Duumviri: Detecting Trackers and Mixed Trackers with a Breakage Detector

He Shuang
University of Toronto
he.shuang@mail.utoronto.ca

Lianying Zhao
Carleton University
lianying.zhao@carleton.ca

David Lie
University of Toronto
david.lie@utoronto.ca

***Abstract*—Web tracking harms user privacy. As a result, the use of tracker detection and blocking tools is a common practice among Internet users. However, no such tool can be perfect, and thus there is a trade-off between avoiding breakage (caused by unintentionally blocking some required functionality) and neglecting to block some trackers. State-of-the-art tools usually rely on user reports and developer effort to detect breakages, which can be broadly categorized into two causes: 1) misidentifying non-trackers as trackers, and 2) blocking mixed trackers which blend tracking with functional components.**

We propose incorporating a machine learning-based breakage detector into the tracker detection pipeline to automatically avoid misidentification of functional resources. For both tracker detection and breakage detection, we propose using differential features that can more clearly elucidate the differences caused by blocking a request. We designed and implemented a prototype of our proposed approach, Duumviri, for non-mixed trackers. We then adopt it to automatically identify mixed trackers, drawing differential features at partial-request granularity.

In the case of non-mixed trackers, evaluating Duumviri on 15K pages shows its ability to replicate the labels of human-generated filter lists, EasyPrivacy, with an accuracy of 97.44%. Through a manual analysis, we find that Duumviri can identify previously unreported trackers and its breakage detector can identify overly strict EasyPrivacy rules that cause breakage. In the case of mixed trackers, Duumviri is the first automated mixed tracker detector, and achieves a lower bound accuracy of 74.19%. Duumviri has enabled us to detect and confirm 22 previously unreported unique trackers and 26 unique mixed trackers.

I. Introduction

Users navigating the web are constantly being monitored. 95% of 21 million pages contain 3rd-party requests to potential trackers [1]. This extensive tracking results in a significant loss of privacy, as it allows users' sensitive information to be used for targeted advertising [2], behavioral profiling, and sold to third parties without their consent [3]. Therefore, there is a need to identify and block trackers to protect users' privacy.

Web trackers come in two types: non-mixed trackers and mixed trackers. Non-mixed trackers send network requests that are purely for the purpose of tracking users. These requests may load tracking code onto the web client or send identifiers and information that enable users to be tracked. Since non-mixed tracker requests only contain tracker data and functionality, they are relatively easy to identify and block. To make the identification and blocking of trackers harder, trackers can be mixed, meaning that requests made by tracking code contain both tracking and legitimate functionality [4], [5]. Most previous work has focused on non-mixed trackers, and has used on network request features [1], [6], as well as both static [7] and dynamic JavaScript features [8], [9] to identify and block such trackers. More recently, there have also been research proposals to deal with mixed trackers by attempting to identify mixed JavasScripts [10] and block the tracking functionality by disabling the tracking components [11].

In general, an effective tracker detector should have two objectives: 1) maximizing privacy by blocking as many trackers as possible and 2) minimizing web page breakage. Breakage can occur due to 1) incorrectly classifying non-tracking components as trackers and blocking them and 2) blocking mixed trackers in their entirety, which ends up blocking their functional component.

Previous approaches train a single model to detect tracking without explicitly addressing the issue of breakage. This approach has two drawbacks: 1) The single tracking model must be highly accurate to avoid misidentification. Blocking a misidentified functional request can result in web page breakage. This is quite common—previous single-model approaches [8] can cause breakage on 15% of the sites. 2) They are unable to block mixed trackers as they classify and block the request in its entirety, and thus are not able to block only the tracking component.

We explore adding explicit breakage detection into tracker detection. We argue that by explicitly considering the breakage introduced by tracker detection, we can minimize the detection imprecision 1) due to misidentification, where the detector made a false prediction and 2) due to mixed trackers, where the detector made a proper prediction, but still breaks a page.

This paper introduces Duumviri, which incorporates two novel mechanisms to address both non-mixed and mixed trackers. First, as implied by its name[1], Duumviri introduces breakage detection into tracker detection pipeline. It uses two models instead of one: one for detecting trackers, and the

Network and Distributed System Security (NDSS) Symposium 2025
23 - 28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.23267
www.ndss-symposium.org

[1]Duumviri is Latin for "two men," denoting a pair of officials sharing power and duty in ancient Rome.

other for detecting which requests that, if blocked, will lead to web page breakage. This breakage detector can thus detect functional requests that were misclassified as tracking for non-mixed trackers, as well as request that contain both functional and tracking functionality for mixed trackers. Duumviri's models work on *differential features*, which are derived from experimentally blocking requests and comparing the resulting page behavior to that of the original page. Using differential not only enables accurate detection of breakage but also enables Duumviri to detect tracking request fields at partial-request granularity, enabling Duumviri to block tracking functionality without blocking legitimate functionality.

In designing Duumviri's breakage detector, we overcame the following challenges: 1) Feature selection. Previous proposals [12] lacked the features for accurate breakage detection. We built our features by comprehensively covering the channels of externally-visible events emitted by a web page during rendering, effectively addressing the symptoms of web page breakages [13]. 2) The lack of a dataset of breakages. It is challenging to find relevant breakage samples on live sites. We solved this by leveraging exception rules in filter lists for up-to-date collection of breakage samples that we reconstructed by "flipping" exception rules into block rules.

This paper makes the following contributions:

- The *design and implementation* of Duumviri. The introduction of the breakage detector flags the breakage caused by misidentification and blocking mixed trackers, increasing overall accuracy. We designed the detectors to use differential features enabling the blocking of request fields to block mixed trackers without causing breakage. We trained the breakage detector by collecting reconstructable breakage samples and the tracking detector for mixed trackers by collecting mixed request trackers from advanced content blockers such as AdGuard and UbO (Ublock Origin).
- An *evaluation of Duumviri on non-mixed trackers identification* on 15K pages. Our results show that Duumviri can reproduce the labels from human-generated filter lists with a 97.44% accuracy. Through our manual analysis of the disagreements between Duumviri and the filter lists, we found 55% of instances to be previously unreported new trackers. In addition, 10% of analyzed cases are filter list-caused web page breakages that Duumviri's breakage detector found. We have reported 22 cases of confirmed new trackers with 175 occurrences in our dataset and 2 instances of confirmed filter list-caused breakage to the community.
- An *evaluation of Duumviri on mixed trackers identification*. We evaluated Duumviri on all resources that filter lists deemed as non-trackers. Through a manual analysis, we found that Duumviri can achieve a lower bound accuracy of 74.19% in detecting tracking fields. In this process, we found 26 mixed trackers with 83 occurrences in our evaluation dataset, which we have reported to the community.

Artifacts. Our artifacts are available on GitHub [2] and further discussed in Appendix §D.

[2]https://github:com/dlgroupuoft/Duumviri-NDSS25

II. Background and Definitions

We give background information on web trackers and web page breakages.

Web Trackers. We adopt a generic definition of web tracking, which is the process of (re-)identifying users in different computation contexts [14]. The computation contexts can be, and are not limited to, user-specific behavior, same-domain pages, browsing modes (e.g., incognito and regular), sites (i.e., cross-site tracking) and devices (i.e., cross-device tracking). The tracking process involves, at a minimum, two stages: 1) storing a server-known identifier on the client or generating an identifier in the computation context and 2) retrieving the identifier or generating the same identifier in a different context.

These two stages of tracking naturally lead to two types of tracking network requests: 1) *incoming tracking responses* that initiate client-side tracking in one computation context. For instance, a response may contain a set-cookie header that stores a server-generated unique identifier for stateful tracking or JavaScript payload that generates fingerprints for stateless tracking. 2) *outgoing tracking requests* that contain privacy-sensitive information (e.g., a user identifier) in a different computation context. Such requests inform the tracking server when a specific action is performed by the user.

Blocking the network requests in either stage (i.e., a tracking response or a tracking request) prevents tracking. Blocking a tracking response prevents storing identifiers or generating identifiers in one context; blocking a tracking request prevents a server from learning that a specific user in a different computation context.

Mixed Trackers. Broadly speaking, a mixed tracker is a tracker that also has legitimate functionality. As the tracking process has two stages, each stage leads to a type of mixed trackers: 1) A mixed response tracker contains a response with mixed tracking and legitimate functionality. For instance, a response may contain JavaScript code that handles web page interaction yet also dynamically generates user fingerprints [5]. 2) A mixed request tracker is a request with request fields (e.g., query string parameters, cookies) that have mixed tracking and legitimate functionality. To illustrate, we provide a real-world example: after clicking on a search result on a popular search engine, the page sends one request (URL shown in Listing 1) that redirects the user to the desired page. In the URL, query string parameters *goods_id* and *sku_id* are functional parameters that redirect to the specific product that the user clicked on, but *_x_ns_msclkid* is a tracking parameter (assigned by Bing) that records an ID of the click. This example mixed request tracker cannot be blocked in its entirety: doing so (or removing any of the functional parameters) breaks redirection as the request will land on a generic page as opposed to the product the user intends to see. However, removing the only tracking parameter while keeping the others stops tracking while preserving the redirection.

Web Page Breakage can occur when 1) privacy developers make mistakes in addressing trackers and 2) erroneously blocking mixed trackers at the request granularity. Human

```
https://www.temu.com/subject/n9/googleshopping-
    landingpage-a-psurl.html?goods_id
    =601099526089385&sku_id=17592258865022&
    _x_ns_msclkid=eeec99c83e911b00583ffc4bc3e34060
```

Listing 1. An example mixed request tracker. Blue indicates functional parameters. Red indicates tracking parameters. Additional parameters omitted for brevity.

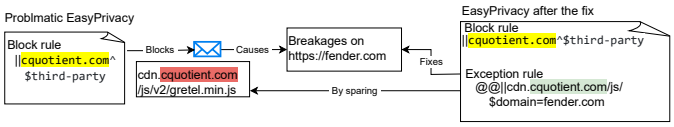

Fig. 1. An example of an exception rule used to 'fix' page breakage. When 'fender.com' fetches 'gretel.min.js' from 'cdn.cquotient.com'. This request is blocked as the domain is listed as a tracking server. However, the particular resource is used for legitimate web page functionality (product recommendation); blocking it causes missing page content. Privacy developers fix this issue by adding an exception rule that makes an exception for 'fender.com' [15].

mistakes can appear in any step of the tracker addressing work flow (e.g., incorrectly identified tracker, an incorrect fix, or failing to identify breakage). The chance of a mistake increases as the number of rules in the filter lists increases. Blocking mixed trackers entirely at the request granularity can lead to web page breakage since the functional request fields are also blocked.

Currently, when a breakage occurs and is experienced by a content blocker user, she reports it to the privacy developers who then address the breakage. While this process can be long and error-prone [16], it is generally difficult for a privacy developer to manually perform breakage evaluation comprehensively due to the lack of domain knowledge on the broken site (e.g., unfamiliarity, site requiring an account for access, site is restricted to certain geographic locations). One common method for fixing breakages is by inserting a new exception rule. The exception rule spares the erroneously blocked content, where we show one example in Fig 1. Such practices can lead to performance degradation due to the number of exception rules and maintainability of the filter lists. An automatic breakage detector can help tracker detection test candidate rules constructed during tracker identification by catching erroneously identified trackers automatically.

