From Syntactic Matching to Taint Tracking and Back: A Comparative Study of Web Tracking Detection Techniques

Stefano Calzavara
Università Ca' Foscari Venezia
stefano.calzavara@unive.it

Samuele Casarin
Università Ca' Foscari Venezia
Scuola IMT Alti Studi Lucca
samuele.casarin@unive.it

Marco Squarcina
TU Wien
marco.squarcina@tuwien.ac.at

Matteo Maffei
TU Wien
matteo.maffei@tuwien.ac.at

Abstract

Traditional web tracking techniques rely on unique identifiers set in the client-side storage and shared with third-party trackers through network requests. Ideally, this phenomenon may be investigated through the classic lens of information flow control, e.g., by using instrumented browsers with taint tracking support. As a matter of fact though, most web privacy research makes use of simple syntactic matching heuristics that merely look for the presence of (possibly transformed) client-side identifiers within network requests, with no visibility of the JavaScript logic. In this work, we perform a comparative study of these two approaches to web tracking detection. Our investigation shows that taint tracking can expose tracking behavior that remains undetected by syntactic matching heuristics, which suffer from a significant number of false positives and false negatives. However, we also show that taint tracking is not strictly superior to syntactic matching, due to a range of different reasons, including the current limitations of state-of-the-art implementations and the complexity of real-world tracking behavior. Overall, we advocate for a critical reflection on the shortcomings of prominent web tracking detection approaches and we propose useful methodologies to improve current measurement practices.

Keywords

web tracking, web privacy, web measurements

1 Introduction

Traditional web tracking, also known as stateful tracking [28], relies on the creation of unique *identifiers* that are first saved in the client-side storage (cookies or web storage) and then communicated to third-party trackers using network requests. If the same identifier is included in multiple network requests, the tracker is able to associate them with the same client identity and build navigation profiles of the tracked client. The privacy concerns raised by this practice motivated significant attention from the research community, which presented multiple studies on the prevalence and dangers of stateful tracking [41].

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–16

https://doi.org/XXXXXXX.XXXXXXX

Conceptually, stateful tracking may be investigated through the classic lens of *information flow control* [33]. Information flow control is concerned with detecting and monitoring how sensitive data (in our case, identifiers) travel within a system, specifically tracing the movement of information from their *source* (the client-side storage) to a *sink* (the network) where disclosure occurs. Dynamic approaches to information flow control are normally based on *taint tracking*: when a sensitive value is read from its source, it is marked as tainted. The taint is then propagated through all computations that depend on the read value, ensuring that any new value derived from it also carries the taint. By maintaining this propagation at runtime, the system can detect when tainted information reaches a designated sink, allowing it to detect cases where information about sensitive data might be disclosed. Taint tracking has a wide range of applications in computer security, but has also been fruitfully applied to the web privacy setting [2, 3, 6, 9, 10, 23, 27, 37, 39, 44, 45].

As a matter of fact though, most web privacy research does not go through the complexity of taint tracking and instead relies on simple *syntactic matching* heuristics to detect web tracking. First, the client-side storage is inspected for the presence of identifiers. Syntactic matching then looks for their occurrence within network requests. Since identifiers are not necessarily communicated as is, e.g., they may be Base64-encoded before transmission, different heuristics support multiple types of transformations to expose additional tracking behavior. Although syntactic matching heuristics are popular in the literature [1, 4, 16, 18, 30, 32, 34, 40], likely due to their efficiency and simplicity, they are inherently limited by their inability to capture all possible identifier transformations. For example, assume that an identifier is encrypted before being sent to the tracker: detecting it in a network request would be impossible without access to the JavaScript logic, since the identifier is transformed in an unpredictable way. Moreover, syntactic matching can only approximate taint tracking, because it does not capture information flows, but merely relies on syntactic evidence. For example, spurious matches can occur because (part of) an identifier accidentally appears in a network request, although there is no tracking script that reads the identifier and sends it over the network.

To the best of our knowledge, there is limited understanding of the strengths and weaknesses of these approaches to web tracking detection. It is clear that syntactic matching suffers from both false positives and false negatives, given its obliviousness to the JavaScript logic. However, there is no objective evaluation of how these limitations affect web privacy analyses at scale. By comparing

the findings enabled by syntactic matching with those supported by a more principled approach like taint tracking, we shed light on the practical limitations of syntactic matching for web tracking detection. As an additional result, our analysis reveals that taint tracking is not necessarily superior to syntactic matching and is not yet ready to replace it as a more principled foundation for web privacy research, due to relevant limitations in current state-of-the-art implementations and the complexity of real-world tracking behavior. Overall, we advocate for a critical reflection on the shortcomings of prominent web tracking detection approaches and propose simple methodologies to improve over current practices.

Contributions. In this work, we make the following contributions:

(1) We construct a dataset of tracking requests detected on the Tranco Top 10k websites using representative implementations of syntactic matching (based on prior work [4, 34, 40]) and taint tracking (as implemented in the Foxhound browser [26]). In total, we collect 40,605 tracking requests, including 17,496 requests detected by syntactic matching alone, 7,021 requests identified by taint tracking alone, and 16,088 requests detected by both approaches.
(2) We provide a critical evaluation of the strengths and weaknesses of syntactic matching and taint tracking for the detection of tracking requests, based on our dataset. Our analysis shows that the two approaches are largely complementary: while syntactic matching inevitably suffers from clear sources of false positives and false negatives that taint tracking can help mitigate, even a state-of-the-art taint tracker such as Foxhound exhibits relevant shortcomings and is not ready to replace syntactic matching.
(3) We show that the observed limitations of syntactic matching and taint tracking are not an artifact of our evaluation, but rather have a significant impact on relevant inferences about web tracking, e.g., the estimation of tracker popularity. Building on the insights from our critical evaluation, we propose sound methodologies and recommendations to support web privacy analyses, providing improved analytical precision by combining multiple tracking detection techniques.

