-tracking behaviors.

7 Discussion

In this section, we discuss some opportunities for future work and limitations of NoT.js.

User interaction limitations. The interactions captured by NoT.js depend on the diversity and intensity of user activity, such as scrolling or clicking on internal pages. NoT.js may miss certain tracking functions due to limited user interactions. To mitigate this, we propose to use forced execution [68, 76, 86] in the future to improve the completeness of the webpage's graph.

Browser-specific deployment. NoT.js uses the Chrome browser and Chrome-based extensions to collect data due to its popularity. Extensions on other browsers (Firefox [26], Safari [35], Edge [29]) have different permissions and access to varying sets of information about a webpage's activity. Porting NoT.js on other browsers may require additional engineering.

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

Resource-constrained environments. While NoT.js is adept at generating surrogate scripts at scale, its deployment on devices with limited storage presents a challenge. We can implement specific strategies for devices with limited storage. First, NoT.js can prioritize caching popular scripts that are frequently accessed, ensuring these replacements are readily available locally to reduce reliance on server requests. Second, we can employ selective prefetching techniques to allow NoT.js to anticipate and fetch surrogate scripts for popular websites or commonly visited domains in advance. Third, for less frequently accessed or uncached scripts, NoT.js can dynamically request them.

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

Usage of bundlers. To investigate mixing tracking with non-tracking functions, we identify bundled scripts within mixed scripts using a high-precision heuristic from the Web Almanac [14], focusing on the webpackJsonp keyword in scripts[8]. This keyword, default in JSONP functions, indicates asynchronous loading of bundled scripts, commonly with Webpack [43]. We observe that 20.2% of mixed scripts are Webpack-bundled, comparable to functional scripts (20%) but exceeding tracking scripts (10%). However, potential imperfect recall exists as webpackJsonp can be altered in Webpack's settings, and this heuristic doesn't cover scripts bundled via tools like Parcel [33], Rollup [34], or Browserify [17], thus underrepresenting the actual count of mixed, bundled scripts. Future investigations could enhance script-mixing understanding by developing more precise heuristics for bundled script identification.

8 Conclusion

NoT.js advances the state-of-the-art by identifying tracking JS functions in mixed scripts and generating surrogates to replace mixed scripts. In doing so, NoT.js achieves 94% precision and 98% recall, surpassing state-of-the-art in terms of accuracy, robustness, and minimizing breakage. Our analysis shows that NoT.js can detect and neutralize mixed scripts that are not blocked by filter lists due to breakage concerns despite being known to engage in tracking activities. As the arms race evolves, finer-grained tracker blocking tools such as NoT.js will be imperative to handle mixed scripts.

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

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

Acknowledgments

This work is partly supported by the National Science Foundation under grant numbers 2103439, 2103038, 2138139, and 2106420. Additionally, part of the work is funded by 4-VA and MECRT, Indonesia. We want to thank the anonymous reviewers for their constructive feedback that helped improve the work.