III. The Design of Duumviri

We describe Duumviri's design and how it enables detection of non-mixed trackers. We begin with Duumviri's workflow, followed by Duumviri's features and finally, how Duumviri collects its dataset for training detectors.

Workflow. Duumviri's workflow mirrors the approach that one might imagine is taken by a human privacy developer in addressing trackers. With components in Fig 2, given a page-under-analysis (PUA), Duumviri iteratively selects an outgoing request-under-analysis (RUA) and executes the following steps: 1) Differential Page Visits. Duumviri visits the PUA to gather a trace of the page rendering process, denoted T. This trace is used for differential feature extraction and is described in detail in §III-A1. Next, Duumviri revisits PUA with RUA blocked, generating another trace, T_B. Blocking a request means intercepting it and terminating it with a response of status code 403. 2) Differential Feature Extraction. Duumviri compares traces T to T_B producing two sets of differential features $F_{tracker}$ and $F_{breakage}$ for the tracking detector and breakage detector, respectively. 3) Tracker Identification. Duumviri invokes its tracking detector with $F_{tracker}$ to obtain a label indicating whether RUA is a tracker. If RUA is not classified as a tracker, Duumviri skips the next steps and proceeds to the next RUA. 4) Breakage Evaluation. If RUA is classified as a tracker Duumviri invokes its breakage detector with $F_{breakage}$ to obtain a label indicating whether RUA breaks the page when blocked. 5) Automatic Fixing. If the breakage detector does not detect breakage, then Duumviri creates a block rule for future RUA. If the RUA is deemed to be a tracker but also causes breakage when blocked, it is labeled a potential mixed tracker, which is described in §IV.

A. Differential Features

We designed differential features for Duumviri's breakage detector and tracking detector because 1) for the breakage detector, differential features describe the *change* in web page state, which contains more information than just looking at a single page. 2) for the tracking detector, differential features provide accurate attribution of potential tracking activities to the RUA and 3) we can draw differential features from blocking requests for non-mixed tracker analysis as well as from blocking request fields for mixed tracker analysis. Due to space constraints, we present the top 10 most important features of the detectors in Table I along with a high-level categorization.

Breakage Detector Features. We designed breakage detector features by modeling web page breakage. We define web page breakage as changes in browser behavior in at least one externally visible channel compared to the vanilla functional page. This could include the absence of certain user interface (UI) elements affecting appearance or the lack of event listeners causing unresponsiveness. We consider the following externally visible channels: 1) web page appearance for user perception, 2) user input handling for interactiveness, 3) network requests for server-side states, 4) writes to persistent storage for client-side states, and 5) temporal performance for user experience.

We developed a total of 63 differential features. Due to space constraints, we describe the complete features in Appendix. As shown in Table I, 9 out of the 10 most important features are unique to Duumviri. We further show that our breakage detector, which relies on these features, contributes to Duumviri's accuracy through an ablation analysis in §V-B2.

Tracking Detector Features. We designed the differential features for the tracking detector in four broad categories: 1) DOM states capture the change in DOM elements and event listeners. Trackers often rely on tracking pixels to send sensitive information or event listeners to trigger request sending. 2) Requests capture communication between a tracker and the remote server, as trackers usually rely on network requests to share sensitive information like user identifiers. 3) JavaScript control flow captures unique tracking activities, such as invoking high-entropy APIs, which differ from functional scripts.

Fig. 2. 1) Duumviri visits a page using two instrumented browser instances capable of produce a rendering trace. Both instances share a network cache. 2) Duumviri conducts differential analysis on the page instances and draws differential features independently for its detectors. 3) the detectors take the features and make predictions. The predictions determine if the potential tracker is added to the lter list.

TABLE I
TOP-10 MOST IMPORTANT FEATURES OF DUUMVIRI'S DETECTORS "IMPORTANCE" SHOWS THE PERMUTATION FEATURE IMPORTANCE USING ACCURACY AS THE METRIC, REPEATED 5 TIMES. "UNIQ" ILLUSTRATES WHETHER THE FEATURE IS PROPOSED BY AND UNIQUE TO DUUMVIRI COMPARED TO PREVIOUS WORKS IN THE AREA

Breakage Detector			Tracking Detector		
Feature	Importance	Uniq	Feature	Importance	Uniq
console logs	3.2	X	parameters of the blocked request	14.8	
page load time	2.81	X	URL length of the blocked request	8.08	
event listeners	2.25	X	response size of the blocked request	3.87	
cookies values	1.07	X	times rst party appear in the blocked request	3.42	
document height	0.79	X	`eval' appear in the response of the blocked request	3.23	
CSS classes	0.73	X	high entropy ngerprinting function executed	0.54	
DOM tree	0.38	X	third party requests blocked	1.03	
listeners on interactable elements	0.35	X	requests blocked	0.3	
HTML tag sequences	0.33		third party requests with sensitive information	0.29	X
full-paged screenshot as a feature vec	0.31	X	`eval' in the ancestors nodes of the blocked request	0.24	

4) Data ow features capture information ow, such as cookies, from one actor (e.g., a network request) to local storage or other actors (e.g., scripts), which is crucial as much tracking information (e.g., user identi ers) must be shared with remote servers to complete tracking.

In the rest of this section, we will detail the trace, followed by how differential features are constructed, and nally, we discuss how we ensure reliable features extraction (e.g., server-side randomness, session-speci c requests).

1) Traces: The traces that Duumviri collects are used for building differential features. Each trace contains the following components: 1) Requests: This component captures the direction, timing, and content (i.e., headers, body) of all network requests during page rendering. Duumviri uses a man-in-the-middle proxy to intercept all requests in decrypted form, enabling access to request plaintext. 2) DOM elements: This component captures the raw DOM elements that are of interest to us. We currently track elements that we believe have a correlation to tracking or breakage, including canvas, audio, buttons, input, span, video, image, script, and hyperlink tags. For each element, we track the layout, position, and content. 3) Events: We track DevTool events such as when a page downloads (downloadWillBegin) and when a page nishes loading (loadComplete). 4) Event listeners: This component tracks all event listeners, including listener type, details (whether it is passive or red once), the target object (e.g., a button), and the event handler (e.g., JavaScript text). 5) Scripts: This component tracks all parsed JavaScript by V8, including external and inline scripts. For both types, we track the script's textual content, position and the source URL if it is an external script. 6) Appearance: This component tracks information related to page appearance. We track a "long" screenshot covering all visual elements, the list of loaded fonts, the list of colors used on all HTML elements, the inner text, the main text of the page [17], and all CSS styles. 7) Storage values: This component describes the storage values including cookies, session, and local storage. 8) Console log: This component contains the console logs that a page prints. We track the level (e.g., serve, message), time, source, and the actual log message. 9) Perceptual adblocker: This component tracks the number of ads found by AdHighlighter [18]. 10) PageGraph: This component is the complete PageGraph [19]. PageGraph is a graph representation of the page rendering process, including nodes describing entities (e.g., HTML element, web resources, JavaScript les), and edges describing the actions (e.g., resource loading, DOM modi cations) by the web entities.

2) Differential Features Construction: Given two traces T_A and T_B, Duumviri produces the differential features based on the type of trace component. A differential feature describes the degree of differences between two trace components. The goal of feature construction is to obtain accurate and consistent differences. Duumviri has two default methods for extracting coarse-grained differences: hamming distance and cosine similarity. While they work well for features such as fonts and colors where we only need to know the degree of differences, they are not suitable for complex comparisons on components such as requests and event listeners. Below, we will describe the reasons and our solution: 1) Requests. Network requests

can be session-specific (e.g., random values for cache busting [20]). However, we need to consider requests of the same origin as identical for accurate feature extraction. To do this, Duumviri implements fuzzy request matching based on request initiator, type (e.g., POST), URL, headers and body. Specifically, we consider two requests to be the same if the request initiator and type are identical, and the URL, headers and body have over 95% similarity as measured by cosine similarity.

2) DOM elements. Similar to requests, the element may be session-specific (e.g., dynamically generated attributes). Duumviri considers two DOM elements to be the same based on a combination of the structural and stylistic similarity as described in [21]. 3) Scripts. We use cosine similarity to measure the similarities among JavaScript text after vectoring the JavaScripts into token counts. 4) Event listeners. We consider two listeners the same if the type of the event, target object (a DOM element), and handler (a JavaScript function) are all identical. 5) Appearance. A common method for comparing screenshots is pixel-based similarity [10], [22]. However, we found this method to be unreliable for two reasons: 1) benign changes may occur due to the non-deterministic behavior of web pages (which we detail below), and 2) not all visual changes equally contribute to breakage. For example, a slight change in the position of an image could result in a large percentage of pixels being different. Instead of directly comparing pixels, Duumviri compares the feature vectors of the screenshots using a pre-trained EfficientNet model [23]. We chose a vision-based model because it mimics what a human user perceives. And, EfficientNet has demonstrated generalization capabilities that often extract semantically meaningful features. By comparing the feature vectors, we can assess "how semantically similar the two screenshots are as perceived by a human user", which, based on our experience, has a higher correlation to breakage than pure pixel-based similarity measurement. 6) PageGraph. We do not measure the similarity of page graphs at the graph granularity. We derive lower-level features as used in AdGraph [8] and WebGraph [9] and draw differential features using the default methods on those features, which we describe in Appendix.

3) Obtaining Reliable Features. One issue in generating differential features is the non-deterministic behavior of web pages, which can arise from various factors involved in the web page rendering process, such as server-state (e.g., network responses), client-state (e.g., existing cookies), and characteristics (e.g., user agent), as well as the web page content itself (e.g., time-sensitive content). This non-deterministic behavior introduces noise into the differential features that are not caused by blocking the request, increasing the likelihood of the breakage detector mistakenly predicting breakage. Such erroneous breakage predictions then decrease Duumviri's overall accuracy by incorrectly identifying trackers as non-trackers. To mitigate this impact on accuracy, Duumviri employs two techniques: minimizing web page non-deterministic behavior through a specialized rendering environment and rendering each page a configurable number (k) times. While we provide details on our rendering environment below, rendering a single page k times results in k vanilla page traces and k page traces of T_B with a blocked request-under-analysis ($T_{B'}$), totaling k^2 pair-wise differences. Given that non-deterministic behavior occurs less frequently than normal behavior, Duumviri retains the most commonly occurring differential features out of all k^2 differences. To illustrate this technique with an example, when visiting colgate.com, a survey dialog may randomly appear, prompting for feedback. If this dialog appears during the differential feature extraction process, Duumviri may incorrectly attribute differences in the user interface to the blocked request. However, since this dialog does not appear most of the time, by visiting the page multiple times, we obtain the majority of differences without the survey dialog. As a result, Duumviri can accurately extract the differences caused by the request.

We explain Duumviri's specialized rendering environment that minimizes web page non-deterministic behavior as follows. 1) Specialized Environment: Servers may react based on client states such as existing cookies, user agents (e.g., desktop, mobile), window sizes, and client locations (i.e., IP addresses). Duumviri ensures a consistent, clean (e.g., no cookies), and plausible (like that used by human users) pre-loading environment. 2) Network Cache: Some servers may send back non-deterministic responses, such as third-party ads. This non-determinism can affect page appearance. To deal with this, Duumviri uses a network cache to record the communication (e.g., requests and responses) to the server when rendering the vanilla page and replays the same response when rendering the second page if an identical request passes through the cache. 3) Dynaminism Mask: There may be non-deterministic HTML elements such as time-sensitive elements (e.g., slide shows, videos). We observed that the layout of the page is usually deterministic; we use the following method to address non-deterministic element appearance: for every screenshot, Duumviri takes another screenshot after a configurable amount of time. The difference between the two screenshots forms a dynamism mask — these are the time-sensitive and non-deterministic areas. These areas are excluded from appearance comparison during feature extraction. Still, Duumviri cannot and does not aim to eliminate all non-deterministic behavior; it will benefit from continued progress in this area [24].