To support open science, we release all our code and data [8].

2 Background and Motivation

To better understand how syntactic matching and taint tracking operate in practice, we present a few examples of (simplified) tracking scripts and we discuss how the two approaches may enable web tracking detection or not, which motivates the complexity and relevance of the problem under study.

In all the examples, tracking is enabled by the local storage item `cid`, which is sent to a third-party tracker in different ways. We assume that the value of `cid` is `TRK-a55bd7c6`, where `TRK` is a constant string and `a55bd7c6` is a unique client identifier previously generated by the tracking script. The goal of syntactic matching and taint tracking here is exposing the presence of a *tracking request*, i.e., an HTTP request bearing an identifier set in the client-side data towards a third-party website.[1]

[1] Although not all such requests are necessarily associated with web tracking practices, we use the term "tracking requests" because they exhibit the essential features of stateful tracking.

Example 1: Direct Communication. Table 1a shows a simple scenario where `cid` is read from the local storage and communicated to the tracker unchanged. The tracking request is straightforward to detect for syntactic matching, because the value of `cid` appears as is in the `uid` parameter of the network request. Similarly, taint tracking readily detects the tracking request, because the `uid` variable is tainted after the local storage is read and the taint reaches a network sink in the next line of code.

Example 2: Encoding. Table 1b shows a variant where `cid` is Base64-encoded before network communication. This case can be handled by syntactic matching by trying to anticipate possible transformations performed during JavaScript execution. If Base64 encoding is supported, syntactic matching will generate the correct Base64 encoding of `TRK-a55bd7c6` and detect it in the `uid` parameter of the network request, hence identifying the tracking request.

Of course, the generality of this approach is inherently limited, as it relies on a predefined set of transformations and, to ensure termination, must bound the transformation depth (e.g., three layers of Base64 encoding) or the number of matching attempts. In turn, taint tracking detects the tracking request out of the box: since `btoa` operates over a tainted value, its result will be tainted, hence an information flow exposing an identifier to the network is detected.

Example 3: Slicing. Table 1c shows a scenario where `cid` is sliced before transmission: the original value `TRK-a55bd7c6` is broken over the dash and just the identifier `a55bd7c6` is sent over the network. Syntactic matching can detect the tracking request if slicing on non-alphanumeric characters is supported. Taint tracking also detects the tracking request, because slicing preserves taint.

This example is interesting as it highlights a potential source of false positives for taint tracking. Sending `a55bd7c6` is enough for the tracker, since the `TRK` prefix is a constant string that does not identify the client. However, if the code instead transmitted `TRK`, taint tracking would still report an information flow, since the entire `parts` array is tainted. This is undesirable, because the transformation of the identifier led to the loss of tracking capabilities.

Syntactic matching handles these scenarios naturally, as possible transformations of identifiers are pre-computed to account for the lack of visibility into the JavaScript logic, hence it is straightforward to filter out those values that are not identifiers anymore. In turn, taint tracking requires more care, because blindly propagating taints may lead to over-reporting tracking requests and identifiers can be transformed along the way in an unanticipated manner.

Example 4: Encryption. Finally, Table 1d shows an example where the value of `cid` is encrypted using a symmetric key before network communication. Of course, syntactic matching cannot expose the tracking request here, because it has no visibility of the JavaScript logic and no access to the key, meaning that it cannot anticipate how the value of `cid` will be transformed. Taint tracking, in turn, can propagate the taint of the `uid` variable through the invocation of the encrypt method of the symmetric key and correctly identify an information flow.

3 Dataset Construction

In this section we describe the dataset we created to perform an empirical comparison of syntactic matching and taint tracking

Table 1: Representative examples of web tracking patterns.

(a) Example 1 - Direct Communication

```
const uid = localStorage.getItem("cid");
fetch("https://tracker.com/collect?uid=" + uid);
```

(b) Example 2 - Encoding

```
const uid = localStorage.getItem("cid");
const encoded = btoa(uid); // encode to Base64
fetch("https://tracker.com/collect?uid=" + encoded);
```

(c) Example 3 - Slicing

```
const uid = localStorage.getItem("cid");
const parts = uid.split("-");
fetch("https://tracker.com/collect?uid=" + parts[1]);
```

(d) Example 4 - Encryption

```
const uid = localStorage.getItem("cid");
const key = localStorage.getItem("k");
const encrypted = key.encrypt(uid);
fetch("https://tracker.com/collect?uid=" + encrypted);
```

when it comes to the detection of stateful tracking. After clarifying the high-level dataset design, we discuss how we implemented representative data collections for the two considered approaches and we conclude by reporting dataset details.

3.1 Dataset Design

Figure 1 depicts the analysis pipeline. At a high-level, our dataset must collect the tracking requests detected on real-world websites using syntactic matching and taint tracking to enable a fair comparison of the two approaches. Our dataset construction operates by accessing popular websites from the Tranco list and scrutinizing the network requests sent from their homepages using both syntactic matching and taint tracking. In particular, after accessing the homepage of each website under analysis, we operate as follows:

(1) We first inspect the client-side storage to look for the presence of identifiers, using standard detection heuristics.
(2) We perform syntactic matching over each outgoing HTTP request towards a third-party website to identify the presence of any of the identifiers found at step 1. If a match is found, we mark the request as a tracking request identified by syntactic matching.
(3) We simultaneously perform taint tracking on the JavaScript code to identify information flows from the client-side storage to an HTTP request. If the information flow reads any of the identifiers found at step 1 and leads to a request towards a third-party website, we mark the request as a tracking request identified by taint tracking.