References

[1] 2016. A New Way to Control the Ads You See on Facebook, and an Update on Ad Blocking. https://newsroom.fb.com/news/2016/08/a-new-way-to-control-the-ads-you-see-on-facebook-and-an-update-on-ad-blocking/.
[2] 2016. Ping pong with Facebook. https://web.archive.org/web/20160818160457/https://adblockplus.org/blog/ping-pong-with-facebook.
[3] 2017. AD Blocker's successful assault on Facebook enters its second month. https://web.archive.org/web/20221011151030/https://adage.com/article/digital/blockrace-adblock/311103.
[4] 2018. Ad Network Uses DGA Algorithm to Bypass Ad Blockers and Deploy In-Browser Miners. https://www.bleepingcomputer.com/news/security/ad-network-uses-dga-algorithm-to-bypass-ad-blockers-and-deploy-in\-browser-miners/.
[5] 2018. Who is Stealing My Power III: An Adnetwork Company Case Study. https://web.archive.org/web/20230408021633/https://blog.netlab.360.com/who-is-stealing-my-power-iii-an-adnetwork-company-case-study-en/.
[6] 2019. AdGuard Scriptlets and Resources. https://github.com/AdguardTeam/Scriptlets.
[7] 2020. WebBundles Harmful to Content Blocking, Security Tools, and the Open Web. https://web.archive.org/web/20230207160252/https://brave.com/web-standards-at-brave/3-web-bundles/.
[8] 2022. EasyList. https://easylist.to/easylist/easylist.txt.
[9] 2022. EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt.
[10] 2022. Enhanced Tracking Protection in Firefox for desktop. https://support.mozilla.org/en-US/kb/enhanced-tracking-protection-firefox-desktop.
[11] 2022. Facebook Is Now Encrypting Links to Prevent URL Stripping. Website. https://web.archive.org/web/20230527091045/https://www.schneier.com/blog/archives/2022/07/facebook-is-now-encrypting-links-to-prevent-url-stripping.html
[12] 2022. Tracking Prevention in Microsoft Edge. https://docs.microsoft.com/en-us/microsoft-edge/web-platform/tracking-prevention.
[13] 2022. uBlock Origin. https://github.com/gorhill/uBlock. https://github.com/gorhill/uBlock
[14] 2022. Web Almanac - JavaScript. https://web.archive.org/web/20230422093455/https://almanac.httparchive.org/en/2022/javascript.
[15] 2023. ads.google. https://shorturl.at/nGNW1.
[16] 2023. Background Script. https://developer.chrome.com/docs/extensions/mv2/background_pages/.
[17] 2023. Browserify. http://browserify.org/.
[18] 2023. Chrome DevTools Protocol — chromedevtools.github.io. https://chromedevtools.github.io/devtools-protocol/.
[19] 2023. chrome-extension. https://developer.chrome.com/docs/extensions/.
[20] 2023. Console Trace. https://developer.mozilla.org/en-US/docs/Web/API/console/trace.
[21] 2023. Content Script. https://developer.chrome.com/docs/extensions/mv3/content_scripts/.
[22] 2023. Cookie Access. https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie.
[23] 2023. Debugger API. https://chromedevtools.github.io/devtools-protocol/tot/Debugger/.
[24] 2023. Declarative Net Request API. https://developer.chrome.com/docs/extensions/reference/declarativeNetRequest/.
[25] 2023. Fetch API. https://chromedevtools.github.io/devtools-protocol/tot/Fetch/.
[26] 2023. Firefox Browser. https://www.mozilla.org/en-US/firefox/.
[27] 2023. Google. 2021. Chrome User Experience Report. https://web.dev/articles/critical-rendering-path/measure-crp.
[28] 2023. Local Storage Access. https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage.
[29] 2023. Microsoft-Edge Browser. https://www.microsoft.com/en-us/edge?form=MA13FJ.
[30] 2023. Migrate from Manifest V2 to Manifest V3. https://developer.chrome.com/docs/extensions/migrating/blocking-web-requests/.
[31] 2023. obfuscater.io. https://obfuscator.io/.
[32] 2023. onLoad Event. https://httparchive.org/reports/loading-speed.
[33] 2023. Parcel. https://parceljs.org/.
[34] 2023. Rollup. https://rollupjs.org/.
[35] 2023. Safari Browser. https://www.apple.com/safari/.
[36] 2023. SCmocks. https://github.com/SugarCoatJS/sugarcoat/tree/master/mocks.
[37] 2023. SCresources. https://github.com/SugarCoatJS/sugarcoat-paper-dataset/tree/master/resources.
[38] 2023. Selenium. https://www.selenium.dev/.
[39] 2023. uBlock Origin Google Analytics Replacement. https://github.com/gorhill/uBlock/blob/master/src/web_accessible_resources/google-analytics_analytics.js.
[40] 2023. uBlock Origin Scriptlets. https://github.com/uBlock-user/uBO-Scriptlets. https://github.com/uBlock-user/uBO-Scriptlets
[41] 2023. uBlock Origin web accessible resources. https://github.com/gorhill/uBlock/tree/master/src/web_accessible_resources.
[42] 2023. V8. https://en.wikipedia.org/wiki/V8_(JavaScript_engine).
[43] 2023. Webpack. https://webpack.js.org.
[44] 2023. Webpack Downloads. https://npmtrends.com/webpack.
[45] 2024. GitHub Issue-1. https://github.com/easylist/easylist/issues/18242.
[46] 2024. GitHub Issue-2. https://github.com/easylist/easylist/issues/18230.
[47] 2024. GitHub Issue-3. https://github.com/easylist/easylist/issues/18243.
[48] 2024. GitHub Issue-4. https://github.com/easylist/easylist/issues/18185.
[49] [Accessed 30-Apr-2023]. DOMDomain. https://chromedevtools.github.io/devtools-protocol/tot/DOM/.
[50] [Accessed 30-Apr-2023]. NetworkDomain. https://chromedevtools.github.io/devtools-protocol/tot/Network/.
[51] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. 2019. Errors, misunderstandings, and attacks: Analyzing the crowdsourcing process of ad-blocking systems. In *Proceedings of the Internet Measurement Conference*. 230–244.
[52] Abdul Haddi Amjad, Shaoor Munir, Zubair Shafiq, and Muhammad Ali Gulzar. 2024. Blocking Tracking JavaScript at the Function Granularity. arXiv:2405.18385 [cs.CR]
[53] 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*. Association for Computing Machinery.
[54] Abdul Haddi Amjad, Zubair Shafiq, and Muhammad Ali Gulzar. 2023. Blocking JavaScript without Breaking the Web: An Empirical Investigation. *arXiv preprint arXiv:2302.01182* (2023).
[55] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. 2022. FP-Radar: Longitudinal measurement and early detection of browser fingerprinting. *Proceedings on Privacy Enhancing Technologies* 2022, 2 (2022), 557–577.
[56] Vivek Balachandran, Darell JJ Tan, Vrizlynn LL Thing, et al. 2016. Control flow obfuscation for android applications. *Computers & Security* 61 (2016), 72–93.
[57] Brave. 2022. A Long List of Ways Brave Goes Beyond Other Browsers to Protect Your Privacy. https://brave.com/privacy-features/. https://brave.com/privacy-features/
[58] 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.
[59] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. 2021. Detecting Filter List Evasion with Event-Loop-Turn Granularity JavaScript Signatures. In *2021 IEEE Symposium on Security and Privacy (SP)*.
[60] YoungHan Choi, TaeGhyoon Kim, SeokJin Choi, and Cheolwon Lee. 2009. Automatic detection for javascript obfuscation attacks in web pages through string pattern analysis. In *Future Generation Information Technology: First International Conference, FGIT 2009, Jeju Island, Korea, December 10-12, 2009. Proceedings 1*. Springer, 160–172.
[61] Ha Dao, Johan Mazel, and Kensuke Fukuda. 2021. CNAME Cloaking-Based Tracking on the Web: Characterization, Detection, and Protection. *IEEE Transactions on Network and Service Management* (2021).
[62] Sean M Devine and Nathaniel D Bastian. 2021. An Adversarial Training Based Machine Learning Approach to Malware Classification under Adversarial Conditions.. In *HICSS*. 1–10.
[63] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. 2021. The cname of the game: Large-scale analysis of dns-based tracking evasion. *Proceedings on Privacy Enhancing Technologies* 2021, 3 (2021), 394–412.
[64] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. 2021. The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion. *PETS* (2021).
[65] Muhammad Ikram, Hassan Jameel Asghar, Mohamed Ali Kaafar, Balachander Krishnamurthy, and Anirban Mahanti. 2016. Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning.
[66] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. 2017. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *Proceedings of the 2017 Internet Measurement Conference*. Association for Computing Machinery.
[67] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2020. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. IEEE.
[68] 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.

