

t.ex-Graph: Automated Web Tracker Detection Using Centrality Metrics and Data Flow Characteristics

Philip Raschke^a and Patrick Herbke Henry Schwerdtner
Service-centric Networking, Technische Universität Berlin, Germany

Keywords: Data Privacy, Web Tracking, Automated Web Tracker Detection, Graph Analysis, Machine Learning.

Abstract: The practice of Web tracking raised concerns of privacy activists and data protection authorities over the past two decades. Simultaneously, researchers propose multiple solutions based on machine learning to automatically detect Web trackers. These solutions, while proving to be promising, often remained proofs-of-concept. This paper proposes t.ex-Graph, a representation that models data flows between websites to detect Web trackers in a graph. We use a publicly available dataset containing HTTP/S requests from a crawl of the Tranco top 10K websites to extract our graph. In the second step, we feed our graph into multiple machine-learning models to predict nodes that carry out tracking activities. Our results show high accuracy of 88% and even detect yet unknown Web trackers. We publish our artifacts for fellow researchers to replicate, reproduce, and advance our results.

1 INTRODUCTION

To some extent, the Web's commercial success relies on providing content and services free of charge. These free offers are paid by users by disclosing their activities, habits, and preferences, often without their knowledge or awareness. Advertising networks and analytics providers carry out web tracking to offer content and service providers an alternative source of revenue and insights to optimize their websites. For this, these websites embed resources of those advertising networks or analytics providers, which implies that every website visitor also sends data to these parties. While well-established on the Web, this practice raises concerns of privacy activists and data protection authorities.

A vast body of research addresses the various aspects of Web tracking and its impact on an individual's privacy. Multiple algorithms based on machine learning models to detect and block Web trackers have been proposed over the past two decades. However, these proof-of-concept implementations were often not applicable in the field, besides not being made publicly available for fellow researchers or users to replicate, reproduce, or use the results.


This paper proposes *t.ex-Graph* a network representation of data flows between websites (or hosts),

which addresses the centrality of Web trackers. We extract our network from a publicly available labeled dataset containing HTTP/S requests and responses recorded while crawling the Tranco top 10K websites. We feed our data into multiple machine-learning models to predict whether a host is a tracker. Our classifier achieves 88% accuracy and identifies yet unknown Web trackers. All our components are publicly available to replicate, reproduce, and advance our results.¹

The remainder of this paper is structured as follows: Section 2 presents related work in the field of automated Web tracker detection, Section 3 discusses the concept of t.ex-Graph, Section 4 explains the data collection and generation processes and performs data analysis to motivate the design of our classifier, Section 5 presents and discusses the results. Finally, we conclude our work in Section 6.

2 RELATED WORK

This section presents related work in the field of Web tracking detection with machine learning. As stated before, the body of literature is massive, and a com-

^a <https://orcid.org/0000-0002-6738-7137>

¹Available on GitHub: see <https://github.com/t-ex-tools/t-ex-graph-converter> and <https://github.com/t-ex-tools/t-ex-graph-classifier>.

plete review is beyond the scope of this paper. We present machine-learning-based Web tracker detection approaches for (i) stateful and (ii) stateless Web tracking. For stateless Web tracking, the state of the client's machine is altered, for example, with cookies. Stateless Web tracking techniques like browser or device fingerprinting compute identifiers based on data with high entropy that the client sends along with every request.

Machine-learning-based approaches to detect unwanted resource loading were already proposed in 1999. Kushmerick presents a supervised classifier to detect online advertisements (Kushmerick, 1999). In 2005, Esfandiari and Nock presented a similar approach to automatically detect online advertisements as well (Esfandiari and Nock, 2005). The detection of advertisements and Web trackers are closely related. It is observable that the focus of researchers shifted from online advertisements to Web trackers in the last decade, while data collection and labeling approaches remained the same. We classify machine-learning-based research into two categories: classifiers for (i) stateful Web tracking for which network traffic is collected and (ii) stateless Web tracking for which JavaScript API access events and JavaScript code are collected.

2.1 Stateful Web Tracking: Network Traffic, HTTP/S, URLs

In 2010, Yamada et al. (Yamada et al., 2010) presented a classifier fed with network traffic data from a corporate network and data generated by crawling the Alexa.com top 100 websites. They further extract the estimated "visiting time" of a user at a specific website, assuming that users spend less time at websites of Web trackers. Yamada et al. label their feature vectors as Web trackers by using a composition of four publicly available blocklists. Only one of these four blocklists (a host file for Windows) is available and actively maintained today. Their supervised classifier achieves a fair accuracy of 62% to 73%.

Gugelmann et al. (Gugelmann et al., 2015) present a supervised classifier in 2015, fed with labeled HTTP/S traffic data captured and "recorded at the upstream router of a university campus" generated by 15K clients in one month. The authors only include services that communicated with at least five clients to address severe implied privacy-related concerns. IP addresses were anonymized for their analysis. It remains unanswered whether users gave consent to the recording or if an opt-out mechanism was offered. Besides the privacy-related issues, the authors' data set has further limitations, as HTTP/S traf-

fic could not be considered. For their feature vector, Gugelmann et al. compute the number and size of HTTP/S requests, the average number of requests to services, requests per client, and so-called "compound features" (like the number of third-party requests or amount of requests containing a cookie) (Gugelmann et al., 2015). For the labeling, they used EasyList and a manual labeling process. Their supervised machine learning algorithm achieved a precision of 80%.