Specifically, we consider cookies and local storage as client-side storage mechanisms, in line with standard practices [3, 32].

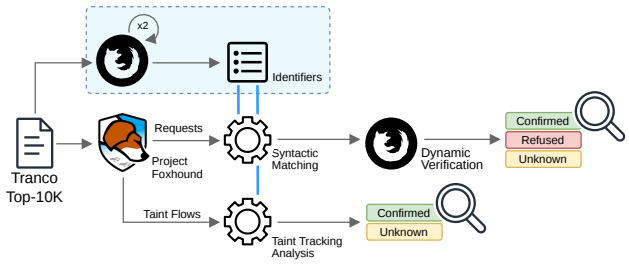

Figure 1: Analysis pipeline.

In the end, our dataset associates each analyzed homepage with a set of tracking requests annotated with the techniques that have been used to detect them (syntactic matching, taint tracking, or both). This way, we can associate each website with a corresponding list of trackers (sites receiving a tracking request) operating on its homepage, using either syntactic matching or taint tracking.

3.2 Identifier Detection

Different papers use distinct techniques to detect identifiers within client-side storage. All these techniques are empirical in nature: since the inner workings of web trackers are generally unknown a priori, researchers proposed a range of recipes to identify client-side storage items that have tracking potential. Standard checks require the content of the storage item to be *long* and *random*, which is a necessary condition for the item to be used as a user identifier on the web, with different papers instantiating the notion of *long* and *random* in various ways. Representative heuristics for identifier detection are reported in Table 2. To the best of our knowledge, there is no consensus on the most effective heuristic due to the absence of a solid ground truth. Further investigating this point is beyond the scope of this paper: since our goal is comparing syntactic matching and taint tracking for the analysis of stateful tracking, we select a representative heuristic and consistently use it in our dataset construction.

In this work, we define identifiers as strings that include at least 8 characters and have an estimated cracking cost of at least 10^9 guesses according to zxcvbn [42]. This representative technique has been used in at least two recent studies presented at major conferences [34, 35]. To improve the accuracy of identifier detection, we also leverage a filtering technique first proposed by Chen et al. [9]: before detecting identifiers, we access the web page twice using two independent clients running on the same machine, and we consider as candidate identifiers only those persistent storage items that are *significantly different* across the two clients. In fact, storage items not meeting this condition cannot be used for tracking, since they do not uniquely identify the client. Two storage items are considered *significantly different* if their Ratcliff-Obershelp similarity score is below 66%, after removing timestamps and longest common subsequences longer than 2. Storage items that do not satisfy this preliminary condition are considered as non-identifiers.

3.3 Syntactic Matching

Syntactic matching algorithms operate by looking for the presence of identifiers within network requests to expose web tracking. They

Table 2: Representative heuristics for identifier detection proposed in the literature. Columns: Lifetime = storage item lifetime. Parse = parsing/tokenization is applied (if positive, checks are performed on individual tokens). Length = number of characters. Randomness = either Ratcliff-Obershelp similarity across unrelated visits (*RO*) or guessability based on zxcvbn (*Guesses*).

Publication	Lifetime	Parse	Length	Randomness	Additional Checks
Roesner et al. [32]	> Session		–	–	Unique across unrelated visits.
Acar et al. [1]	> 30 days	✓	–	$RO < 33\%$	Same across related visits. Same length across unrelated visits.
Englehardt et al. [17]	> 90 days		–	$RO < 55\%$	Same across related visits. Same length across unrelated visits. Unique across unrelated visits.
Englehardt and Narayanan [16]	> 90 days	✓	8–100	$RO < 66\%$	Same across related visits. Unique across unrelated visits.
Papadopoulos et al. [30]	> Session	✓	≥10	–	–
Sánchez-Rola et al. [35]	–		–	$Guesses \geq 10^9$	–
Fouad et al. [18]	–	✓	–	–	–
Chen et al. [9]	> Session		≥8	$RO < 66\%$	Length evaluated after URL decoding. RO evaluated after removing timestamps and longest common subsequences longer than 2.
Sánchez-Rola et al. [34]	–		–	$Guesses \geq 10^9$	–
Bahrami et al. [4]	–	✓	≥8	–	–
Our work	> Session		≥8	$RO < 66\%$ $Guesses \geq 10^9$	Length evaluated after URL decoding. RO evaluated after removing timestamps and longest common subsequences longer than 2.

have to deal with two main challenges. First, identifiers can be transformed, e.g., using Base64 encoding, before being sent over the network, meaning that effective matching algorithms have to account for such transformations as thoroughly as possible. Second, network requests must be correctly parsed and analyzed to avoid spurious matches.

3.3.1 State of the Art. Different papers rely on distinct sets of identifier transformations and request parsing techniques to implement syntactic matching. Based on the analysis of several published papers [1, 4, 9, 16, 18, 30, 32, 34, 40], we identify common techniques considered in the literature (see Table 3) and we start from them to implement our syntactic matching algorithm. Common transformations in the state of the art include various forms of slicing, encoding and decoding operations applied throughout the syntactic matching pipeline, i.e., to transform identifiers and to parse requests. At a high level, existing syntactic matching algorithms share similar intuitions, but differ in specific implementation details, e.g., which forms of encoding are supported, which components of the network requests are scrutinized for tracking detection, and the recursion depth applied when nesting multiple steps of the matching algorithm. For example, up to three layers of encoding may be used to transform identifiers.