TABLE II
THE TRAINING SIZE, TEST ACCURACY, AND CROSS-VALIDATION (5 FOLDS) ACCURACY OF THE DETECTORS

Detector	Training Size	Test Size	Test Accuracy	Cross-validation F1	Cross-validation STD
Breakage	15,854	3,171	98.30	0.9591	0.0028
Tracker	27,721	5,545	93.62	0.9268	0.0026
Mixed Tracker	1,976	396	85.10	0.8499	0.1923

B. Training Dataset

1) Breakage detector. One problem in training the breakage detector is the lack of positive data points—i.e., samples of real breakage. While previous work reconstructs breakage from commit messages [12], our evaluation shows that it is an unreliable source (§V-A2). Instead, Duumviri gathers breakage samples from the exception rules in the current filter

lists. While exception rules are used by privacy developers to temporarily “ x” broken pages (§II and Fig 1), they have two practical advantages 1) exception rules are constantly validated by lter list developers to ensure relevancy—irrelevant rule that no longer x breakage are removed by developers a 2) exception rules are written and tested by developers exclude the exact resource that causes breakage. We dis these points in more detail in §V-A.

Duumviri reconstructs a breakage by pping an exception rule into a block rule. This process blocks the exact resour whose blocking causes breakage and it reconstructs the exact breakage that the privacy developer faced before adding the exception rule. Speci cally, Duumviri: 1) ips the exception rule to a block rule 2) navigates to the URL indicated by the domain modi er associated with the rule and 3) generates differential features by comparing the vanilla page and the resulting page.

Not all exception rules can reconstruct the breakage. While developers do monitor these rules, some can still inevitably be come stale (e.g., page has changed but the rule is not updated). In addition, the domain speci er can be ambiguous (e.g., the speci er points to a domain, but the breakage only occurs on a speci c page within that domain). When reconstructing breakage, Duumviri identi es such cases by monitoring the number of resources blocked by the ipped rule. Flipped exception rules that do not block any resources are discarded. We demonstrate the effectiveness of this detection in §V-A1. We gathered a total of 13,921 exception rules and constructed 2,308 breakages, which were sampled to ensure correctness.

For the non-breakage samples, we need page changes that do not disrupt functionality. We want the breakage detector to “accept” legitimate changes. One of the legitimate changes is the blocking of trackers. Thus, we draw differential features typical to blocking trackers using EasyList and EasyPrivacy as the ground truth. While these lter lists themselves may be inaccurate and cause breakage, we only collect non-breakage samples from the top 5K sites from the Alexa Top List, generated in May 2022. Popular sites tend to have issues discovered and resolved more quickly due to their large visitor numbers.

The breakage detector was trained on 15,854 data points, comprising 2,308 reconstructed breakages making up the positive label (quality evaluated in §V-A1) and 13,546 cases of regular trackers as the negative label. We use an XGBoost [25] model.

2) Tracking detector:We train our detectors using requests encountered while crawling[3] the top 5K sites from the Alexa Top 1M List generated in May 2022. Using EasyList and EasyPrivacy as ground truth, we have 12,936 (46.66%) cases of trackers and 14,785 (53.34%) cases of non-trackers. We also use XGBoost model.

IV. Adapting Duumviri for Mixed Trackers

In this section, we discuss how we adopt Duumviri to automatically identify mixed trackers. Based on the de nition of tracking from §II, we now describe a baseline model of how users deal with mixed trackers. Filter lists such as EasyList and EasyPrivacy cannot deal with mixed trackers directly. They check resources at request granularity: a request is either blocked or spared entirely. Such a crude decision implies a lose-lose situation for mixed trackers: blocking a mixed tracker entirely potentially breaks the web page due to the blocking of the functional request elds; sparing a mixed tracker hurts user privacy. Advanced content blockers, such as UbO (Ublock Origin) and AdGuard, perform analysis at partial-request granularity. A request eld is a component of a HTTP request whose content is application-de ned. Instances of request elds include query string parameters, request headers and body. UbO and AdGuard have added the ability to inspect and alter individual request elds, enabling them to remove only the request elds of mixed trackers, thus preserving website functionality. We list the rules in UbO and AdGuard that are capable of addressing mixed trackers including the type of mixed trackers they address in the rst two columns of Table III.

TABLE III
Statistics on currently identified mixed trackers on top 15K sites. [1] a UbO rule. [2] an AdGuard rule.

Type	Rule Type	# Rules	# Instances	# Uniq	# Domains
Mixed Response	redirect[1,2]	1,914	9,052	7,214	4,197
	replace[2]	718	52	48	18
	empty[1]	5	0	0	0
	jsonprune[2]	26	0	0	0
	hls[2]	2	0	0	0
Mixed Request	removeparam[1,2]	130	578	485	357
	removeheader[2]	2	0	0	0
	cookie[2]	717	108,491	98,923	4,904

Prevalence of Mixed Trackers. Using these rules, we conducted a prevalence study on both types of mixed trackers. First, we gathered lter lists from content blockers compatible with UbO and AdGuard's rule syntax, including UbO, AdGuard, Ad lt, and ClearURLs. All rules utilizing this syntax as of March 2024 were collated and applied to the traf c encountered during the crawl of 15K pages in §V-B2. We collected tuples of (domain, requests, and applied rules) for cases agged by any rule and exclude requests that are blocked entirely by UbO or AdGuard, as these are likely non-mixed trackers. The results, presented in the last ve columns of Table III, reveal that the majority of mixed trackers fall under the mixed request type. It is worth noting that, as these rules can address a broader range of undesired information leakage within requests beyond trackers (e.g., performance measurement), the statistics collected may overestimate the real number of trackers. To reduce the overestimation, the lter lists we used are from privacy-focused content blockers.

In this work, we thus focus on blocking mixed trackers by blocking mixed requests—that is requests that contain both functional and tracker parameters. The reasoning for this is that 1) mixed trackers are prevalent as shown by Table III, and 2) blocking either one of mixed requests and mixed responses can equally prevent tracking by mixed trackers. The downside of this approach is that mixed response trackers still execute on the client machine, potentially wasting computation resources.

[3]We use the term `crawl' in this paper to mean visiting the URL and wait for page to load.

We believe that Duumviri's method can be extended to identify mixed response trackers, but leave an exploration of this to future work.

In the current implementation, Duumviri performs analysis on the request field by assuming that tracking information and functional information reside in different request fields. This assumption is reasonable as making each request field either for tracking or other functionality eases server-side request field parsing, and the fact that there is no known tracker that mixes tracking and functional information within a single request field. In fact, there is a proposal to extend UbO to block at sub-field granularity [26], but it has not been implemented due to the lack of evidence of need for this feature. Nonetheless, we designed Duumviri to be extensible to conduct analysis at partial-request field granularity, provided that methods for separating tracking information from functional information within a mixed request field are available. For instance, assuming one POST request body is mixed as proposed in [26], one separation method is to use content type-specific parsing: using regular URL parsing for content type application/x-www-form-urlencoded and JSON parsing for application/json We leave the implementation of this separation method as future work.

Workflow. Given an FUA, Duumviri detects whether it contains tracking information on a page-under-analysis (PUA) by performing the following steps.

1) Differential Page Visits. Duumviri renders the vanilla page once to produce a page trace (T). It then revisits the PUA, and intercepts the request containing FUA and modifies the request by blocking FUA from it. This modified request is sent to the server, whose response (from the server) is sent back to the page. This process produces another trace (T_B).
2) Feature Extraction and Identification. Duumviri extracts the differential features $F_{tracker}$ and $F_{breakage}$, from the traces T and T_B. Duumviri then classifies whether a particular FUA is used for tracking or not by invoking tracking detector with $F_{tracker}$. Similarly, it infers whether the FUA breaks the page when blocked by invoking breakage detector with $F_{berakage}$.
3) Fixing. For all FUAs that are used for tracking, Duumviri automatically creates filter rules blocking them in UbO's syntax: `removeparam` and `cookie` syntax, preventing tracking. If a FUA is both tracking and breaks the page when blocked, it means the FUA contains both tracking and functional information not separated by request structure violating Duumviri's assumption. We leave such cases as non-tracker as Duumviri cannot currently handle such a case.

Training Dataset. We trained a new tracking detector for mixed trackers using the mixed trackers found using existing filter lists and tools in the prevalence study in §IV, as this identified specific requests fields that privacy developers have labeled as being used for tracking. We use them as a ground-truth for mixed tracking request fields. Although they may contain noise as discussed in §II, they are the best mixed-request tracker samples we can find. Specifically, we collected 71 and 905 cases of tracking parameters and cookies totaling 976 instances of tracking request fields. We randomly sampled 1,000 cases of non-tracking request fields from two sources: 1) functional parameters from the request not blocked by EasyPrivacy and 2) functional cookies as indicated by Cookiepedia [27]. This process yields a balanced dataset of 1,976 data points in total. We again use an XGBoost model. We report the test accuracy in Table II. Due to the lack of the breakage specifically caused by mixed trackers, we did not train a separate breakage detector for mixed trackers and use the same breakage detector we used for non-mixed trackers.

TABLE IV
RESULT OF OF BREAKAGE RECONSTRUCTABILITY EVALUATION

Description	Exception rules	Commit message
Reconstructable&properly reconstructed	22	15
Reconstructable&incorrectly reconstructe(	0	6
Non-reconstructable	18	19

V. EVALUATION

We demonstrate the effectiveness of Duumviri's breakage reconstruction method and its ability to handle both non-mixed and mixed trackers.

A. Breakage Reconstructability

We aim to answer the following research questions

- Q1: Can Duumviri's method of flipping exception rules reconstruct breakages? (§V-A1)
- Q2: How does Duumviri's method of breakage reconstruction compare to previous works? (§V-A2)

1) Q1: Exception Rule-based Breakage Reconstructability: We conducted an experiment to determine during Duumviri's training set construction: 1) do the reconstruction heuristics properly reconstruct breakage? 2) What percentage of breakages are non-reconstructable by Duumviri's training process? Evaluation Dataset. We reconstruct breakages based on user reports containing ground truth descriptions. Our dataset was constructed by: 1) Finding all exception rules in EasyList, uBlock Origin, and AdGuard repositories as of June 1, 2024, and 2) Filtering out rules not referenced by user reports created between January 1 and May 31, 2024. We randomly sampled 40 rules from 142 exception rules to form our evaluation dataset.

Like previous work [12], we consider breakage to be "reconstructable" if the flipped commit blocks at least one resource. However, just because a resource is blocked, this does not mean that the page is necessarily broken, as the page may continue to work even if some resource is blocked. Thus, we also check whether breakage is properly reconstructed. Our results in Table IV show that: 1) All breakages that are reconstructable by Duumviri are also properly reconstructed as they matched user report descriptions, and 2) Duumviri found 45% of the breakages to be non-reconstructable.