Another publication in 2015 was published by Li et al. (Li et al., 2015). They feed a supervised classifier with data collected by crawling the Alexa.com top 10K global websites. They explicitly state that their crawler only visited the homepages of the selected websites. They solely considered the cookie information transmitted in the HTTP/S request and response headers. To train their classifier, Li et al. extract the minimum lifetime of a cookie, the number of third-party cookies in a set of cookies, and "augmented lifetime", which considers the length of cookie values (Li et al., 2015). They manually label third-party hosts in their training and test data set, each consisting of 500 requests. For this, they look up the website of the third party, use "[...] [block] lists specifically created for third-party tracker" (Li et al., 2015), and review the cookie properties. With this approach, they achieve remarkable 99.4% precision and 100% recall.

In the same year, Metwalley et al. (Metwalley et al., 2015a) used a "custom" browser extension and Selenium to crawl the Alexa.com top 200 websites multiple times to generate training data for their automated Web tracker detection algorithm. Their unsupervised machine learning approach aims to identify user identifiers among key-value pairs transmitted in the query string of an HTTP/S GET request. The authors state that their approach can be easily adapted to detect user identifiers embedded in HTTP/S POST requests or cookies. As ground truth, the authors use a compiled list of 443 different Web trackers from one of their previous studies (Metwalley et al., 2015b). Their algorithm detected 106 Web trackers present in their data set, of which 34 were not included in their list but were identified as Web trackers after a manual review.

Dudykevych et al. crawled the Alexa.com top 250 websites with a Firefox extension to log the HTTP/S traffic. They manually created accounts when necessary and signed in so that they could consider Web traffic occurring behind login forms. Their crawler then opened all links on a website and repeated this action three times on each opened link. They used website categories from Ghostery and IBM X-Force Exchange to label their data. Their supervised classi-

fier uses statistical characteristics of third-party cookies as features and achieves an accuracy of 95%.

Yu et al. are researchers from the developers of the Cliqz browser, a privacy-focused browser. They analyzed Web traffic generated by 200K users for seven days. A similar approach to the one of Metwalley et al. (Metwalley et al., 2015a) is used for their Web tracker detection algorithm. They investigate query parameters of third-party HTTP/S requests for cookie and browser information. HTTP/S requests that disclose information to third parties are marked “unsafe” and will not be executed. Yu et al. introduce two metrics to evaluate their approach: *protection coverage* and *site breakage*. The authors claim a higher protection coverage than Disconnect, (Disconnect Inc., 2021), which is publicly available. The reload rate of websites is used to quantify the site breakage, i.e., dysfunctions of websites caused by interfering with the intended execution of their algorithm. For this, the authors assume that a broken website is reloaded more often than a functional one.

2.2 Stateless Web Tracking: JavaScript API Access Events and Code

In 2013, Bau et al. (Bau et al., 2013) developed a supervised machine learning algorithm crawling the Quantcast (an online marketing company) top 32K websites of the United States with *FourthParty*, which is an auditing tool developed by Mayer et al. (Mayer and Mitchell, 2012) in 2012. The domain list is not publicly available. Their crawler followed five links on each website to emulate a browsing experience. The paper misses a formal definition of their feature vector, but a graph reflects connections between websites based on referenced JavaScript files. For the labeling, the authors used “a popular block list”, adding the note that this list was “manually edited” (Bau et al., 2013). Their algorithm achieves an unweighted precision of 43% to 54%. They found that the majority of trackers only appeared six times or fewer. This is in line with the “long but thin tail of Web trackers” found by Englehardt and Narayanan (Englehardt and Narayanan, 2016). The authors use this circumstance to motivate a weighting function leading to a weighted precision of 96.7% to 98%.

In 2016, Kaizer et al. (Kaizer and Gupta, 2016) developed a supervised machine learning algorithm that classifies URLs as tracker or non-tracker based on JavaScript property access (the navigator and screen interface). Using a Firefox add-on, they crawled several children-targeted websites taken from three different sources: the former open directory project *dmoz* (Schild, 2021) (now *curlie.org*), “[a] combina-

tion of the websites from the top children focused mobile applications on the Google Play store and the Apple iTunes store” (Kaizer and Gupta, 2016), and a random selection of 500 out of the Alexa.com top 5K websites. The domain list had not been made available. As ground truth, the authors used a combination of community-curated blocklists: EasyList and EasyPrivacy, “MalwareByte’s ATS ad/tracking server lists”, which are not available anymore, and “YoYo Ad-network list”, which is still available online. Their machine learning algorithm classifies 97.7% of Web trackers accurately.

The classifier of Wu et al. (Wu et al., 2016), which they presented in the same year, investigates the impact of JavaScript, which is suspected of tracking the user, on the implied third-party HTTP/S communication. Their classifier considers a 505-dimensional vector for each JavaScript file, which holds counts of JavaScript API calls the script initiated. Labeling of tracking JavaScript code was realized with EasyList, EasyPrivacy, and Ghostery (Ghostery GmbH, 2021). Their crawler repeatedly visits the website with specific JavaScript files disabled to extract and block HTTP/S requests triggered by trackers. The authors assume that the then missing requests can be safely blocked. Wu et al. crawled the Alexa.com top 10K websites using a browser with a modified WebKit to log JavaScript API calls. This approach achieved accuracy greater than 95% for various classifying algorithms they tested.

In 2017, Ikram et al. (Ikram et al., 2017) studied the similarity of JavaScript code that implements Web tracking techniques. They crawled 95 websites of the Alexa.com top 50 and a selection of 45 random websites within the range of the top 5K to 45K. Ikram et al. used Selenium to extract all embedded and referenced JavaScript code from a website’s DOM tree. They collected 2,612 JavaScript programs, which they manually labeled according to a self-defined process consisting of 12 rules. As they were only interested in the JavaScript code, focusing on static references is reasonable, yet additional JavaScript could be loaded dynamically. Their classifier determines the similarity between script files based on natural language processing models. They can classify 99% of JavaScript code accurately as tracking code.