To the best of our knowledge, there is no consensus on which syntactic matching implementation should be considered the state of the art. This is evidenced by the varied nature of specific implementation details observed across prior work over time, often without presenting strong motivations behind specific design choices. For example, some work supports the encoding of identifiers but not their decoding; other work looks for identifiers within request parameters and ignores the path component, while other papers look into both. Rather than evaluating an extensive set of previous heuristics from the literature, which would fragment our findings and complicate interpretation, we implement a representative syntactic matching algorithm based on the union and generalization of previous proposals from the literature. As such, our goal is not to favor any specific prior design, but to capture the common design space that has emerged across prior efforts. An overview of our implementation is summarized in the last line of Table 3 and we discuss details below.

Note that, while previous work only considered encoding storage values, we support common decoding transformations for storage values as well. This allows us to detect flows where a script stores an encoded value (e.g., a Base64-encoded JSON object) and later decodes it to extract an identifier, which is attached to a network request. Such cases may be missed if identifier detection was not performed after decoding values. Moreover, two transformations from prior literature are not considered in our request parsing heuristics, i.e., *Deflate* and *Split*. Regarding Deflate, although two prior papers considered it, compression in general is not a single canonical transformation: Deflate is one possible scheme among others, and it can appear in different wrapper formats such as raw Deflate, zlib, or gzip. We therefore focus on broadly used and commonly accepted encoding/decoding functions. As for Split, we exclude it for performance reasons, as it tends to decompose request payloads into a large number of substrings that hinder the performance of syntactic matching. Based on a preliminary experiment with 1,000 websites, we observed that Split introduces an overhead of 2.5x in the analysis time, but leads to just a +0.6% increase in the number of detected tracking requests.

3.3.2 Identifier Transformation. In our implementation, we assume that identifiers set in cookies and local storage can be subject to the following transformations before network communication:

(1) *Slicing*: an identifier can include multiple sub-identifiers, possibly conforming to a format. We consider different slicing techniques, in particular: extracting values from JSON and query strings (e.g., `key1=val1&key2=val2...`), and splitting by non-alphanumeric characters (e.g., `val1|val2...`). Note that we only consider sliced components that still comply with our definition of identifier.
(2) *Encoding*: an identifier can be encoded before being communicated over the network. We consider different encoding

Table 3: Syntactic matching heuristics proposed in the web privacy literature. Transformations excluded by our heuristics at a certain step are formatted in *italics*. We use "k=v" to denote value extraction from `key1=val1&key2=val2`... strings and "Split" to represent splitting by common delimiters.

Publication	Storage Identifier Transformation				Request Parsing					
	Slicing	Encoding	Decoding	Depth	Query	Path	Body	Parsing	Decoding	Depth
Roesner et al. [32]	–	–	–	–	✓	✗	✓	–	–	–
Acar et al. [1]	Split	–	–	1	✓	✓	✗	–	–	–
Englehardt and Narayanan [16]	Split	–	–	1	✓	✓	✗	–	–	–
Papadopoulos et al. [30]	Split	–	–	1	✓	✓	✗	–	–	–
Starov and Nikiforakis [40]	–	–	–	–	✓	✗	✓	JSON, k=v	Base64-Dec, URL-Dec, *Deflate*	1K[1]
Fouad et al. [18]	Split	–	–	1	✓	✗	✗	k=v, *Split*	Base64-Dec	1
Chen et al. [9]	Split	Base64-Enc, Hashing	–	1	✓	✓	✓	–	–	–
Sánchez-Rola et al. [34]	–	–	–	–	✓	✗	✓	JSON, k=v	Base64-Dec, URL-Dec, *Deflate*	1K[1]
Bahrami et al. [4]	Split	Base64-Enc, Hashing	–	1	✓	✗	✗	k=v, *Split*	–	1
Our work	JSON, k=v, Split	Base64-Enc, URL-Enc, Hashing	Base64-Dec, URL-Dec	3	✓	✓	✓	JSON, k=v	Base64-Dec, URL-Dec	3

[1] A queue is used to iteratively apply candidate transformations and test syntactic matches, up to 1,000 attempts.

and hashing techniques: URL encoding, Base64 encoding, MD5, and SHA1.

(3) *Decoding*: an identifier may be decoded before being communicated over the network. We consider two decoding techniques: URL decoding and Base64 decoding.

We assume that up to three layers of these transformations can be applied in any order, e.g., an identifier may be first URL-decoded, then split on non-alphanumeric characters, and each of the parsed components may be Base64-encoded before network communication. Given an identifier i, we let $Trans(i)$ stand for the set of strings built using the proposed algorithm.

3.3.3 Request Parsing and Matching. The matching algorithm finally looks for (possibly transformed) identifiers within different parts of the network requests. To this end, each network request is subject to various parsing techniques to extract its individual parameters:

(1) *Query string parsing*: we extract all the values of the query string parameters of the request URL.
(2) *Path parsing*: we split the path part of the request URL by `'/'` to obtain all its segments.
(3) *Body parsing*: if the network request includes a body and the `Content-Type` request header is set to `application/json` or `application/x-www-form-urlencoded`, we parse the request body as JSON or query string, respectively, and extract all values. For all other cases, we extract the body as is.