2) Q2:Comparison to Commit Messages:We compare our method of breakage reconstruction with that of [12] (referred to as BoB thereafter), which uses commit messages as opposed to exception rules. BoB begins by collecting breakage- xing commits as determined by commit messages, and attempts “ ip” such commit into changes that cause breakage. Duumviri is not directly comparable with BoB as they operate on different subjects: Duumviri works with current exception rules in lter lists, while BoB works with historic commit messages. Still, we implemented BoB for comparison to answer the same questions in the previous evaluation: 1) when BoB indicates a breakage is reconstructable, does it properly reconstruct it? and 2) what percentage of breakages are non-reconstructable?

Evaluation Dataset. We used the same method to collect breakage- xing commits as described in [12], ltering for commits referenced by user reports in EasyList, uBlock Origin, and AdGuard repositories between July 1, 2023, and May 31, 2024. This ltering was necessary for veri cation, as user reports provide the ground truth breakage descriptions. We randomly sampled 40 commits from 41 for our evaluation dataset.

Our results in the third column of Table IV show 6 cases where BoB indicated that a breakage occurred (i.e., was successfully reconstructed) but our manual investigation did not observe the same symptom as that described in the user reports. Such cases create noise in the training data. We manually investigated the causes of failure. We found that for 4 cases, the page changed between the breakage reconstruction time and breakage-reporting time (by comparing the current version with the closest version before the breakage-reporting time, using Wayback Machine [28]). This means that, at breakage reconstruction time, the web pages no longer have any breakage and successfully blocking the breakage- xing resource no longer reproduces the breakage. We were unable to con rm the remaining two cases. However, the root cause of our inability to properly reconstruct breakage is that commits cannot be removed from git repositories, and so they cannot take into account changes to the web page after the commit is made. The exception rule-based method does not suffer from this as rules can be updated—exception rules that do not trigger breakages tend to be actively removed by developers.

During our evaluation, we found two general issues with BoB that, although they did not lead to incorrect reconstructions in our sample, they could still have potentially introduced noise: 1) Fixes requiring multiple commits: this occurs when a breakage is xed with multiple commits rather than a single commit. This can happen when an initial breakage- xing commit did not fully x the breakage or caused other breakages, hence the need for subsequent commits. We observed 6 such cases in our sample. Multiple commits directly contradicts with BoB's assumption that a breakage- xing commit contains all necessary changes to x a breakage and ipping them reconstructs the breakage, leading to incorrect reconstruction. 2) Tangled commits: a tangled breakage- xing commit includes both relevant and irrelevant changes for xing the breakage. We observed 3 such cases in our sample. Flipping tangled commits can lead to noise in the reconstructed breakage due to the irrelevant changes.

These issues arise because a commit is not always a good representation of breakage- xing changes. A single commit may contain more changes than necessary to x a breakage (tangled commits) or insuf cient changes (multiple commits). In comparison, exception rules represent the exact changes to x a breakage, as developers use them to specify the exact resource necessary for xing breakages, avoid these issues.

TABLE V
SUMMARY OF THE CRAWL USED TO EVALUATE DUUMVIRI ON NON-MIXED TRACKER IDENTIFICATION. D STANDS FOR DUUMVIRI , A STANDS FOR ADGRAPH.

	Filter lists eval (§V-B2)	AdGraph eval (§V-B3)	
Measurement period	Oct 2023	June-July 2024	
Crawl list	Alexa Top 1M	Tranco List	
# pages analyzed	6,489	D:2,645	A:2,335
Size on disk (compressed	3.2 TB	D:1.2GB	A:154MB
Avg # requests of a page	142.13	D:133.48	A:114

B. Non-mixed Trackers Detection

In non-mixed tracker evaluation, we answer the following questions:

- Q1: Do the detectors have predictive power? (§V-B1)
- Q2: As a base test, how does Duumviri's accuracy compare to manually constructed lter lists such as EasyList and EasyPrivacy? Can it identify additional trackers unreported on the lter lists? Can it identify breakage caused by lter lists? (§V-B2)
- Q3: Does Duumviri exceed the state-of-the-art non-mixed tracker identi cation work? (§V-B3)
- Q4: Other than discovering new trackers, how does Duumviri bene t the content blocker community? (§V-B4)

1) Q1: Detector Accuracy:We perform standard 5-fold cross-validation on the detectors to establish baseline classi cation accuracy on the training set. We report the mean F1 scores and standard deviations in Table II. With mean F1 scores of 0.9591 and 0.9268 for the breakage and tracking detectors, respectively, we conclude that the models are correctly trained and have predictive power.

2) Q2: Comparison to Filter Lists:One problem with using lter lists as the ground truth for comparison is that they are imperfect: they may miss trackers and cause breakages. Thus, we rst compute a tentative accuracy that does not account for cases where Duumviri is correct and lter lists are wrong. We then perform a disagreement analysis where we manually inspect and assign a label to a sample of the disagreements between Duumviri and lter lists. Using the disagreement analysis result, we can calculate an adjusted accuracy accounting for lter lists' inaccuracies. Finally, we perform an ablation analysis by running Duumviri's detectors alone. We show that the breakage detector is essential for Duumviri's overall accuracy. We note that since EasyList and EasyPrivacy are not capable of handling mixed trackers and thus will not block any mixed trackers. In this evaluation, we treat any identi ed

mixed tracker as non-tracker so it is in line with lter lists' labels. We evaluate Duumviri's capability on mixed tracker identi cation in §V-B4.

Dataset. We construct our evaluation dataset using requests observed while crawling 15K pages from each of the top, middle, and bottom 5K of the Alexa Top 1M List generated in May 2022, covering a variety of tracking methods. We label requests by invoking Duumviri on all requests fetching JavaScript or containing request parameters as they may transmit tracking identi ers. In total, we analyzed 53,217 requests. The crawling statistics are summarized in Table V

Tentative Accuracy. By comparing Duumviri's labels to lter lists' labels, we calculated an accuracy of 96.53%. This is a tentative accuracy as it may include cases where Duumviri is correct and lter lists are wrong. There are 1,905 disagreements: 898 “raw false positives” and 1,007 “raw false negatives”. We use the term `raw' as the results are pending manual investigation.

Disagreement Analysis. We manually label the disagreements. We nd that Duumviri is capable of identifying trackers missed by lter lists and instances where lter lists causes page breakage. Using the ratio of true positives and true negatives in the analyzed disagreements, we estimate Duumviri's adjusted accuracy to be 97.44%.

Our goal is to assign a request to one of following labels below: 1) Breakage: the request serves functionality and removing it causes site breakage, 2) Stale request: the request serves functionality yet removing it does not cause immediate breakage, 3) Tracker: the request is a tracker, and removing it does not lead to breakage. 4) Undecided: we cannot decide if the request is a tracker or part of the functionality. For mixed tracker, where the request is a tracker, but its removal breaks the page, due to the lack of labels in EasyPrivacy, we treat such cases as breakage.

Due to the dataset's size, we cannot analyze all disagreements. We sample 40 cases from each type of disagreement, totaling 80 cases for manual labeling.

Methodology. We describe our methodology for assigning labels. To determine if a request is a tracker, we gather information from online documentation, the request's initiator, URL, headers, and response. First, we search for any of cial documentation of the request. Tracking requests, especially third-party ones, often have of cial documentation (e.g., Twitter [29], Adobe [30]). We label a request as a tracker if the documentation clearly states its tracking purpose. For requests without of cial documentation, we look for discussions or consensus on similar requests and adopt the agreed-upon decision. For instance, Google's ge?204 [31] is generally recognized as a tracker. We refrain from labeling a request as a tracker if documentation or discussions indicate its involvement in A/B testing, integration, or optimizations, as we found in three cases. Next, we examine the request's initiator. If the initiator is a known tracking script, we deem the request to be tracking. We also look for tracking-related keywords such as `track' and `analytics'. If a request has a response, we inspect the response content. Speci cally, we read le-level header comments in JavaScript responses for references to documentation that describe the le's purpose (see Fig 5 in Appendix C for an example). We also manually read unobfuscated or non-minimized code, determining the request as tracking if it involves fetching or sharing data with known tracking URLs. Finally, we check other resources hosted on the same domain using DuckDuckGo's Tracker Radar Wiki [32] for potential clues. If the label remains undetermined after these steps, we categorize the request as undecided.

To determine if a request is legitimately required for functionality, we follow this procedure: We search for documentation related to the request. Many legitimate requests are self-explanatory and well-documented. We label a request as legitimate if we nd a documentation page that exactly describes it. If documentation is unavailable, we perform a differential analysis to infer the request's purpose. We use two page instances: blocking the request in one and leaving it untouched in the other. We compare the behavior of the two instances, noting any differences that impact our ability to use the page. Speci cally, we observe for visual breakages, such as missing content, iframes, images, and text. We also interact with the pages by scrolling, clicking links and buttons, resizing, hovering, and providing inputs to test input suggestions. We check if hyperlinks work but do not check the referenced page. If the request has a response, we examine it for hints on how the page may be impacted. For instance, if the response JavaScript interacts with a share button, we check if that share button on the page is broken. We determine that a request is legitimately required for functionality if there is a clear association between the request and changes in page behavior. If the purpose of the request remains indeterminate after all procedures, we designate it as undecided.

Raw False Positives. We rst present the results of analyzing “raw false positives”, where Duumviri labels a request as a tracker but lter lists does not, summarized in the rst four columns of Table VI. In these cases, Duumviri was correct 55% of the time. Through reporting newly discovered trackers to EasyPrivacy, we found that 30% of these cases were new trackers not previously reported, and 25% of the trackers could not be reproduced with lter lists enabled. We have two explanations: 1) probabilistic requests that do not always transmit, and 2) requests dependent on existing known trackers. All con rmed new trackers were reported to privacy developers, who added corresponding rules to lter lists. Non-reproducible trackers were not reported to avoid wasting the community's effort.

Raw False Negatives Analysis. Next, we analyze “raw false negatives”, where Duumviri labels a request as non-tracker but lter lists labels it as a tracker, summarized in the last three columns of Table VI. In these cases, 70% were actual trackers, contributing to a high false negative rate. This is not surprising, as lter lists is used by billions of users, and breakages are promptly reported and xed. We found that the false negatives were due to higher-than-expected predictions by the breakage detector, mistakenly labeling requests as legitimate when they aren't. Despite this, Duumviri's low false positive rate suggests

TABLE VI
MANUAL ANALYSIS RESULTS FOR A SAMPLE OF "RAW FALSE POSITIVES" (DUUMVIRI LABELED AS TRACKERS AND FILTER LISTS LABELED AS NON-TRACKERS) AND "RAW FALSE NEGATIVES" (DUUMVIRI LABELED AS NON-TRACKERS AND FILTER LISTS LABELED AS TRACKERS).

	False Positives			False Negatives		
	#	%	Estimated	#	%	Estimated
Tracker	22	55	494	28	70	705
Breakage	4	10	90	2	5	50
Stale requests	8	20	180	3	7.5	76
Undecided	6	15	135	7	17.5	176

TABLE VII
DUUMVIRI'S ACCURACY NUMBERS

Description	Accuracy (%)
Tentative accuracy	96.53
Adjusted accuracy	97.44
Tracking detector only (ablation analysis §V-B2)	94.34

that Duumviri and filter lists find different sets of trackers and can complement each other. Additionally, due to the breakage detector, Duumviri identified instances of filter lists-caused page breakages, detailed in §V-B2.