In 2019, Iqbal et al. (Iqbal et al., 2019) developed a graph-based machine learning approach, which they call *AdGraph*. Their algorithm aims to detect advertisements and Web tracking. They crawl the Alexa.com top 10K websites to generate a training and test data set. They modify Chromium to log DOM-changing events and the JavaScript file that

caused them. Due to the graph representation, a set of structural features (like graph size, degree, number of siblings) is considered besides multiple characteristics addressing the contents of a request (like request type, length of the URL, or ad keywords in request). As ground truth, Iqbal et al. use a combination of eight different lists: EasyList, EasyPrivacy, Anti-Adblock Killer, Warning Removal List, Blockzilla (not available anymore), Fanboy Annoyances List, Peter Lowe's List (called YoYo Ad-network list in (Kaizer and Gupta, 2016)), and Squid Blacklist (not available anymore). Their machine learning algorithm achieves an accuracy of 95.33%. Iqbal et al. also measured the site breakage of their approach for which manual inspection was used.

In 2021, Rizzo et al. (Rizzo et al., 2021) developed a classifier to detect JavaScript code that generates fingerprints. To generate training and test data sets, they acquired users who had to install the software (Ermes Proxy (Department of Electronics and Telecommunications and Politecnico di Torino, 2021)) that intercepted and recorded HTTP/S traffic for one month. Impressively, 982 users volunteered for this study. The authors state that data privacy regulations and means of data protection were considered and applied. The authors claim that actual user data offers them access to JavaScript code, which they would not be able to crawl when using crawled-based data. Rizzo et al. also use OpenWPM to crawl the Alexa.com top 1 million websites to log JavaScript API calls. The retrieved JavaScript files are transformed into Abstract Syntax Trees (ASTs) to detect fingerprinting patterns in the code (similar to (Iqbal et al., 2021)). The authors also used simple string matching to support this process. The counts of detected patterns in a file are used as a feature vector for the classification. As ground truth, the authors follow a manual inspection process but include blocklists (Disconnect, EasyList, and EasyPrivacy). The authors miss 40% of code files in a subsequent process step to load it from the server due to the content not being available anymore. Despite this data loss, their classifier accurately identifies 94% of fingerprint-generating JavaScript code.

3 CONCEPT: T.EX-GRAPH

This section discusses the concept of *t.ex-Graph*. We, therefore, explain how a directed graph can be derived from a series of HTTP/S requests, how data flows can be modeled, and which features our model uses. See Figure 1 for a visualization of a subgraph we extracted.

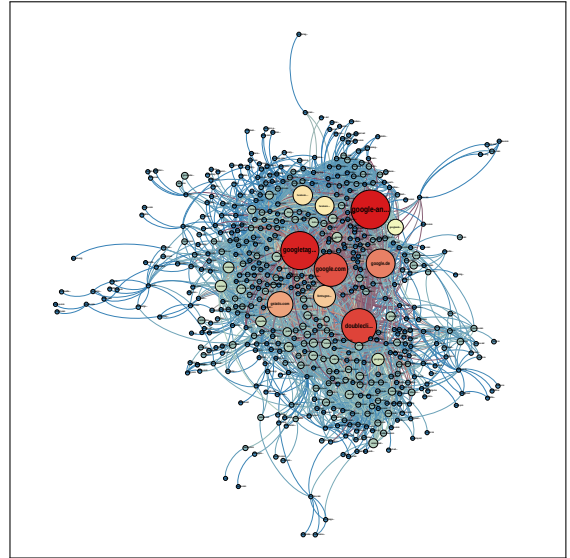


Figure 1: Visualized subgraph of *t.ex-Graph* using the *SLD* data set. Nodes with high centrality tend to carry out Web tracking activities.

3.1 Relationships Between Hosts

An HTTP/S request is always addressed to a specific *target* host. Users do not initiate most requests, e.g., when they enter a URL into the address bar. They are triggered by the browser, which assembles the requested website (in the following *first party*) by fetching multiple resources, often from entirely different hosts (in the following *third party* or *third parties*).

For these browser-initiated HTTP/S requests, we can model the first party as *source* of a request. Together with the *target* address, we derive a directed graph $G := (V, E)$ with hosts as vertices and requests as edges. Assuming a website *a.com*, which embeds an image of website *b.com*, then $a.com, b.com \in V$ and $((source, a.com), (target, b.com)) \in E$.

Hosts are addressed by fully qualified domain names (FQDN), including a host's second-level domain (SLD). All reachable subdomains (or FQDNs) share one SLD. The structure of the graph G and its properties vary, depending on how we model hosts, either as FQDNs or SLDs; thus, we derive two variants of the graph: $G_{FQDN} := (V_{FQDN}, E_{FQDN})$ and $G_{SLD} := (V_{SLD}, E_{SLD})$. Consider a website *c.com*, which embeds an image from *image.d.com*, then $((source, c.com), (target, image.d.com)) \in E_{FQDN}$, but $((source, c.com), (target, d.com)) \notin E_{FQDN}$. However, for G_{SLD} $((source, c.com), (target, d.com)) \in E_{FQDN}$ applies. More abstractly, G_{SLD} is an aggregation of G_{FQDN} .