[69] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium (NDSS 2019)*. https://doi.org/10.14722/ndss.2019.23386
[70] Su-Chin Lin, Kai-Hsiang Chou, Yen Chen, Hsu-Chun Hsiao, Darion Cassel, Lujo Bauer, and Limin Jia. 2022. Investigating Advertisers' Domain-changing Behaviors and Their Impacts on Ad-blocker Filter Lists. In *Proceedings of the ACM Web Conference 2022*. 576–587.
[71] Jonathan R Mayer and John C Mitchell. 2012. Third-party web tracking: Policy and technology. In *2012 IEEE symposium on security and privacy*. IEEE, 413–427.
[72] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. 2014. Guided mutation testing for javascript web applications. *IEEE Transactions on Software Engineering* 41, 5 (2014), 429–444.
[73] Muhammad Haris Mughees, Zhiyun Qian, Zubair Shafiq, Karishma Dash, and Pan Hui. 2016. A first look at ad-block detection: A new arms race on the web. *arXiv preprint arXiv:1605.05841* (2016).
[74] Shaoor Munir, Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. 2023. COOKIEGRAPH: Understanding and Detecting First-Party Tracking Cookies. arXiv:2208.12370 [cs.CR]
[75] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. Using Dynamic Analysis to Generate Disjunctive Invariants. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 608–619. https://doi.org/10.1145/2568225.2568275
[76] Fei Peng, Zhui Deng, Xiangyu Zhang, Dongyan Xu, Zhiqiang Lin, and Zhendong Su. 2014. {X-Force}:{Force-Executing} Binary Programs for Security Applications. In *23rd USENIX Security Symposium (USENIX Security 14)*. 829–844.
[77] Yan Qin, Weiping Wang, Zixian Chen, Hong Song, and Shigeng Zhang. 2023. TransAST: A Machine Translation-Based Approach for Obfuscated Malicious JavaScript Detection. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 327–338.
[78] Kunlun Ren, Weizhong Qiang, Yueming Wu, Yi Zhou, Deqing Zou, and Hai Jin. 2023. An Empirical Study on the Effects of Obfuscation on Static Machine Learning-Based Malicious JavaScript Detectors. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1420–1432.
[79] Iskander Sanchez-Rola, Matteo Dell'Amico, , Davide Balzarotti, Pierre-Antoine Vervier, and Leyla Bilge. 2021. Journey to the center of the cookie ecosystem: Unraveling actors'; roles and relationships. In *S&P 2021, 42nd IEEE Symposium on Security & Privacy, 23-27 May 2021, San Francisco, CA, USA*.
[80] Asuman Senol, Alisha Ukani, Dylan Cutler, and Igor Bilogrevic. 2024. The Double Edged Sword: Identifying Authentication Pages and their Fingerprinting Behavior. In *The Web Conference (WWW), 2024*.
[81] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. 2022. WebGraph: Capturing Advertising and Tracking Information Flows for Robust Blocking. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association.
[82] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. 2020. Filter List Generation for Underserved Regions *(WWW '20)*. Association for Computing Machinery.
[83] 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.
[84] Peter Snyder, Antoine Vastel, and Ben Livshits. 2020. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. *Proc. ACM Meas. Anal. Comput. Syst.* (2020).
[85] Peter Snyder, Antoine Vastel, and Ben Livshits. 2020. Who filters the filters: Understanding the growth, usefulness and efficiency of crowdsourced ad blocking. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 2 (2020), 1–24.
[86] Zhenhao Tang, Juan Zhai, Minxue Pan, Yousra Aafer, Shiqing Ma, Xiangyu Zhang, and Jianhua Zhao. 2018. Dual-force: Understanding webview malware via cross-language forced execution. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 714–725.
[87] Phani Vadrevu and Roberto Perdisci. 2019. What you see is not what you get: Discovering and tracking social engineering attack campaigns. In *Proceedings of the Internet Measurement Conference*. 308–321.
[88] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. 2016. Webranz: web page randomization for better advertisement delivery and web-bot prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 205–216.
[89] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. 2016. WebRanz: Web Page Randomization for Better Advertisement Delivery and Web-Bot Prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Association for Computing Machinery.
[90] Zhi Yue Wang and Wei Min Wu. 2014. Technique of javascript code obfuscation based on control flow tansformations. *Applied Mechanics and Materials* 519 (2014), 391–394.
[91] Shiyi Wei and Barbara G Ryder. 2014. State-sensitive points-to analysis for the dynamic behavior of JavaScript objects. In *ECOOP 2014–Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings 28*. Springer, 1–26.
[92] Shiyi Wei, Franceska Xhakaj, and Barbara G Ryder. 2016. Empirical study of the dynamic behavior of JavaScript objects. *Software: Practice and Experience* 46, 7 (2016), 867–889.
[93] Xiaofei Xie, Bihuan Chen, Yang Liu, Wei Le, and Xiaohong Li. 2016. Proteus: Computing Disjunctive Loop Summary via Path Dependency Analysis. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2950290.2950340
[94] Wei Xu, Fangfang Zhang, and Sencun Zhu. 2012. The power of obfuscation techniques in malicious JavaScript code: A measurement study. In *2012 7th International Conference on Malicious and Unwanted Software*. IEEE, 9–16.