Adjusted Accuracy. Based on the numbers in Table VI, we calculate an upper bound for false positives as the sum of breakage, stale requests, and undecided cases, totaling 405 (90+180+135). The upper bound for false negatives, counting all known trackers, stale requests, and undecided cases, is 957 (705+76+176). We estimate Duumviri's accuracy rate to be 97.44% (((22839+494)+(28473+50))/53217), referred to as the post-adjusted accuracy, and an F1 score of 0.9716. Duumviri's accuracy numbers are shown in Table VII, and we compare Duumviri's accuracy to the state-of-the-art in §V-B3.

Duumviri Findings. In this section, we detail Duumviri's findings. We identified 22 new trackers, with a total of 175 occurrences in our evaluation dataset, indicating that these trackers are not rare.

We also detail two cases of EasyPrivacy-caused page breakages. One case involves eroadvertising.com, where the script www.eroadvertising.com/js/controllers.js is blocked by the EasyPrivacy rule /eroadvertising. This overly generalized rule blocks the functional script controllers.js, which loads the main body content on the site, resulting in a broken page missing the main body content, as shown in Fig 3 in Appendix C.

Another example is on seznam.cz where the script ssp.seznam.cz/static/js/ssp.js?nocache=1 is blocked by the EasyPrivacy rule ||ssp.seznam.cz^. This rule, designed to block trackers from ssp.seznam.cz, overly generalizes and blocks ssp.js, responsible for loading the cookie consent dialog. With EasyPrivacy on, the user does not see this dialog.

Ablation Analysis. In this analysis, we demonstrate that Duumviri cannot achieve its accuracy without the breakage detector. We calculate Duumviri's accuracy using only its tracking detector, tabulated in Table VII. The results show that Duumviri achieves its highest accuracy through a combination of the two detectors.

To confirm the breakage detector's role in increasing accuracy, we manually analyzed the top 20 cases with the highest breakage detector predictions. Our analysis shows that 7 requests fetch general-purpose JavaScript libraries, 7 fetch JavaScript with specific functionality, such as push notifications and font-loading, and 6 are responsible for page content. We confirmed that blocking these requests leads to missing content, ranging from icons to sub-documents. We could not confirm the remaining two cases. This study shows that all analyzed requests are functional resources, indicating the breakage detector's prediction power for detecting web page functionality resources. When used with the tracking detector, the breakage detector can correct tracking detector's mispredictions on functional resources.

TABLE VIII
COMPARE DUUMVIRI TO ADGRAPH [8] ON 5K SITES FROM TRANCO LIST

Metrics	AdGraph	Duumviri
AuROC	0.9669	0.9682
Accuracy (%)	93.51	93.85
Precision (%)	89.46	88.97
Recall (%)	67.74	83.13

3) Q3: Comparison to the State-of-the-art. We compared the accuracy of Duumviri with AdGraph in classifying web requests as tracking or non-tracking. We selected AdGraph over other works [9], [33], because it was the only one we could execute successfully. Note that we used Duumviri's previously trained model (as mentioned in §III-B) for this evaluation.

We begin by describing our evaluation dataset. To enable a head-to-head comparison, we used both AdGraph and Duumviri to crawl the top 5K sites from the Tranco List [34] simultaneously, between June 22, 2024, and July 9, 2024. We tabulate the crawl information in Table V. Each tool uses its own browser to establish a web session with the web servers. The set of requests that each tool analyzed are different due to factors such as session-specific requests (e.g., URLs containing session IDs) — we denote AdGraph's set as R_A and Duumviri's as R_D. To form our evaluation dataset, we took the intersection of R_A and R_D to get a dataset that both tools analyzed excluding requests only analyzed by individual work. To prevent a few very commonly used trackers, such as Google Analytics, from dominating our results, we performed a deduplication by request URL so that each request appears only once in our dataset. Our evaluation dataset contained a total of 18,122 requests. We first compare the accuracy of both tools against labels derived from EasyList and EasyPrivacy (referred to as filter lists thereafter). Of the 18,122 requests, the filter lists label 15,233 as non-trackers, and the remaining 2,889 as trackers. We then tabulate the accuracy of Duumviri and AdGraph at predicting the filter list labels in Table VIII. We observed that AdGraph and Duumviri achieved similar performance using filter lists as the ground truth. However, since filter lists are imperfect (e.g., Duumviri found EasyPrivacy-caused breakages in §V-B2), in the next section, we further analyze instances where Duumviri and AdGraph disagree.

Disagreement Analysis. We conducted a manual analysis of the disagreements between AdGraph and Duumviri to determine which method is more likely to make accurate tracker predictions. There are two types of disagreements: 242 re-

quests where AdGraph labeled the request as a tracker while Duumviri labeled it as a non-tracker, and 2,034 requests where AdGraph labeled the request as a non-tracker while Duumviri labeled it as a tracker. We manually sampled 40 cases from each type of disagreement, totaling 80 requests. Each request was manually labeled as either a tracker, a non-tracker or undecided using the same methodology described previously in §V-B2.

Out of the 40 samples that were labeled by AdGraph as trackers, AdGraph was correct in 27 (67.5%) of the cases and Duumviri was correct in the remaining 13 (32.5%) of the cases. Notably, we found two instances (from our sampled set, and ve such requests in the whole set) of functional non-tracker requests being mis-labeled by AdGraph as trackers. Blocking such requests caused web page breakage. Both instances were requests on engadget.com that load images as part of the content from Yahoo's image resizing and optimization service. Blocking these requests broke the page as the images were absent; such requests share similarities with tracking requests syntactically. Duumviri, on the other hand, was able to detect the breakage with its breakage detector — the breakage detector returned a higher-than-threshold probability indicating that a breakage occurred when the request was blocked. As a system of two detectors, Duumviri does not label such requests as trackers when the breakage detector detects breakage. Out of the 40 samples where Duumviri labeled requests as trackers, 30 (75%) requests are actual trackers that AdGraph failed to identify and 8 (20%) requests were non-trackers. The remaining 2 (5%) requests were undecided. We did not observe any cases where non-trackers caused breakage.

We note that Duumviri labels far more requests as tracking and is also correct more often when it labels a request as tracking. By taking the rates at which Duumviri is correct for the two disagreement types (13/40 and 30/40) and weighting them by the number of each disagreement (242 and 2034), we can expect Duumviri to be correct in roughly 71% of the cases when the tools disagree, demonstrating bene ts of Duumviri's approach over AdGraph. In comparison, a similar analysis would reveal that AdGraph is only correct in 25% of the cases where they disagree (we cannot estimate the remaining because of the 2 undecided cases). When Duumviri's accuracy compared to AdGraph is combined with the previous results from §V-B2, where Duumviri was correct in 55.1% (494/898) of the cases when Duumviri labels a tracker and lter lists label it as a non-tracker, we believe Duumviri is more likely to make the correct prediction compared to AdGraph and lter lists. Additionally, Duumviri's breakage detector was able to correctly identify two non-tracker functional requests where AdGraph misidenti ed them as trackers, causing page breakage.

4) Q4: Bene t to the Community:*Speed.* On average, to analyze a single resource, Duumviri needs 363.64 seconds. This includes performing differential analysis (225.66s, 62.06%), feature extraction (137.93s, 37.93%) and invoking the detectors (0.05s, 0.01%). The differential analysis step took the longest as this step is mostly CPU-bound: Duumviri needs to launch two browser instances and load the page. Duumviri is single-threaded, and thus, the computation time applies to our 2.4 GHz vCPU without any parallelism or GPU. We believe this setup is average, and the computation time is repeatable by others. In the evaluation, we parallelized 72 instances of Duumviri on 72 cores and analyzed 53,217 requests in slightly less than four days. This evaluation generated 23,737 rules, which translates to 5,934 (23,737 / 4 days) rules per day. In comparison, lter lists insert 29.8 rules daily [16].

Cost. Based on current spot rate of USD $0.00269/vCPU hour on Google Cloud at Las Vegas [35], it takes USD $0.02663 ($0.00269*60*60 / 363.64s) to analyze a single candidate with Duumviri. We estimate that it could cost USD $ 1785.036 (0.02663*67,031) to analyze the whole dataset if ran on Google Cloud. This translates to an hourly cost of USD $18.59. Duumviri is cost-effective compared to human developers that manually identify trackers with an average hourly worker wage of $48.32 [36]. Since Duumviri is able to reproduce lter list labels quicker and at a lower cost, we believe it bene ts the tracker detection community.

C. Mixed Trackers Detection

In mixed tracker evaluation, we aim to answer the following questions empirically:

Q1: Can Duumviri's predictors effectively identify mixed request trackers? (§V-C1)
Q2: Can Duumviri stop mixed response trackers? (§V-C2)
Q3: Can Duumviri identify mixed request trackers in the wild? (§V-C3)

We refrain from a direct comparison with previous work as we were unsuccessful in executing the previous work's code.

1) Q1: Detector Accuracy:*We report the 5-fold cross-validation accuracy numbers in the last row of Table II.*

2) Q2: Stopping Mixed Response Trackers:*To answer whether Duumviri's method of blocking mixed request trackers can also stop requests with privacy sensitive information initiated by mixed response trackers, we rst gathered mixed response trackers from our 15K page crawl described in §V-B2. Mixed response trackers are labeled by speci c rules from UbO and AdGuard: redirect, replace, jsonprune, and hls. We identi ed 8,677 instances of mixed response trackers with demographics detailed in Table XV in Appendix C. We then collect requests initiated by mixed response trackers. To properly attribute outgoing requests to mixed response trackers, we rst render the page containing mixed response trackers. Then, we rely on Chrome DevTool Protocol's requestWillBeSent event. In the event handler, we check the request's initiator stack. A request is initiated by a mixed response tracker if the mixed response tracker appears anywhere in the initiating stack of that request. In total, we collected 4,734 requests initiated by mixed response trackers. We again use lter lists to obtain a set of ground truth labels: 4,603 of 4,734 requests were labeled as trackers. This high percentage of trackers is expected because, as lter lists cannot block mixed response trackers directly, they instead block the requests that such trackers send out. Invoking Duumviri on these requests, we achieved an accuracy of 95.39%, indicating that*

Duumviri can effectively stop mixed response trackers from sending privacy-sensitive information.

3) Q3: Detecting Mixed Trackers in the Wild.We evaluate Duumviri's ability to automatically identify mixed request trackers and assess their impact.

Evaluation Dataset. To build our evaluation dataset, we starting from the complete 15K page crawl dataset used the non-mixed tracker evaluation, and performs the following procedure:

1) Filter out trackers by lter lists. We lter out all requests that lter lists label as trackers.
2) Filter out tracker-dependent requests. We lter out all requests that are not observable when lter lists are enabled: they are either probabilistic or dependent on lter lists-labeled requests.
3) Expand into (request, request eld) pairs. To prepare for partial-request granularity analysis, for each request in the dataset, we expand it into (request, request eld) pairs by parsing the URL parameters and cookie elds. Each parameter and each cookie will be a row in the dataset.
4) Filter out non-identi er request elds. At this step, we have a dataset of 251,038 (request, request eld) pairs. Based on the average execution speed in §V-B4, it will take 2.89 years to exhaustively analyze all of them. However, we realize that not all request elds contain privacy-sensitive information: previous work has focused on identi er-like strings in network requests as they contain potentially privacy-sensitive information [37], [1], [37]. To compose a feasible dataset for evaluation, we adopt the same heuristic by ltering out all request elds that are non-identi er like. We leave the detail of this ltering step in Appendix.