We only consider edges where $source \neq target$; consequently, nodes do not have self-loops, and nodes

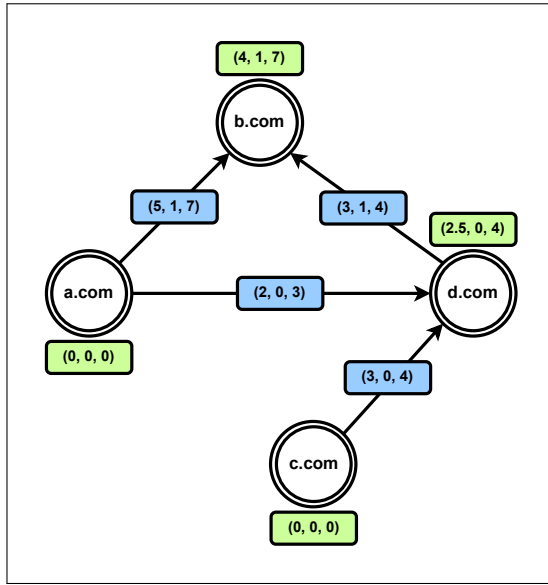


Figure 2: Concept of t.ex-Graph: depicting the computation of node attributes and the accumulation of edge attributes at the node level.

with no edge to another target are neglected (i.e., $\notin V$). Generally speaking, we deem first-party requests benign and only third-party requests as potential tracking requests. The motivation for this is that Web trackers like advertising networks or analytics providers have websites themselves, which a user might intend to visit. First-party communication in this context is benign. However, in a different context (i.e., while visiting a different website) might not. The edge attributes of self-loops would add noise to the node attributes, which could affect the results of our classifier. Nodes with no connection to another node but themselves can be considered *harmless* because the website visit causes no data flow to third parties. However, the website might still be malicious beyond Web tracking capabilities (e.g., malware or crypto mining scripts).

3.2 Modeling Data Flows

The goal of our graph is to represent the data flows between hosts. Hence, we compute attributes of edges, which, in the second step, are accumulated at the graph nodes. We extend our vertices $v \in V$ and edges $e \in E$ by a tuple $attrs$. The attributes at the edge level are derived from the HTTP/S requests and can be *counts*, *sums* (e.g., of URL lengths), or *maxima* of features of a request. The edge attributes are aggregated at the node level. We, therefore, accumulate for each node the attributes of all *incoming* edges. For this, we either derive the *sum*, a *maxima*, or a *boolean*

value encoded in 0 (for false) or 1 (for true). The sum of features can be further divided by the number of requests or in-neighbors.

See Figure 2 for an example: the graph has four nodes (a.com, b.com, c.com, and d.com) and four edges. We define $e_{a \rightarrow b}$ as the edge $((source, a.com), (target, b.com), (attrs, (2, 0, 3))) \in E$ and the remaining edges analogously. The first entry of the tuple $attrs$ is a count, the second a boolean value, and the third a sum. The node attributes for a.com and c.com are (0, 0, 0) since these nodes have no incoming edges. The first entry of the node attributes is the average of all first entries of edge attributes, the second uses the *OR* operator, and the third is choosing the maximum, analogously. Thus, the node attributes of b.com and d.com are (4, 1, 7) and (2.5, 1, 4), respectively.

3.3 Edge and Node Attributes

Generally, in an HTTP/S request, there are five elements of an HTTP message in which arbitrary data can be transmitted: (i) the HTTP body as most obvious, (ii) the URL itself (as part of a dynamic path), (iii) the query parameters (or search string), and (iv) the request headers including (v) the cookie fields. We further consider the HTTP/S response since it can hold valuable information about the target host. Although technically, this constitutes a data flow from target to source, we see the response as a deterministic part of a request. See below a detailed list of attributes we compute for each node:

count - The total number of HTTP/S requests a host retrieves from all its in-neighbors.

tracking - The total number of tracking requests a host retrieves divided by *count*.

requestType - The interface `webRequest` of the WebExtensions standard classifies each HTTP/S request into one of the following categories: `xmlhttprequest`, `image`, `font`, `script`, `stylesheet`, `ping`, `sub_frame`, `other`, `main_frame`, `csp_report`, `object`, `media` (Chrome Developers, 2022). For each request type, we compute the total number of requests with that specific type and divide it by *count*. Consequently, we compute for each type a different node attribute.

requestMethod - For each HTTP/S method, we compute the total number of requests (issued with the respective request method) and divide it by *count*. HTTP/S request methods are: GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, TRACE, and PATH (MDN Web Docs, 2022a). Analogously

to *requestType*, we compute a distinct node attribute for each type.

firstPartyDisclosed - We count the number of HTTP/S requests in which the first party is disclosed to the third party via the *Referer* or *Origin* header. This number is divided by *count* to compute the ratio of requests in which the target knows the source to the total number of requests.

cookiesSet - We count the number of cookies set by a node and divide this number by the indegree of the node.

thirdPartyCookie - We count the number of third-party cookies (i.e., *source* \neq *target*) set by a node. This count is divided by the corresponding node's total number of cookies. To be more precise, we compute the ratio of third-party cookies to all cookies set by a node.

avgUrlLength - The average URL length of all incoming requests. For this, we sum up the lengths of all URLs targeted to a specific host (node) and divide it by *count*.

avgReqPerNeighbor - The average number of incoming HTTP/S requests per in-neighbor. For this, a node's *count* is divided by its indegree.

avgQpPer - The number of query parameters of all incoming HTTP/S requests are summed up and divided by *count* and the indegree.

avgRhPer{Req | Neighbor} - The average number of request headers per HTTP/S request. We sum up the number of header fields transmitted to the target and divide it by *count* and the indegree.

avgRespHPer{Req | Neighbor} - The average number of response headers per HTTP/S request analogously computed to *avgRhPerReq* & *avgRhPerNeighbor*.

avgCookieFieldsPer{Req | Neighbor} - The average number of cookie fields per HTTP/S requests analogously computed to *avgRhPerReq* & *avgRhPerNeighbor*.

maxSubdomainDepth - The maximum subdomain depth of a node. For G_{FQDN} , this depth equals the subdomain depth of the node itself, while for G_{SLD} , the deepest subdomain is computed from all incoming requests. For example, consider *a.b.c.d.com* whose subdomain depth is 3. In G_{FQDN} *a.b.c.d.com* $\in V_{FQDN}$ and $\notin V_{SLD}$ applies, however, in G_{SLD} the subdomain depth of *d.com* $\in V_{SLD}$ is 3.

avgSubdomainLength - The average subdomain length per HTTP/S request or, more precisely, the ratio of subdomain lengths to *count*.

avgPathLength - The average path length of an HTTP/S request. The *pathname* of a URL is the URL without the FQDN, and the search string (MDN Web Docs, 2022b).