Since those parameters can also be the result of diverse types of encoding, they may also undergo up to three layers of decoding before being inspected for syntactic matching. Notice that this is a necessary step to avoid false negatives, as applying the encoding transformations to the storage items is not sufficient to ensure the detection of all possible matches. For example, assume that an identifier is included in a JSON data structure that is Base64-encoded before being sent over the network, e.g., `base64('{"uid": 1234}')` which corresponds to the string `'eyJ1aWQiOjEyMzR9'`. In this case, applying a Base64 encoding transformation to the identifier `1234` in the storage would not help, because the resulting string `'MTIzNA=='` is not present in the network request.

Given a network request r, we let $Parse(r)$ stand for the set of strings built using the proposed algorithm. We then stipulate that r is a tracking request if and only if it is sent towards a third-party website and there exists an identifier i in the client-side storage, a string $s \in Trans(i)$, and a string $s' \in Parse(r)$ such that s is a substring of s'.

3.4 Taint Tracking

To experiment with taint tracking, we use Foxhound [26], a fork of Firefox which modifies its internal string operations to propagate taint information from a set of sources (including client-side storage) to a set of sinks (including network requests). Foxhound is a state-of-the-art tool for dynamic taint tracking of JavaScript code, as shown in a recent research paper showing its superiority over current alternatives [7]. Foxhound was already used in several web measurement studies [2, 6, 25] and thus serves as a useful baseline for the adoption of taint tracking in web privacy research.

There are a few distinct challenges to deal with when using Foxhound to investigate stateful tracking. The main source of complexity is that Foxhound tracks all the information flows from the client-side storage to a network sink. In this work, we are only concerned about flows involving identifiers and transformations that preserve their tracking potential. This means that we have to filter out (i) all the flows originating from cookies and local storage that do not contain identifiers and (ii) all the flows that transform identifiers in a way that thwarts web tracking, e.g., the length of a

tainted string is tainted according to Foxhound, because it certainly reveals information about the string itself, however it cannot be used to uniquely identify the client. Note that we use the same heuristics for identifier detection as in syntactic matching.

Implementing the first filter is apparently straightforward, but unfortunately more complicated than expected. The main problem is that the taints propagated by Foxhound have insufficient granularity, because the taint associated with cookie values does not contain any information about the cookie which was actually read. This is an artifact coming from the rudimentary access to the cookie jar available in web browsers: cookies are read by accessing the `document.cookie` property to retrieve all the cookies as a string of concatenated key-value pairs, which is subsequently parsed by JavaScript to extract the cookies of interest. We then modified Foxhound to improve taint granularity, in particular by replacing the single taint `document.cookie` with a more refined set of taints which include the cookie name. For example, assume that a script reads the first two characters of the cookie `k` and stores them in the variable `x`. In the original Foxhound, `x` would be given taint `document.cookie`, while in our revised version `x` is given taint `document.cookie[k]`.

The second implemented filter is conceptually more interesting and requires some care. Taint tracking propagates taints across all string transformations, although some of them may not preserve tracking potential (see Example 3 in Table 1c). To avoid over-reporting, we modified Foxhound once more to further enhance taint granularity. In the original version, string values returned from storage sources have the same taint for all characters, thus denying the possibility to track flows of individual characters along the execution. We then replaced the single taint associated with each character of a read storage item with a specific taint per character, essentially leading to byte-level taint tracking over the considered storage sources. Taking up the last example, in our revised version of Foxhound the variable `x` is now given taint `document.cookie[k][0:1]`, indicating that only the first two characters of the cookie `k` were stored into `x`. With this refinement in place, we leverage the taints observed in the sink invocation to identify all the characters that have been read at the source, and we only keep those flows where the concatenation of the read characters produces an identifier according to our heuristics. This way, we are able to deal with scenarios where an identifier initially is read, but just some non-identifier component reaches the sink, thus correctly dealing with cases like Example 3.

To sum up, our modifications to Foxhound refine the granularity of already supported taint flows (from cookies and local storage to common network sinks), enabling the attribution of specific request components to individual client-side identifiers at a character level. These minimal changes allow us to effectively measure stateful tracking using the current implementation of Foxhound. In the end, we identify tracking requests by running our modified version of Foxhound to collect all the information flows from the client-side storage to a network sink that pass the explained filters. This way, we target our analysis over information flows that involve identifiers and are based only on transformations that preserve tracking potential. The HTTP requests generated at the sink are marked as tracking requests.

3.5 Dataset Construction Details

To compare the relative effectiveness of syntactic matching and taint tracking for the analysis of stateful tracking, we perform an empirical evaluation based on data collected from the websites in the Tranco Top 10k [31] generated on February 8, 2026. In particular, we set up a crawler that controls an instance of Foxhound and visits each website two times to collect tracking requests.

First, the crawler navigates the website landing page and waits up to 60 seconds for the page to load, followed by an additional 10 seconds to allow dynamic content to render. This first navigation has the purpose of letting JavaScript tracking code save identifiers within the client-side storage, which is a prerequisite to stateful tracking. The crawler navigates the website landing page again, applying the same timeouts as defined above. This second navigation collects all the tracking requests detected by syntactic matching and taint tracking as explained in the previous sections. By doing so, we observe how the reuse of identifiers allows trackers to potentially correlate two visits, which is the essence of stateful tracking.

This process led to a dataset of 40,605 tracking requests from 7,614 correctly accessed websites. Syntactic matching found 33,584 tracking requests, while taint tracking exposed 23,109 tracking requests; 16,088 tracking requests were finally discovered by both detection approaches.

4 Empirical Comparison

We now use our dataset of tracking requests to better understand the relative strengths of syntactic matching and taint tracking when it comes to the study of stateful tracking on the web.