Invoking Duumviri on these elds, we obtained 7,133 positive labels and 11,302 negative labels. Lacking existing ground-truth labels for mixed trackers, and considering that advanced content blockers may block a broader range of information, we manually analyzed a subset by randomly sampling 40 cases from each prediction label, totaling 80 cases.

Manual Analysis. Our goal is to assign each request eld to one of the following labels: 1) Breakage: if the eld removal causes page breakage. 2) Stale eld: if the eld serves functionality but removing it has no impact on the page. 3) Tracking eld: if the eld serves tracking, and eld removal does not break the page. 4) Undecided: if we cannot decide on the purpose of the request or the eld.

Methodology. We detail the procedure for assigning labels manually. While we follow the same method to assign request purposes as in §V-B2, we detail how we assign eld purposes and impact. Determining eld purposes is challenging as they are server-designed and used; it is possible that a server names a eld in a common way and uses it for a different purpose. We determine eld purposes based on the following: 1) Documentation. We rst try to search for any documentation on the eld if possible. For query string parameters, we label a eld as tracking if the documentation states that it is related to advertising, analytics or user identi cation. For cookies, we look up the cookie's purpose on Cookiepedia. 2) Field name and value. If the eld has a common name, we look at how other parties use the same eld if the eld value format also matches (e.g., date, hash value). While this process is relatively easy for some query string parameters such as `hash' for hash value and `v' for version number, it is dif cult to draw any conclusion for other query string parameters like `token' or `id'. We apply a conservative approach and assign a label of undecided if we have no concrete evidence to assign other labels. To assign eld impact, we block the eld using the `removeparam' and `cookie' rule and use the same method described in §V-B2 to evaluate if the web page is broken.

TABLE IX
MANUAL ANALYSIS RESULTS OF THE POSITIVE AND NEGATIVE CASES

	Positive Cases			Negative Cases		
	#	%	Estimated	#	%	Estimated
Tracking elds	26	65	4,636	2	5	565
Immediate breakage	4	10	713	18	45	5,086
Potential breakage	1	2.5	178	0	0	0
Stale elds	6	15	1,070	14	35	3,956
Undecided	3	7.5	535	6	15	1,695

Positive Case Analysis.We present a summary of the analysis results in the rst four columns of Table IX. Out of the positive cases that we analyzed, 65% elds are potentially for tracking purposes. For instance, the `sessionId' parameter in Twitter's pro le fetching and Tweet fetching API is potentially tracking. While we can never con rm whether it is used for tracking, we, through our manual analysis, found that this parameter in Twitter-set, has high entropy and removing it does not break Twitter's API on the page. We show one example on `myblogguest.com' in Fig 4 in Appendix C. None of the other parameters in the API are considered for tracking purposes. We detail additional ndings at the end of this section. We found 10% cases are page breakage. For instance, kyoto-ryokan.co.jp sends a request to fetch TripAdvisor's certi cate of excellence logo with a parameter `wtype=certi cateOfExcellence'. While blocking this parameter changes the logo's appearance slightly, we deemed this as not tracking due to the nature of the logo. We also found 2.5% to be parameters that have no immediate impact on the page but de ne an action that is executed conditionally at a later time dependent on user interactions. For instance, YouTube automatically tries to sign in to the user's Google account, and the request `https://accounts.google.com/ServiceLogin' contains a continuation parameter that details the subsequent action after successful sign-in (continue=`https://youtube.com/signin'). Blocking this parameter has no immediate impact on the page but may break functionality if triggered by user interaction. Thus, for this type of interaction-dependent breakage, we introduce a new label: potential breakage. We also found 15% of the elds to be stale parameters. Blocking these elds has no impact on the server response. We were unable to decide on the remaining 7.5% of the elds.

Negative Case Analysis.We present a summary of the analysis results in the last three columns of Tab IX. We found 5% cases are tracking elds that Duumviri mistakenly predicted as

functional. 45% cases would cause breakage if blocked, 35% are stale fields and 15% cases are undecided.

Using the estimated numbers, we can calculate a lower bound for Duumviri's accuracy in detecting mixed request fields to be 74.19%((4,636+5,086+3,956)/18,435). We believe that our accuracy can be improved with a better training dataset as our current training dataset contains noise as discussed in §II.

Duumviri Findings. While we believe there are more trackers to be discovered in our dataset, we measured the occurrence of confirmed trackers since our dataset consists of unique trackers. We observed a total of 83 occurrences in our evaluation dataset. We detail additional findings below. One case involves yandex.com, which uses yandex.com/portal/set/any to store user session data such as configuration data, considered legitimate site functionality. However, Duumviri determined one query string parameter, szm=1:800x600:780x476, to be tracking, as strings containing resolutions are commonly used for device fingerprinting. Removing this parameter had no impact on user experience across sessions in our manual analysis. Another case involves aaalife.com, which loads siteintercept.qualtrics.com::FeedbackButtonModule.js?Q_CLIENTVERSION=1106.0&_CLIENTTYPE=web&Q_BRANDID=aaalife, a script rendering and supporting the feedback button, deemed legitimate page functionality. Upon closer inspection of its query string parameters, Duumviri identified Q_BRANDID as a tracking parameter because it contains the source domain of the page. While such analytical information also appears in the Referer header, Duumviri helps users who blocks this header ensuring such information is not leaked.

VI. Related Work

A. Tracker Identification

We discuss proposals by the proposed feature source.

Based on Network-level Information. Several works hypothesize that trackers exhibit distinctive characteristics at the network level and thus extract features from network requests and responses. Bhagavatula et al.[38] focus on request query parameters, while Gugelmann et al.[6] construct features from both requests (e.g., partiness) and responses (e.g., response size). Additionally, detecting cookie syncing has employed request and response features [39].

A core insight used in many works, including Duumviri for its mixed tracker evaluation, is that tracking relies on communicating identifier-like strings to servers. Yu et al.[1] emphasize the crucial role of such identifiers in tracking. The flow of unique values enables a server to distinguish clients[37], [1] or engage in cookie syncing [40]. Privacy Badger [37] employs the heuristic that a third-party domain is classified as a tracking domain if it sets unique identifier-like strings on more than three other domains.

Based on JavaScript Features. Another set of works hypothesizes that trackers exhibit different behavior (e.g., API invocations, DOM accesses) compared to functional JavaScript, extracting features based on static or dynamic JavaScript code features. Ikram et al.[7] leverage syntactic and semantic JavaScript code features to detect trackers. Static analysis is vulnerable to obfuscation[41], making this approach less effective. On the other hand, Kaizer et al.[42] and DOMTrackerDetector[43] rely on dynamic features, such as accessed properties, to build classifiers.

Based on Rendering Graph. Recently, proposals have emerged that instrument the browser (e.g., Chromium [19] and Firefox [9]) to allow complete attribution of document modification, network resources, and JavaScript execution. These proposals model a web page's rendering process as a graph, from which works (Duumviri included) extract features to predict trackers. All existing works train a single model for tracker detection. Duumviri incorporates additional breakage detector, increasing overall accuracy while minimizing breakages.

1) **Mixed Trackers and Defenses.** Amjad et al. [44], [45] conducted the first analysis of the prevalence of mixed trackers, revealing that 15% of scripts are mixed trackers that cause breakage when blocked at the file granularity. One imprecision in Amjad's work is assuming that all requests not flagged by filter lists are functional requests. This is an overestimation as filter lists may not be complete, potentially inflating the number of identified mixed trackers.

SugarCoat [11] is the first work addressing mixed trackers by attempting to automate the fixing of pre-identified mixed trackers. SugarCoat sanitizes mixed trackers by generating resource replacements where sensitive APIs are patched. While SugarCoat does not address the identification problem, it assumes that all privacy-sensitive APIs in a mixed tracker are used for tracking purposes. This assumption can lead to broken pages if the functional component of a mixed tracker also involves privacy-sensitive APIs, as patching such APIs can break the functional component. SugarCoat users must manually inspect to ensure no breakage.

ASTrack [10] operates on the observation that much third-party tracking code is embedded into first-party contexts, sharing code structures across first parties. By leveraging developer-identified tracking code patterns, ASTrack automatically identifies other tracking code finding similar patterns in presumably mixed scripts. However, it is unclear how ASTrack will work on newly constructed trackers, meaning it can have low recall and potentially miss new trackers. Duumviri does not directly identify mixed response trackers but can identify privacy-sensitive information sent by mixed response trackers (§V-C2), stopping their tracking activities.

B. Web Page Breakage Evaluation

One reason for web page breakage is the use of content blockers who reply on developer-generated filter lists to remain effective [16]. Breakage can occur due to mistakes in any step (identification, fixing, and breakage evaluation) of the developer workflow. Additionally, web page admins may intentionally mix trackers with functional resources to evade detection [44], and blocking mixed trackers can lead to breakages. Very few works have systematically studied the symptoms of breakage. Mathur et al. [46] conducted a survey on

content blocker usage, evaluating why users adopt or avoid content blockers and how frequently they encounter blocker-induced breakage along with its symptoms. While users reported that content blockers rarely break sites, users might lack the technical background to identify certain types of breakage. Nisenoff et al. [13] systematically studied breakages caused by content blockers, analyzing user reports of broken pages and providing statistics on broken symptoms (e.g., missing content, non-responsiveness). We designed our breakage detector by encompassing their comprehensive set of symptoms.

Automatic Breakage Evaluation. Breakage evaluation in current tracker blocker implementations is predominantly manual. Content blockers like Adblock Plus and uBlock Origin depend on user reports of page breakages [47]. Research proposals like AdGraph [8] rely on manual evaluation, which is subjective and does not scale with automatic tracker detection.

Research proposals attempt automatic breakage evaluation. Yu et al. [1] propose using the page reload rate to detect web page breakage, assuming users will reload a web page if they experience a broken site. This assumption relies on the user to notice and react to breakages, which can be demanding. Moreover, a page reload may occur for reasons not related to breakage (e.g., to keep a live session). PriVaricator [21] and ASTrack [10] use changes in web page appearance to determine breakage. A page is potentially broken if its appearance differs from a non-broken page. This approach can be inaccurate as 1) there exist breakages without visual change (e.g., non-responsiveness), 2) this method suffers from visual non-determinism (§III-A3) where page appearance changes without breakage. AutoFR [48] uses changes in the number of images and text before and after a page change to determine breakage. However, its heuristic is oversimpli ed and incomplete as it does not account for other ways a user can perceive page breakages (e.g., non-responsiveness). Blocked or Broken [12] attempts to automatically predict broken pages based on features extracted from PageGraphs [19]. While its goal is similar to Duumviri's breakage detector (one of Duumviri's components), the approaches differ in data collection and the used features . Blocked or Broken collects broken page samples through commit messages, which are less relevant (as the commit may be outdated) and less standardized (i.e., not all commit messages follow conventions) compared to Duumviri's exception rules. SINBAD [49] is another work for breakage prediction based on user report forums data. Similar to commit messages, forums data is also a record of breakage occurred in the past and can be outdated.