3.4 Centrality Metrics

We extend the node attributes by common graph centrality metrics, which we compute for each node. See below a list of the attributes:

in-, out-, degree - For each node the in-, out-, and degree centrality is computed.

eccentricity - The *eccentricity* of a node $\epsilon(v)$ is defined by the greatest distance of a node to any other node. More precise, $\epsilon(v) := \max d(v, u)$, $\forall u \in V$, where $d(v, u)$ is the distance between two nodes.

closenessCentrality - For each node v there is a shortest path between v and all other nodes in the network. The average of shortest path lengths between node v and $\forall u \in V$ is called the *closeness centrality* of node v .

harmonicClosenessCentrality - The *harmonic closeness centrality* is a *closeness centrality* variant, which performs better on disconnected graphs. Usually, both metrics strongly correlate. However, our *tex-Graph* contain subgraphs, which are not connected; thus, both metrics are considered.

betweennessCentrality - The *betweenness centrality* identifies nodes, which connect two clusters. It is determined for each node v by generating the shortest paths between all node pairs $(s, t) \in V$ and counting how often a node v is included in the shortest path. This count is divided by the total number of shortest paths between s and t .

eigenCentrality - The *eigenvector centrality* aims to rank nodes in the graph according to their *importance*. For this, the *importances* of the neighbors are also considered.

hubs & authorities - The metrics *hub* and *authority* had been proposed by Kleinberg (Kleinberg, 1999) in 1999 to, back then, rank websites. It is reasonable to apply this metric, designed for the Web, to our graph.

pageRanks - Similar to Kleinberg's intention, Page et al. (Page et al., 1999) implemented the famous *PageRank* algorithm in 1999 to rank websites according to their *relevance*. As an additional centrality metric specifically designed for the Web, we expect it to be well-suited for our goal.

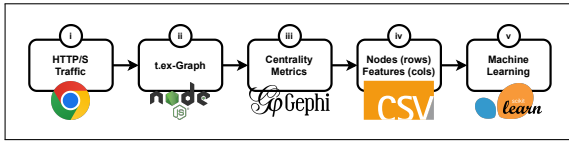


Figure 3: Overview of the processing pipeline depicting the transformation from HTTP/S traffic to a graph, whose nodes feed a machine learning classifier.

componentNumber & strongCompNum - Each node is assigned an ID of the *graph component* it is part of. Furthermore, each *strongly connected component* of the graph is identified, and an ID is assigned to its member nodes. A component of a graph is an isolated subgraph, which has no connections to other subgraphs. A strongly connected component is a subgraph in which all member nodes are directly connected. However, this subgraph might have connections to other strongly connected components.

modularityClass - By computing the *modularity* of the graph, we identify communities (or clusters) within the graph. Each is defined by a class: the *modularity class*, which is assigned to all its member nodes.

statInfClass - Another approach to detect communities in a graph is through statistical inference. We define *statistical inference classes*, which we assign to the corresponding nodes, analogously to the *modularity class*.

clustering - We compute the *clustering coefficient* for each node v , which is defined by the ratio of the actual number of edges between all neighbors of v to the total number of possible edges between all neighbors of v .

4 DATASET & ANALYSIS

In this section, we discuss the data collection and labeling process, present the pipeline to replicate and reproduce our data generation process, and finally provide an analysis of our two datasets to motivate the design of our classifier.

4.1 Dataset

To generate our graph, we use a dataset containing many HTTP/S requests and responses, which is publicly available (Raschke, 2022b). See Figure 3 for an overview of the different processing steps. We generated this dataset in a previous study (Raschke and Cory, 2022) using our Web privacy measurement framework called T.EX (Raschke, 2022a) to crawl the

Tranco (Le Pochat et al., 2019) top 10K websites with three different browsers (Chrome, Firefox, Brave) simultaneously. The first version of T.EX visualized the collected HTTP/S requests in a graph (Raschke et al., 2019), while in its most recent version (v3.2.0), the data visualization capabilities are absent. We extend our initial approach, hence, the name t.ex-Graph.

The dataset contains six separate crawls for each of the three browsers (except Firefox, which only has four). To generate our dataset, we only use one crawl conducted with Chrome (Chrome-run-1). In this crawl, 891,276 HTTP/S requests and responses (i) were recorded (including erroneous requests). We use the latest version of T.EX to export the dataset into JSON files. During this export process, the requests are labeled by T.EX using EasyList and EasyPrivacy. From this exported data, we generate our two t.ex-Graph variants G_{FQDN} and G_{SLD} with a Node.js application (ii) we developed for this purpose. This application encodes our graph in a format that can be processed by Gephi (a graph visualization and analysis tool). We extract 25,273 and 13,737 nodes for G_{FQDN} and G_{SLD} respectively. We use Gephi (iii) to compute the centrality metrics. Finally, we export two CSV (iv) files (for each variant) with nodes as rows and features as columns.

4.2 Data Analysis

We use the two variants of t.ex-Graph: the one in which nodes represent FQDNs, and the one, in which nodes represent SLDs. We derive a label *binary_tracker* from the column *tracking*, representing the ratio of incoming tracking requests to the total number of incoming requests (in the following: *tracking_ratio*). We label a node as *tracker* (1) if the ratio is greater than threshold t . The label *non-tracker* (0) is assigned to nodes with *tracking_ratio* $\leq t$. We choose $t = 0.5$ to label a node as a tracker if it receives more tracking than non-tracking requests.