4.1 Preliminary Analysis

In an ideal world, taint tracking may discover more tracking requests than syntactic matching. The reason is that stateful tracking is based on the communication of client-side identifiers to a third-party website and a sound taint tracker would correctly propagate all the information flows from a source (the storage) to a sink (the network), irrespective of the complicated and variegated nature of the intermediate operations involved in the process. Conversely, syntactic matching is inherently limited by its inability to cover all possible transformations of identifiers, most notably when they depend on secrets that are not exposed to the matching algorithm, e.g., encryption keys (see Example 4 in Table 1d).

As a matter of fact though, our data immediately show that the tracking requests detected by syntactic matching are not a subset of those exposed by taint tracking. In particular, our dataset includes 17,496 tracking requests (43%) that are exposed by syntactic matching alone. This may underscore some limitations of the taint tracking logic implemented in Foxhound that hamper its ability to detect tracking requests (false negatives of taint tracking), or reveal the presence of spurious syntactic matches that artificially inflate the set of tracking requests (false positives of syntactic matching). Perhaps less surprisingly, we also identify 7,021 tracking requests (17%) that are only found by taint tracking. These requests may reveal relevant limitations of syntactic matching that prevent the detection of tracking requests (false negatives of syntactic matching), or be associated with spurious information flows observed by Foxhound (false positives of taint tracking).

4.2 Automated Validation

To estimate the prevalence of false positives and false negatives in the tracking requests exposed by syntactic matching and taint tracking, we developed an automated technique to further scrutinize tracking requests. Specifically, we test whether an identifier is actually read from the client-side storage and sent over the network, i.e., the reported identifier sharing is actual. This technique can be applied to all the tracking requests identified at least by syntactic matching, amounting to the vast majority of our dataset (83%).

4.2.1 Intuition. The core idea of our analysis technique is that, if an identifier was read from the client-side storage and communicated over the network, then any client-side modification of the identifier must be reflected in subsequent network requests. For example, assume that the syntactic matching algorithm finds a correspondence between the cookie `cid` with value `43b89cd7c` and a parameter `uid` with the same value occurring in a network request, thus marking it as a tracking request. If the match is not spurious and we modify the cookie `cid` to the new value `26c45ad9b`, we expect the same network request to be sent with an updated value of the `uid` parameter when the page is visited again.

4.2.2 Implementation. This simple idea is technically challenging to implement in practice for multiple reasons. First of all, it requires matching requests from two different visits. In our simple example, we referred to the *same* network request to clarify the intuition, however this is an abuse of terminology because the second visit will fire new requests. As a matter of fact, finding the correct request to match in the second visit requires some care. Web pages are dynamic in nature, meaning that multiple executions of the same JavaScript code may fire requests that differ, e.g., for the value of HTTP parameters. For this reason, we match requests from different visits by means of a *template* built from the tracking request observed in the first visit. The template of the request includes the origin, the path, and the set of all parameter names. The template also makes explicit where the syntactic matching took place by means of a placeholder. For example, given a request with URL https://tracker.com/track?uid=43b89cd7c where a match involving the `uid` parameter is found, the resulting template is https://tracker.com/track?uid=$ID. We use the template of the tracking request found in the first visit to identify a set of corresponding requests in the second visit by means of regular expression matching over them.

Moreover, our simple example deliberately abstracts from other sources of complexity. In our example, we replaced the original cookie value `43b89cd7c` with the new value `26c45ad9b`. This change preserved the structure of the cookie value, i.e., a hexadecimal string of length 8. Indeed, significant changes are better avoided, because they might break the analyzed web application, e.g., when JavaScript performs some structural checks over the cookie value before communicating it over the network. In our implementation, we rely on *minimal* transformations of identifiers to mitigate this problem. In particular, we select the last alphanumeric character in the identifier and we replace it with the corresponding lexicographic successor in the same character class (upper letters, lower letters, digits). We consider only these types of characters because special characters are often employed as delimiters within storage items, thus replacing them would increase the risk of breaking the parsing of structured identifier values. In our example, the original value of the cookie `cid` `43b89cd7c` would then be modified to `43b89cd7`**`d`**, rather than to the value shown before.

Finally, we observe that in our simple example we communicated the identifier over the network as is. As we explained, this is not always the case in practice, because identifiers can be encoded before being set in the client-side storage and transformed by JavaScript before being communicated over the network. For example, the cookie `cid` with value `43b89cd7c` may be Base64-encoded by JavaScript before being sent over the network. In this case, we would not observe the modified value `43b89cd7d` over the network, but rather its Base64 encoding. This problem is easily solved, because our syntactic matching algorithm attempts multiple transformations of identifiers and tracks those that enabled the syntactic match (see Section 3.3). In our second visit, we must confirm the syntactic match with the modified value using the same sequence of transformations.

Similar but subtler is the case where transformations are not applied before network communication, but take place before the identifier is set in the client-side storage. For example, if `cid` did not just contain the value `43b89cd7c` but its Base64 encoding, we would need to replace the cookie value with the Base64 encoding of `43b89cd7d` to minimize the risk of breakage due to a wrong structure of the cookie. To account for those cases, we try to infer the structure of the cookie before performing the modification of the identifier therein. In particular, we use a brute-forcing approach similar to what we do to identify syntactic matches. Starting from the original value, we perform multiple decoding and parsing attempts with the following transformations: parsing as JSON, Base64 decoding, URL decoding, parsing as query string, splitting by non-alphanumeric characters. Transformations are applied in the reported order and the first successful one is prioritized.