Duumviri designed its features by modeling externally-visible channels covering all breakage symptoms discussed in [13]; it collects its training data from exception rules from current lter lists which are constantly checked by privacy develoeprs. Consequently, Duumviri's breakage detector achieves higher test accuracy (shown in Table II) than previous works.

VII. Limitations

First, the breakage detector currently only measures breakage that occurs immediately on the client side. This means that it is unable to detect 1) breakages that do not occur immediately (e.g., after user interactions) or 2) breakages that only involve server-side symptoms. To be able to model non-immediate breakages, Duumviri will need precise instructions as to how the breakages can be triggered (e.g., interactions to perform). While some issue reports contain such instructions, and tools such as large language models can help extract such instructions, we leave it as future work. To be able to model breakages with server-side symptoms, server cooperation is needed, which is challenging as the server is the party that tracks. Second, Duumviri only support stateless analysis (e.g., no support for authenticated user sessions) because the modified requests it sends out, during mixed tracker analysis, may affect server states leading to server-side breakages. Third, Duumviri currently does not detect dependencies among request-or-request elds. It is possible that blocking multiple requests/request elds at the same time lead to an opportunity of tracker blocking, however, it is expensive (exponential to the number of request/request elds) to enumerate all such possible combinations and we leave it as a future work. Last, a server may detect and react to the modi ed requests that Duumviri sends out during mixed tracker detection. The reaction may include, and is not limited to, sending fake responses or refusing to provide service, preventing Duumviri from working.

VIII. Conclusion

In this paper, we introduced Duumviri, a framework designed to identify both non-mixed and mixed trackers. Duumviri has two novel mechanisms: 1) the addition of a breakage detector into tracker detection, which enables Duumviri to detect web page breakage due to misidenti cation of non-tracker as tracker and blocking mixed trackers. 2) the use of differential features which enables Duumviri to identify tracking components contained in mixed trackers.

Our results demonstrate Duumviri's accuracy of 97.44% in detecting non-mixed trackers when compared to labels provided by lter lists across 15K pages. Moreover, we uncovered 22 unreported trackers and identi ed 2 cases of page breakage due to lter lists. In mixed tracker evaluation, Duumviri achieves an accuracy of 74.19%. Our manual analysis led to the discovery of 26 mixed request trackers. By promptly reporting our newly identi ed trackers and breakages to developers, we anticipate that Duumviri will prove valuable to the community.

Acknowledgments

We thank the anonymous reviewers for their feedback, and the artifact evaluation committee members for feedback on our artifacts. We also thank Bright Data for providing us with access to their proxies, without which this work would not be possible. Funding for this work was provided in part by ONR Contract N00014-17-1-2889, a contract with Telus, and NSERC Alliance Grant ALLRP 586310-23. He Shuang was supported by Ontario Graduate (OGS) and Bell Canada Scholarships. David Lie is supported by a Tier 1 Canada Research Chair.

References

[1] Z. Yu, S. Macbeth, K. Modi, and J. M. Pujol, "Tracking the trackers," in Proceedings of the 25th International Conference on World Wide Web, ser. WWW '16. Republic and Canton of Geneva, CHE: International World Wide Web Conferences Steering Committee, 2016, p. 121–132. [Online]. Available: https://doi.org/10.1145/2872427.2883028

[2] B. Liu, A. Sheth, U. Weinsberg, J. Chandrashekar, and R. Govindan, "Adreveal: Improving transparency into online targeted advertising," in Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, 2013, pp. 1–7.

[3] C. Castelluccia, L. Olejnik, and T. Minh-Dung, "Selling off privacy at auction," in Network and Distributed System Security Symposium (NDSS), 2014.

[4] Q. Chen, P. Snyder, B. Livshits, and A. Kapravelos, "Detecting filter list evasion with event-loop-turn granularity javascript signatures," 2021 IEEE Symposium on Security and Privacy (SP). IEEE, 2021, pp. 1715–1729.

[5] J. Rack and C.-A. Staicu, "Jack-in-the-box: An empirical study of javascript bundling on the web and its security implications," in Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, 2023, pp. 3198–3212.

[6] D. Gugelmann, M. Happe, B. Ager, and V. Lenders, "An automated approach for complementing ad blockers' blacklists," Proc. Priv. Enhancing Technol., vol. 2015, no. 2, pp. 282–298, 2015.

[7] M. Ikram, H. J. Asghar, M. A. Kaafar, A. Mahanti, and B. Krishnamurthy, "Towards seamless tracking-free web: Improved detection of trackers via one-class learning," Proceedings on Privacy Enhancing Technologies, vol. 2017, no. 1, pp. 79–99, 2017.

[8] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, "Adgraph: A graph-based approach to ad and tracker blocking," 2020 IEEE Symposium on Security and Privacy (SP). IEEE, 2020, pp. 763–776.

[9] S. Siby, U. Iqbal, S. Englehardt, Z. Shafiq, and C. Troncoso, "Webgraph: Capturing advertising and tracking information flows for robust blocking," arXiv preprint arXiv:2107.11309, 2021.

[10] I. Castell-Uroz, K. Fukuda, and P. Barlet-Ros, "Astrack: Automatic detection and removal of web tracking code with minimal functionality loss," arXiv preprint arXiv:2301.10895, 2023.

[11] M. Smith, P. Snyder, B. Livshits, and D. Stefan, "Sugarcoat: Programmatically generating privacy-preserving, web-compatible resource replacements for content blocking," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021, pp. 2844–2857.

[12] M. Smith, P. Snyder, M. Haller, B. Livshits, D. Stefan, and H. Haddadi, "Blocked or broken? Automatically detecting when privacy interventions break websites," Privacy Enhancing Technologies Symposium (PETS), p. 6–23, 2022.

[13] A. Nisenoff, A. Borem, M. Pickering, G. Nakanishi, M. Thumpasery, and B. Ur, "De ning "broken": User experiences and remediation tactics when ad-blocking or tracking-protection tools break a website's user experience," in Proceedings of the 32nd USENIX Security Symposium, 2023.

[14] P. Snyder, S. Karami, A. Edelstein, B. Livshits, and H. Haddadi, "PoolParty: Exploiting browser resource pools for web tracking," 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 7091–7105.

[15] ryanbr, "easylist/easylist commit d56ebb6," https://github.com/easylist/easylist/commit/d56ebb6be562f61047efebe7bdf2a2f9dfaf4778, (Accessed on 08/01/2024).

[16] P. Snyder, A. Vastel, and B. Livshits, "Who lters the lters: Understanding the growth, usefulness and ef ciency of crowdsourced ad blocking," in Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems, ser. SIGMETRICS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 75–76. [Online]. Available: https://doi.org/10.1145/3393691.3394228

[17] A. Barbaresi, "Tra latura: A Web Scraping Library and Command-Line Tool for Text Discovery and Extraction," in Proceedings of the Joint Conference of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing: System Demonstrations. Association for Computational Linguistics, 2021, pp. 122–131. [Online]. Available: https://aclanthology.org/2021.acl-demo.15

[18] G. Storey, D. Reisman, J. Mayer, and A. Narayanan, "Perceptual ad highlighter," 2017, chrome Extension: https://chrome.google.com/webstore/detail/perceptual-ad-highlighter/mahgileahghaapkboihnbhdplhnchp.

[19] Brave, "Pagegraph · brave/brave-browser wiki · github," https://github.com/brave/brave-browser/wiki/PageGraph, (Accessed on 08/01/2024).

[20] A. Market, "How google analytics collects data - analytics-market," https://www.analytics-market.com/blog/how-google-analytics-collects-data/, (Accessed on 08/01/2024).

[21] T. Gowda and C. A. Mattmann, "Clustering web pages based on structure and style similarity (application paper)," 2016 IEEE 17th International conference on information reuse and integration (IRI). IEEE, 2016, pp. 175–180.

[22] N. Nikiforakis, W. Joosen, and B. Livshits, "Privaricator: Deceiving ngerprinters with little white lies," in Proceedings of the 24th International Conference on World Wide Web, 2015, pp. 820–830.

[23] M. Tan and Q. Le, "Ef cientnet: Rethinking model scaling for convolutional neural networks," in International conference on machine learning. PMLR, 2019, pp. 6105–6114.

[24] A. Goel, J. Zhu, R. Netravali, and H. V. Madhyastha, "Jawa: Web archival in the era of JavaScript," in 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), 2022, pp. 805–820.

[25] D. D. M. L. Community, "Xgboost documentation — xgboost 2.0.3 documentation," https://xgboost.readthedocs.io/en/stable/, (Accessed on 08/01/2024).

[26] gwarser, "Implement blocking by post body parameters · issue #1357 · ublockorigin/ublock-issues," https://github.com/uBlockOrigin/uBlock-issues/issues/1357, (Accessed on 08/01/2024).

[27] Cookiepedia, "All you need to know about cookies," https://cookiepedia.co.uk/, (Accessed on 08/01/2024).

[28] I. Archive, "Wayback machine," https://web.archive.org/, (Accessed on 08/01/2024).

[29] Twitter, "Tracking tags — docs — twitter developer platform," https://developer.twitter.com/en/docs/twitter-ads-api/campaign-management/api-reference/tracking-tags, (Accessed on 08/01/2024).

[30] Adobe, "Adobe analytics 2.0 api reference," https://developer.adobe.com/analytics-apis/docs/2.0/apis/, (Accessed on 08/01/2024).

[31] jordanelliott23, "youtube tracker · issue #7878 · easylist/easylist," https://github.com/easylist/easylist/issues/7878, (Accessed on 08/01/2024).

[32] B. Slayter, "Tracker radar wiki," https://slaytergw.github.io/tracker-radar-wiki/, (Accessed on 08/01/2024).

[33] Z. Yang, W. Pei, M. Chen, and C. Yue, "Wtagraph: Web tracking and advertising detection using graph neural networks," IEEE Symposium on Security and Privacy, 2022.

[34] V. L. Pochat, T. Van Goethem, S. Tajalizadehkhoob, M. Korczyński, and W. Joosen, "Tranco: A research-oriented top sites ranking hardened against manipulation," arXiv preprint arXiv:1806.01156, 2018.

[35] Google, "Pricing — compute engine: Virtual machines (vms) — google cloud," https://cloud.google.com/compute/all-pricing, (Accessed on 08/01/2024).

[36] U. B. of Labor Statistics, "Table b-3. average hourly and weekly earnings of all employees on private nonfarm payrolls by industry sector, seasonally adjusted - 2023 m10 results," https://www.bls.gov/news.release/empsit.t19.htm, (Accessed on 08/01/2024).

[37] E. F. Foundation, "Privacy badger," https://privacybadger.org/, (Accessed on 08/01/2024).

[38] S. Bhagavatula, C. Dunn, C. Kanich, M. Gupta, and B. Ziebart, "Leveraging machine learning to improve unwanted resource filtering," in Proceedings of the 2014 Workshop on Arti cial Intelligent and Security Workshop, 2014, pp. 95–102.

[39] P. Papadopoulos, N. Kourtellis, and E. Markatos, "Cookie synchronization: Everything you always wanted to know but were afraid to ask," in The World Wide Web Conference, 2019, pp. 1432–1442.

[40] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in Proceedings of the 2016 ACM SIGSAC conference on computer and communications security, 2016, pp. 1388–1401.