4.3 Imbalanced Tracker Distribution

We investigate the distribution of *tracker* and *non-tracker*. As shown in Figure 4, most nodes are labeled as *non-tracker*, i.e., these nodes do not retrieve more tracking than non-tracking requests. The distribution varies between the two variants of t.ex-Graph. For the $FQDN$ dataset, we observe roughly a 70% to 30% split, while for the SLD dataset, we see an 80% to 20% split into non-tracker and tracker, respectively. Possible explanations for the decreased share of trackers in the SLD dataset are that (i) trackers use more subdomains than non-trackers and that (ii) the aggre-

Table 1: Distributions of the `tracking_ratio` in the two datasets. The table shows that nodes either retrieve only or no tracking requests at all. This circumstance supports the design of a binary classification task.

	<code>tracking_ratio = 0</code>	<code>1 > tracking_ratio > 0</code>	<code>tracking_ratio = 1</code>	<code>binary_tracker = 0</code>	<code>binary_tracker = 1</code>
t.ex-Graph (FQDN)	65.00	3.33	31.67	67.63	32.37
t.ex-Graph (SLD)	76.52	7.45	16.04	82.13	17.87

gation at SLD-level (i.e., the overall `tracking_ratio` of all fully qualified domain names) fosters lower tracking ratios.

An example for (i) is `online-metrix.net`, which appears in the *FQDN* dataset multiple times with fully qualified domain names like `o7f[...]295aml.e.aa.online-metrix.net`. This tracker uses random but unique subdomains for each website that connects to `online-metrix.net`. We found that this URL belongs to a service called *ThreatMetrix* offered by the company *LexisNexis*, which promises "fraud prevention solutions" (LexisNexis, 2022). However, it is listed by EasyPrivacy (easylist, 2022) and thus interpreted as a tracker by us.

An example for (ii) is `azure.com`, whose subdomain `applicationinsights.` is listed by EasyPrivacy (easylist, 2022). However, the *SLD* dataset `azure.com` has a `tracking_ratio` of 0.37. This circumstance highlights the weakness of Web tracker detection at the SLD level: Generally speaking, tracking activity can be justified by providing enough benign functionality.

4.4 Bimodal Distribution

For both datasets, we observe a strongly bimodal distribution, i.e., nodes either retrieve no or only tracking requests. The table below shows that for the *FQDN* dataset, the share of nodes with `1 > tracking_ratio > 0` is 3.33%, and for the *SLD* dataset, 7.45%. The structure of EasyList and EasyPrivacy can explain this circumstance: both lists heavily rely on filter rules based on the domain name (eyeo GmbH, 2022; Sny-

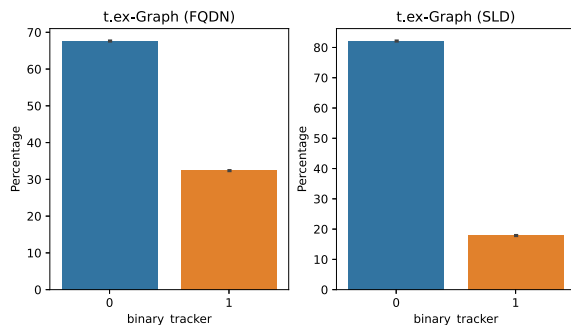


Figure 4: Distribution of `binary_tracker` (0 = blue and 1 = orange), which is derived from the `tracking_ratio` of a node.

Table 2: Classification results of the selected machine learning models for the *FQDN* dataset.

t.ex-Graph (FQDN)	accuracy	precision	recall	f1_score
XGBClassifier	0.883	0.867	0.867	0.867
RandomForestClassifier	0.881	0.862	0.871	0.866
GradientBoostingClassifier	0.858	0.836	0.856	0.844
SVC	0.840	0.819	0.852	0.828
AdaBoostClassifier	0.840	0.816	0.836	0.824
DecisionTreeClassifier	0.834	0.810	0.819	0.814
KNeighborsClassifier	0.833	0.809	0.820	0.814
LogisticRegression	0.809	0.789	0.822	0.796

Table 3: Classification results of the selected machine learning models for the *SLD* dataset.

t.ex-Graph (SLD)	accuracy	precision	recall	f1_score
XGBClassifier	0.880	0.790	0.810	0.799
RandomForestClassifier	0.875	0.781	0.826	0.800
GradientBoostingClassifier	0.861	0.763	0.827	0.787
AdaBoostClassifier	0.856	0.754	0.805	0.774
DecisionTreeClassifier	0.851	0.745	0.782	0.761
KNeighborsClassifier	0.831	0.730	0.813	0.755
LogisticRegression	0.820	0.731	0.842	0.756
SVC	0.819	0.719	0.806	0.744

der et al., 2020). We also observe evidence for (ii) of the previous section: the majority of the 7.45% of nodes (in the *SLD* dataset) has a `tracking_ratio` greater than 0 but lower or equal to t , thus, is labeled with 0, while only 1.83% of these nodes have a `tracking_ratio` greater than t and are labeled with 1 consequently.

Based on this bimodal distribution, it is reasonable to model Web tracking detection as a binary classification task. Only for a small fraction of nodes `1 > tracking_ratio > 0` applies. Alternatively, the threshold t , used to derive the labels, can be chosen differently to distribute these nodes equally (with `1 > tracking_ratio > 0`) among the two classes.