9 Appendix

9.1 Shortcomings in WebGraph's graph representation.

Figure 9 in the extended version [52] shows a graph constructed by WebGraph for the execution flowchart in Figure 5. There are key differences in the graph representation of WebGraph compared to NoT.js. First, nodes in the WebGraph's graph representation are at the script-level instead of the function-level. Thus, in WebGraph, all functions within a script will be assigned the same label and features as the script, leading to over-approximation. Second, WebGraph does not capture the JS execution call stack, missing all the edges between caller-callee scripts, such as the edge between script `main.js` and `app.js`. Third, WebGraph includes two storage APIs (*i.e.,* local storage and cookies) and does not include the rest of the Web APIs that are implemented in NoT.js. It also lacks the calling context of the function when it was called, such as the arguments of the function. As shown earlier, this calling context at finer granularity plays a crucial role in detecting tracking functions. In short, WebGraph's graph representation misses an average of 262.3 nodes and 673.5 edges per webpage which are key to fine-grained detection of tracking JS code.

Actual	Predicted	
	0 (non-tracking)	1 (tracking)
0 (non-tracking)	216,996	8,011
1 (tracking)	2,720	133,325

Table 8: The classification report and corresponding confusion matrix for the NoT.js.

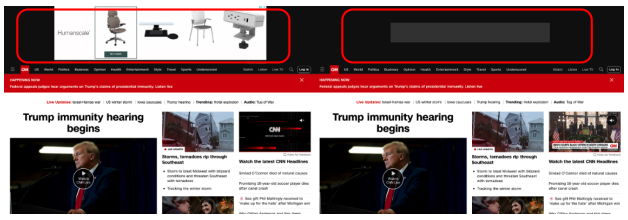


Figure 9: In the appearance category, 4% reported minor breakage attributed to NoT.js is, in fact, not a breakage but simply a leftover overlap after tracking is removed.