[41] H. Le, F. Fallace, and P. Barlet-Ros, "Towards accurate detection of obfuscated web tracking," in 2017 IEEE International Workshop on Measurement and Networking (M&N). IEEE, 2017, pp. 1–6.

[42] A. J. Kaizer and M. Gupta, "Towards automatic identi cation of javascript-oriented machine-based tracking," in Proceedings of the 2016

ACM on International Workshop on Security And Privacy Analytics 2016, pp. 33–40.
[43] Q. Wu, Q. Liu, Y. Zhang, P. Liu, and G. Wen, "A machine learning approach for detecting third-party trackers on the web," in European Symposium on Research in Computer Security. Springer, 2016, pp. 238–258.
[44] A. H. Amjad, D. Saleem, M. A. Gulzar, Z. Shafiq, and F. Zaffar, "Trackersift: Untangling mixed tracking and functional web resources," in Proceedings of the 21st ACM Internet Measurement Conference, 2021, pp. 569–576.
[45] A. H. Amjad, Z. Shafiq, and M. A. Gulzar, "Blocking javascript without breaking the web: An empirical investigation," arXiv preprint arXiv:2302.01182, 2023.
[46] A. Mathur, J. Vitak, A. Narayanan, and M. Chetty, "Characterizing the use of Browser-Based blocking extensions to prevent online tracking," in Fourteenth symposium on usable privacy and security (SOUPS 2018), 2018, pp. 103–116.
[47] A. Plus, "Adblock plus - open adblock plus forums," https://forum.adblockplus.org/viewforum.php?f=6&sid=687bc0da62aa1dcc59b1393439808def, (Accessed on 08/01/2024).
[48] H. Le, S. Elmalaki, A. Markopoulou, and Z. Shafiq, "Autofr: Automated filter rule generation for adblocking," arXiv preprint arXiv:2202.12872, 2022.
[49] S. E. H. Chehade, S. Siby, and C. Troncoso, "Sinbad: Saliency-informed detection of breakage caused by ad blocking," 2024 IEEE Symposium on Security and Privacy (SP). IEEE Computer Society, 2024, pp. 211–211.
[50] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma, "Vips: a vision-based page segmentation algorithm," 2003.
[51] M. Cormer, R. Mann, K. Moffatt, and R. Cohen, "Towards an improved vision-based web page segmentation algorithm," 2017 14th Conference on Computer and Robot Vision (CRV). IEEE, 2017, pp. 345–352.
[52] r2fo, "Block brightdata · issue #1580 · adguardteam/adguardsdnsfilter," https://github.com/AdguardTeam/AdGuardSDNSFilter/issues/1580, (Accessed on 08/01/2024).
[53] Zenodo, "Zenodo," https://zenodo.org/, (Accessed on 08/30/2024).

Appendix A
Identifier-like Field Filtering

Duumviri uses entropy to select request fields for evaluation. The main insight is similar to previous works: a request field is likely to be used for tracking if it contains high entropy. Tracking identifiers tend to have high entropy as they need to uniquely identify a user. One problem with using field entropy to find identifier is that an identifier may be assembled from multiple low entropy request fields. For instance, a tracking identifier may be transmitted as a single value in a single request field (e.g., uid=aX13nL) or it can be transmitted as query string parameters in several requests and only assembled on the server side.

Duumviri employs two methods for entropy calculation: 1) per field: this method calculates the total entropy of a single request field by estimating the total number of combinations that the field can hold based on the character set size and the field length. 2) per server: this method calculates the total entropy of query string parameters and cookies of all requests sent to a single server to determine if the server can potentially assemble a high-entropy identifier.

For mixed tracker evaluation, Duumviri uses two configurable thresholds, 1 billion for per field and 1 trillion for per server, for the entropy calculation and keeps all fields that identified by either methods. The threshold setting poses a trade-off between performance and precision. A low threshold produces a larger set of request fields for evaluation with more irrelevant ones, and thus Duumviri runs slower, while a high threshold produces a smaller set of more relevant request fields for faster evaluation but it is at the risk of missing mixed trackers. We picked our thresholds based on the identifiers observed in current non-mixed trackers.

TABLE X
Compare Duumviri's entropy filtering methods to the trackers identified by EasyList and EasyPrivacy. Agreements refer to the number of requests where Duumviri and filter lists agree.

Method	Agreements	Duumviri Only	Filter List Only
Per field	2812 (60.36)	1526 (32.76)	320 (6.87)
Per server	2194 (47.1)	2370 (50.88)	94 (2.02)

To show the effectiveness of our heuristics and the thresholds, we compare the number of requests with identifiers that Duumviri flags on the top 100 sites of Alexa Top 1M List generated in May 2022, to those identified by human-constructed filter lists such as EasyList and EasyPrivacy. If our methods can identify a close super-set of all trackers, then it is effective in finding tracking identifiers yet saving analysis on fields that are unlikely to be trackers. We show the numbers in Table X. We see that both methods produce requests sets that contain the majority of filter lists identified trackers with minimum misses (6.87% and 2.02%) respectively.

Appendix B
Breakage detector Features

We list all differential features used in Duumviri's breakage detector. We show all appearance features in Table XI, input handling features in Table XII, request features in Table XIII and the remaining features in Table XIV.

Appendix C
Examples

We show the following examples.

1) An example of filter lists-caused page breakage is shown in Fig 3. This example happens on `ero-advertising.com', where the script `www.eroadvertising.com/js/controllers.js' is erroneously blocked by EasyPrivacy on this domain causing the absence of main body content as shown on the right side of the figure. The functional page is shown on the left side. Although `ero-advertising.com' is an ads-serving domain, it is still a breakage as other users may want to visit the page (e.g., a customer of the company). Easylist and EasyPrivacy have historically fixed similar breakage [52].
2) An example of Duumviri discovered mixed tracker is shown in Fig 4. We found the `sessionId' parameter in the request `https://platform.twitter.com/embed/Tweet.html' to be tracking on `myblogguest.com'. Removing this request breaks the page as the Tweet fails to display properly as shown on the right side of the figure. The functional page with the Tweet display is shown on the left.
3) We also shown an example of trackers identified through internal documentation in Fig 5.
4) We show the demographics of mixed response tracker from mixed response tracker evaluation (§V-C2) in Table XV.

TABLE XI
APPEARANCE-RELATED FEATURES

#	Feature Name	Description
1	VIPS screenshot	Cosine similarity among the largest section of page screenshots as returned by VIPS [50] and Cormer et
2	Cormer screenshot	
3	Main screenshot	Cosine similarity between the screenshot of the first main/section tag
4	Section screenshot	
5	Feature vectors	Cosine similarity of the feature vectors obtained by passing the screenshots to EfficientNet [23]
6	Text	Cosine similarity of the bag of words of the document text
7	Readability text	Cosine similarity of the document text extracted from Trafilatura [17]
8	Document style	Cosine similarity of the CSS classes
9	Structure similarity	Cosine similarity of the sequence of the HTML tags
10	HTML	A joint of style and structure similarity
11	Fonts	between the loaded fonts
12	Color	of the unique colors used
13	Height	document height
14	Canvas	of canvas/audio/button/input/link(*a* tag)/script/span element
15	Audio	
16	Button	
17	Input	
18	Links	
19	Dom scripts	
20	Span	
21	Unloaded diff	of the number of *Window.before_unload* event
22	CSS files	of the number of CSS files parsed
23	Videos small	The difference in the number of video/image/iframe tag elements of small/large and sensitive size. Small size means the total area of the element is less than 10px. Sensitive size means the width and height of the element matches are commonly used for ads.
24	Videos large	
25	Video sensitive size	
26	Images small	
27	Images large	
28	Images sensitive size	
29	Iframes small	
30	Iframes large	
31	Iframes sensitive size	
32	Ads iframes	The number of iframes that has no inner text
33	Ad highlighter	of the number of ads as identified by a perceptual ad detector [18]

TABLE XII
INPUT HANDLING-RELATED FEATURES.

#	Feature Name	Description
34	Specific listeners	of the event listeners on specific/generic/sensitive/critical elements.
35	Generic listeners	
36	Sensitive listeners	
37	Critical listeners	
38	Functionality related listeners	of the event listener types that are commonly used to serve functionality.
39	Listeners	of all event listeners

TABLE XIII
REQUEST FEATURES. FP REFERS TO FIRST PARTY, TP REFERS TO THIRD PARTY.

#	Feature Name	Description
40	# requests blocked	The # and the % of blocked requests
41	% requests blocked	
42	URL length	The length of the blocked requests
43	Total parameters	Total number of parameters in blocked requests
44	Ad dimensions	# of requests that contain dimension-like string in its URL
45	# semicolon	Total number of semicolons in the blocked requests
46	# screen	# of the word 'screen' is in the blocked requests
47	# FP in blocked requests	# of times that the first party domain exists in the blocked requests
48	# FP req blocked	# requests that are first/third party
49	# TP req blocked	
50	# ad keywords	# of ad keywords in the blocked requests
51	# storage values out	Number of storage values (local and session storage and cookies) in the blocked requests
52	API static	The number of sensitive API calls that are usually used for fingerprinting in the blocked request responses
53	Eavl keyword	# of times that the keyword 'eval' occurred in the blocked request responses
54	Total response size	Total/average sized of the blocked request responses
55	Avg response size	
56	Sensitive FP	Number of first/third party sensitive requests.
57	Sensitive TP	

TABLE XIV
STORAGE, TEMPORAL PERFORMANCE AND DEVICE INTERFACE FEATURES.

#	Feature Name	Description
58	Storage	
59	Session storage	of local storage/session storage and cookies
60	Cookies	
61	Load time	of the page loading time in seconds
62	Logs	in console log
63	Downloads	in the *downloadWillBegin* event

TABLE XV
TOP 5 MOST FREQUENT HOSTNAMES OF MIXED RESPONSE TRACKERS

Mixed Response Tracker Hostname	Count
www[.]googletagmanager[.]com	7556
securepubads[.]g[.]doubleclick[.]net	663
www[.]googletagservices[.]com	346
acdn[.]adnxs[.]com	12
connect[.]mail[.]ru	11

APPENDIX D
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: The artifacts are publicly available on GitHub [4]. The `main` branch contains the latest version of the code. We utilize Docker to provide access to our working environment. The Docker image is available on Docker Hub and is also uploaded to Zenodo [53] [5] as a tar file. Once downloaded, the tar file can be loaded into Docker with the following command: `docker load < ndss_ae_docker.tar.gz`

2) Hardware requirements: Our artifacts can be run on a commodity desktop machine with a x86-64 CPU. To ensure that all artifacts run correctly, a machine with at least 8 cores and 16 GB of RAM is recommended.

3) Software requirements: A recent Linux operating system and Docker are required. Our artifacts have been tested on Ubuntu 20.04 LTS (Focal Fossa) with Docker version 20.10.21.

4) Benchmarks: None.

B. Artifact Installation & Configuration

Our docker image is available on Docker Hub [6]. You can download our image with the following command: `docker pull 8759s/ndss_ae_docker:latest`

You can run it with the following command: `docker run -it --device /dev/fuse --privileged 8759s/ndss_ae_docker /bin/bash`

[4] https://github[.]com/dlgroupuoft/Duumviri-NDSS25

[5] https://zenodo[.]org/records/13621822

[6] https://hub.docker.com/r/8759s/ndss_ae_docker