5 EVALUATION

Since all features are numeric values, we perform no feature encoding. We eliminate columns that only have 0 as a value for each row. Generally, we do not interpret 0 values as an absence of information. Thus, we do not address missing data in our datasets with imputation, for example. However, we acknowledge that our datasets are relatively sparse with a density of 0.39 and 0.36 for the *FQDN* and *SLD* datasets, respectively. Despite this sparsity, we decided against eliminating sparse features since the density leans towards 0.5. We expect the performance of some ma-

chine learning models to be affected by this circumstance. Therefore, we chose additional models which are more robust to sparse data.

As already discussed, in contrast to our features, which are entirely numerical values, we design our target variable, which shall be predicted by the classifier, as a binary variable (0 for non-, and 1 for tracker). Initially, we intended to model the Web tracking problem as a regression problem, in which the model should predict the *tracking_ratio*. However, the bimodal distribution discussed in Section 4 shows clearly that a binary classification problem is better suited than a regression problem. To address the imbalanced distribution of non- and tracker, we use Synthetic Minority Over-sampling TEchnique (SMOTE) (Chawla et al., 2002), which balances the two classes by adding synthetic values to the minority class while under-sampling the majority class. We use a variant of SMOTE, which is called Borderline-SMOTE (Han et al., 2005), which over-samples the minority class with instances near the borderline of the two classes. SMOTE is only applied to the training data.

We use 80% of the data for training and 20% for testing. We use the following machine learning models for the prediction: logistic regression (LR), k-nearest neighbors (k-NN), support vector machines (SVC), decision tree (DC), random forest (RF), adaptive boosting (AdaBoost), and gradient boosting (GradientBoost) including its optimized variant eXtreme gradient boosting (XGBoost).

5.1 Results

As we can see in Table 2 and Table 3, our classifier achieves high accuracy between 80% and 88%, with XGBoost achieving 88% accuracy on both datasets. The RF model achieves 88.1% and 87.5% accuracy on the *FQDN* and *SLD* datasets, respectively. The third model with high accuracy on both datasets is GradientBoost, with 86.1% and 85.8% accuracy. On the lower end of the performance scale, we find LR and k-NN with accuracy scores below 85% for both datasets. Surprisingly, SVC performs much better on the *FQDN* dataset (84%) while achieving the lowest accuracy (81.9%) on the *SLD* dataset. The models AdaBoost and DC achieve moderate performances and are both outperformed by their more advanced related models, GradientBoosting and RF, respectively. Furthermore, we observe a significantly lower precision achieved by all models on the *SLD* dataset. A much higher false positive rate can explain this circumstance. The lower precision affects the F1 scores, which are also significantly lower.

5.2 Discussion

Our model can predict tracking nodes with high accuracy. We observe that XGBoost and RF are well-suited models for the classification task. Furthermore, the performance results clearly show that fully qualified domain names are more suitable for detecting Web trackers. All models had significantly lower precision and, therefore, lower F1 scores on the *SLD* dataset, which leads to more false positives. The *tracking_ratio*, as a metric to quantify “how much” a node tracks, has proven insignificant for the classification task.

An investigation of the false positives and negatives reveals that our classifier can identify yet unknown trackers. However, it classifies many content delivery networks (CDNs) as Web trackers. CDNs have a high centrality since websites embed resources from CDNs, which promise low latency delivery to end users. However, we want to point out that CDNs (i) might deliver resources like fingerprinting scripts and (ii) aggregate user statistics to finance their services. Filter lists avoid blocking CDNs to avoid site breakage. Thus, our classifier is partially correct, as these nodes hold the potential for carrying out Web tracking activities. In future works, a thorough analysis of the misclassifications is needed to evaluate the quality of the ground truth.

6 CONCLUSION

This paper presents t.ex-Graph, a network graph that models data flows among hosts to detect nodes that carry out Web tracking activities. We thoroughly discussed features of nodes extended by standard centrality metrics of graphs to automatically detect with machine learning whether a node is a Web tracker. Our results show that t.ex-Graph achieves high accuracy on large datasets extracted from crawling the Tranco top 10K websites. Our components are publicly available to replicate and reproduce our results. We encourage fellow researchers to extend, modify, and optimize our approach to achieve better results. In future work, we plan to enrich nodes with external information to add more semantics to the graph, like website categories, the company name, which registered the domain, or whether a host is a CDN. We are confident that our approach can detect new trackers and constitutes an alternative to the tedious manual inspection of network traffic.

ACKNOWLEDGMENTS

This work has been carried out within the project TRAPEZE. The TRAPEZE project receives funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 883464