After reconstructing the transformation chain, we modify the decoded identifier and re-encode it again using the same sequence of transformations. In our running example, we would identify that `cid` contains the Base64 encoding of `43b89cd7c`, modify this value to `43b89cd7d`, and re-encode it to Base64; again, the second visit should confirm the syntactic match using the sequence of transformations observed for the original syntactic match.

4.2.3 Final Algorithm. To sum up, we automatically analyze each tracking request identified by syntactic matching to look for spurious matches as follows. First, we identify the client-side storage item (e.g., the cookie `cid`) and the network request component (e.g., the parameter `uid`) that led to the syntactic match, keeping track of the sequence of transformations enabling the match. We then modify the client-side storage item to replace the identifier therein with a new value (the *canary*) as explained above and visit the web page again. We finally use the template of the tracking request to identify a set of *matching requests* R in the second visit. The tracking request can be confirmed as true positive, refuted as false positive, or marked as unconfirmed (i.e., unknown class):

- If R contains a request triggering a syntactic match for the canary using the same sequence of transformations adopted for the original match, the tracking request is confirmed, because there is definite evidence of an information flow

from the client-side storage to the network, i.e., we have a true positive for stateful tracking.
- If $R \neq \emptyset$ and all its requests contain a syntactic match for the original identifier rather than the canary, the tracking request is refuted, because it appears that there is no information flow from the client-side storage to the network, suggesting a spurious match, i.e., we have a false positive for stateful tracking.
- In all the other cases, the tracking request is unconfirmed.

Finally, we perform a post-processing step to improve the reliability of our findings. In particular, if multiple requests share the same template but have been labeled differently using the described approach, we uniform their labels to avoid inconsistencies, because their tracking behavior is expected to be the same. When fixing these inconsistencies, the confirmed label has priority over the refuted label, which in turn subsumes the unconfirmed label.

4.3 False Positive Analysis

Using the presented procedure, we automatically processed all the 33,584 tracking requests exposed by syntactic matching, representing 83% of the total number of tracking requests in our dataset. In total, we managed to identify a non-empty set of matching requests for 31,126 requests (93%). The absence of a matching request in some cases can be explained by the non-deterministic behavior of dynamic web pages, leading to specific requests being sent only under particular conditions, or by our active tampering with the client-side storage, that may lead to breakage and prevent network communication. Nevertheless, our analysis technique has an excellent coverage, because the vast majority of the tracking requests identified by syntactic matching can be automatically scrutinized. We analyze the presence of systematic breakage introduced by our automated testing procedure in Appendix A.

4.3.1 Syntactic Matching. The 31,126 tracking requests that we tested are distributed as follows: 25,440 (82%) are confirmed, 4,858 (16%) are refuted, and 828 (3%) are unconfirmed. This means that 16%–19% of the tracking requests exposed by syntactic matching are likely false positives. This percentage is significant and can have a non-negligible impact on web privacy measurements.

To confirm this finding, we performed additional evaluations. First, we recomputed the class distribution for the subset of 17,496 tracking requests identified *exclusively* by syntactic matching, i.e., requests that were not also detected by taint tracking. These requests are inherently more uncertain, as they lack confirmation from the alternative detection technique. Of these, we were able to automatically assess 15,872 requests (91%), distributed as follows: 11,249 (71%) were confirmed, 4,212 (27%) were refuted, and 411 (3%) remained unconfirmed. This result indicates that tracking requests detected solely via syntactic matching are more prone to false positives. In particular, the estimated false positive rate increases from 16%–19% overall to 27%–30% when considering only those requests not corroborated by taint tracking.

Finally, we complement our automated analysis with a qualitative manual validation of the results. Specifically, we extracted a random sample of ten refuted requests and manually inspected them. Given that these requests may appear across multiple websites due to the reuse of the same tracking script, considering all occurrences of the selected templates results in 1,128 analyzed requests, corresponding to approximately 23% of all refuted requests. Through manual inspection of the requests and direct interaction with the corresponding websites, we confirmed that all refuted requests were false positives. In most cases, they involved the transmission of constant strings that cannot be used for tracking, such as the publisher's domain name or other identifiers bound to the website rather than to the client.

The reason why these constant strings were not filtered out by our heuristics for identifier detection lies in the limitations of the Ratcliff-Obershelp similarity test. Recall that our heuristics rely on this test to discard client-side storage values that do not significantly differ across accesses from different clients, as such values cannot enable tracking. However, this method is not robust in all cases. The constant strings we identified were often embedded as substrings within more complex client-side storage values (e.g., of the form `<random_value>-<constant_value>`). Consequently, the similarity test detects differences caused by the variable portion of the value, while the constant substring (later transmitted over the network) remains undetected, leading to false positives.

4.3.2 Taint Tracking. We first observe that 16,088 out of the 23,109 tracking requests exposed by taint tracking (70%) are identified by syntactic matching as well, meaning that they can be effectively tested using the automated technique presented in Section 4.2. In particular, our tests covered 15,254 requests (95%) detected through taint tracking, which are distributed as follows: 14,191 (93%) are confirmed, 646 (4%) are refuted, and 417 (3%) are unconfirmed. For this set of tracking requests, the estimated amount of false positives is then 4%–7%, which is significantly lower than what we estimated for syntactic matching in general (16%–19%).

Automatically analyzing the other 7,021 tracking requests exposed by taint tracking for the presence of false positives is arguably more challenging than processing the tracking requests detected by syntactic matching, because taint tracking propagates taints across all the JavaScript operations, meaning that the communicated identifier may be transformed in unanticipated ways. This means that the canary-based approach we put forward for syntactic matching does not readily generalize to taint tracking.