REFERENCES

- Bau, J., Mayer, J., Paskov, H., and Mitchell, J. C. (2013). A Promising Direction for Web Tracking Countermeasures. page 5, San Francisco, US. IEEE.
- Chawla, N. V., Bowyer, K. W., Hall, L. O., and Kegelmeyer, W. P. (2002). SMOTE: Synthetic Minority Over-sampling Technique. *Journal of Artificial Intelligence Research*, 16:321–357.
- Chrome Developers (2022). chrome.webRequest.
- Department of Electronics and Telecommunications and Politecnico di Torino (2021). Ermes Project.
- Disconnect Inc. (2021). Disconnect.me.
- easylis (2022). EasyPrivacy.
- Englehardt, S. and Narayanan, A. (2016). Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*, pages 1388–1401, Vienna, Austria. ACM Press.
- Esfandiari, B. and Nock, R. (2005). Adaptive Filtering of Advertisements on Web Pages. In *Special Interest Tracks and Posters of the 14th International Conference on World Wide Web, WWW '05*, pages 916–917, New York, NY, USA. Association for Computing Machinery. event-place: Chiba, Japan.
- eyeo GmbH (2022). How to write filters | Adblock Plus Help Center.
- Ghostery GmbH (2021). Ghostery.
- Gugelmann, D., Happe, M., Ager, B., and Lenders, V. (2015). An Automated Approach for Complementing Ad Blockers' Blacklists. In *Proceedings on Privacy Enhancing Technologies*, volume 2015, pages 282–298, Philadelphia, PA, USA. Sciencdo. Section: Proceedings on Privacy Enhancing Technologies.
- Han, H., Wang, W.-Y., and Mao, B.-H. (2005). Borderline-SMOTE: A New Over-Sampling Method in Imbalanced Data Sets Learning. In Huang, D.-S., Zhang, X.-P., and Huang, G.-B., editors, *Advances in Intelligent Computing*, Lecture Notes in Computer Science, pages 878–887, Berlin, Heidelberg. Springer.
- Ikram, M., Asghar, H. J., Kaafar, M. A., Mahanti, A., and Krishnamurthy, B. (2017). Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning. In *Proceedings on Privacy Enhancing Technologies*, volume 2017, pages 79–99, Minneapolis, USA. Sciencdo. Section: Proceedings on Privacy Enhancing Technologies.
- Iqbal, U., Englehardt, S., and Shafiq, Z. (2021). Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1143–1161. ISSN: 2375-1207.
- Iqbal, U., Snyder, P., Zhu, S., Livshits, B., Qian, Z., and Shafiq, Z. (2019). AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *2020 IEEE Symposium on Security and Privacy*, pages 763–776, Online. IEEE.
- Kaizer, A. J. and Gupta, M. (2016). Towards Automatic Identification of JavaScript-oriented Machine-Based Tracking. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics, IWSPA '16*, pages 33–40, New York, NY, USA. Association for Computing Machinery.
- Kleinberg, J. M. (1999). Authoritative sources in a hyper-linked environment. *Journal of the ACM*, 46(5):604–632.
- Kushmerick, N. (1999). Learning to Remove Internet Advertisements. In *Proceedings of the Third Annual Conference on Autonomous Agents, AGENTS '99*, pages 175–181, New York, NY, USA. Association for Computing Machinery. event-place: Seattle, Washington, USA.
- Le Pochat, V., Van Goethem, T., Tajalizadehkhoob, S., Korczynski, M., and Joosen, W. (2019). Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings 2019 Network and Distributed System Security Symposium*, San Diego, CA. Internet Society.
- LexisNexis (2022). ThreatMetrix - Cybersecurity Risk Management.
- Li, T.-C., Hang, H., Faloutsos, M., Efstathopoulos, P., Faloutsos, M., and Efstathopoulos, P. (2015). Track-Advisor: Taking Back Browsing Privacy from Third-Party Trackers. In *International Conference on Passive and Active Network Measurement*, volume 8995, pages 277–289, Cham. Springer International Publishing. Series Title: Lecture Notes in Computer Science.
- Mayer, J. R. and Mitchell, J. C. (2012). Third-Party Web Tracking: Policy and Technology. In *2012 IEEE Symposium on Security and Privacy*, pages 413–427. ISSN: 2375-1207.
- MDN Web Docs (2022a). HTTP request methods - HTTP | MDN.
- MDN Web Docs (2022b). URL.pathname - Web APIs | MDN.
- Metwalley, H., Traverso, S., and Mellia, M. (2015a). Un-supervised Detection of Web Trackers. In *2015 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, San Diego, CA, USA. IEEE.
- Metwalley, H., Traverso, S., Mellia, M., Miskovic, S., Baldi, M., Mellia, M., Miskovic, S., and Baldi, M. (2015b). The Online Tracking Horde: A View from Passive Measurements. In *TMA 2015: Traffic Monitoring and Analysis*, volume 9053, pages 111–125, Barcelona, Spain. Springer International Publishing.
- Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The PageRank Citation Ranking: Bringing Order to the Web. Techreport. Publisher: Stanford InfoLab.

- Raschke, P. (2022a). T.EX - The Transparency Extension.
- Raschke, P. (2022b). Tranco 16-5-22 top 10K crawled with T.EX. Version Number: 1.0 Type: dataset.
- Raschke, P. and Cory, T. (2022). Presenting a Client-based Cross-browser Web Privacy Measurement Framework for Automated Web Tracker Detection Research. In *2022 3rd International Conference on Electrical Engineering and Informatics (Icon EEI)*, pages 98–103.
- Raschke, P., Zickau, S., Kröger, J. L., and Küpper, A. (2019). Towards Real-Time Web Tracking Detection with T.EX - The Transparency EXtension. In Naldi, M., Italiano, G. F., Rannenberg, K., Medina, M., and Bourka, A., editors, *Privacy Technologies and Policy*, Lecture Notes in Computer Science, pages 3–17, Cham. Springer International Publishing.
- Rizzo, V., Traverso, S., and Mellia, M. (2021). Unveiling Web Fingerprinting in the Wild Via Code Mining and Machine Learning. In *Proceedings on Privacy Enhancing Technologies*, volume 2021, pages 43–63, Online. Sciendo. Section: Proceedings on Privacy Enhancing Technologies.
- Schild, T. (2021). dmoz.de.
- Snyder, P., Vastel, A., and Livshits, B. (2020). Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, volume 4, pages 26:1–26:24.
- Wu, Q., Liu, Q., Zhang, Y., Liu, P., Wen, G., Liu, Q., Zhang, Y., Liu, P., and Wen, G. (2016). A Machine Learning Approach for Detecting Third-Party Trackers on the Web. In *Computer Security – ESORICS 2016*, volume 9878, pages 238–258, Heraklion, Greece. Springer International Publishing.
- Yamada, A., Masanori, H., and Miyake, Y. (2010). Web Tracking Site Detection Based on Temporal Link Analysis. In *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pages 626–631, Perth, Australia. IEEE.