For these requests, we performed an additional analysis to heuristically estimate the presence of false positives. In particular, we estimated the number of tracking requests supported by an information flow that reads at least eight consecutive characters from an identifier and transmits a sequence of consecutive characters over the network that (*i*) has at least the same length, (*ii*) carries the same taint labels, and (*iii*) occurs within a single request parameter. This simple heuristic captures a number of scenarios that take as input an identifier, transform it in some way, and communicate it over the network in a format that the server can readily process.

As it turns out, 7,003 out of 7,021 tracking requests respect these conditions and are likely true positives. To get additional assurance about this claim, we sampled ten requests that comply with our heuristics and manually inspected them. Again, these requests may be repeated on different websites due to the presence of the same tracking script. If we count the different occurrences of the selected requests, our analysis covered 6,321 requests with the same templates, which is more than 90% of the total cases. By hooking

appropriate breakpoints in the scripts sending the detected tracking requests, we were able to manually confirm all these cases. Moreover, we independently confirmed the 18 not satisfying the conditions described above as true positives. This further demonstrates the precision of dynamic taint tracking.

4.4 False Negative Analysis

We here explain how we use the collected data to estimate the prevalence of false negatives for syntactic matching and taint tracking. Of course, estimating false negatives is harder than estimating false positives, because there may be tracking requests that are missed by both syntactic matching and taint tracking. Still, we can look into the tracking requests detected by just one of the two approaches to shed light on the limitations of the other.

4.4.1 Syntactic Matching. In total, our dataset includes 7,021 tracking requests that are exposed by taint tracking alone, meaning that syntactic matching would not detect around 17% of the requests in the entire dataset. Recall that we estimated the vast majority of these cases to be true positives, hence they are false negatives for syntactic matching. To better understand this from a qualitative perspective, we analyzed in more detail the manually confirmed true positives from the set of requests identified by taint tracking alone. As it turns out, 6,111 of these requests (87%) are sent to domains operated by Google Analytics.

The reason why these tracking requests are missed by syntactic matching is readily explained. Google Analytics sets a tracking cookie with a value of the form `GAX.Y.Z.C`, where `Z` and `C` are unique random numbers. The network requests to the tracking endpoints include a parameter `cid` with value `Z.C`, i.e., the value of the tracking cookie without the `GAX` and `Y` components, which suffices to track the client. When syntactic matching computes the transformations of the cookie value, it also splits `GAX.Y.Z.C` over the dots to produce `GAX`, `Y`, `Z`, and `C` separately. Unfortunately, `Z` and `C` are not always detected as identifiers by themselves, meaning that syntactic matching will not look for their occurrence in the network request. This problem does not affect taint tracking, which correctly propagates the taints associated with `Z` and `C` to the network sink; since the concatenation of `Z` and `C` is most often detected as an identifier, because the guessability of `Z + C` is lower than that of `Z` or `C` in isolation, the network request is marked as tracking.

Besides the Google Analytics case, we also manually confirmed examples where identifiers reached network requests after being transformed in subtle ways that syntactic matching cannot anticipate. For example, the tracker featuregates.org receives the identifier after it is Base64-encoded and the resulting characters are reversed, while pagead2.googlesyndication.com receives the identifier after its letters are converted to uppercase. These examples require manual investigation to find out and we do not perform an extensive analysis of all the observed behavior. Still, this gives preliminary concrete evidence that syntactic matching is inherently limited, because it is impossible to accommodate all possible transformations that trackers may put in place.

4.4.2 Taint Tracking. In total, our dataset includes 17,496 tracking requests that are detected by syntactic matching alone. The estimated number of true positives among these requests is between 11,249 and 11,652, representing 28%–29% of the entire dataset. This means that taint tracking may miss around one third of the tracking requests. To better understand why, we randomly sampled a few requests that have been detected by syntactic matching alone and confirmed as true positives by our automated validation technique. As it turns out, taint tracking may fail in practice for multiple reasons. First, we have implementation issues and limitations that lead to losing taints when they should be propagated. For example, we observed that Foxhound only propagates taint across string operations, meaning that when identifiers are transformed to other data types, their taint is lost. This occurred for example in all cases involving the tracker mc.yandex.com, in which the numeric string read from storage (tainted) is converted to a number and then back to an untainted string before reaching the network sink.

As another example, we observed complexities of the JavaScript semantics that led to losing taints even when operating with strings. For example, we observed cases with region1.google-analytics.com where a string read from storage (tainted) is used as a key in a JavaScript object, and reading it back via the `Object.keys` JavaScript method produces an untainted string.

Finally, we observed a conceptual limitation of Foxhound that limits its ability to observe web tracking behavior. Since Foxhound operates at the JavaScript level, it cannot detect tracking practices performed through redirects. For example, a tracker called sb.scorecardresearch.com receives an identifier read from storage (tainted) as a request parameter, which is then reflected in a redirect response and included as a parameter in the subsequent request to another tracker, end.mpod.ch. While the former request is promptly detected by both techniques, the latter is not, because Javascript-based taint tracking has no visibility of HTTP redirects.

5 Implications for Web Privacy Analyses

Our previous investigation focused on tracking requests and performed a comparative analysis of two different approaches to detect them based on a common dataset. We here extend our analysis to understand to which extent syntactic matching and taint tracking are effective at characterizing the web privacy ecosystem in terms of relevant research inferences.

5.1 Measuring Web Tracking

Web privacy research does not just quantify tracking requests, but uses different measures to characterize the web privacy ecosystem. Relevant measurable aspects covered in prior work include:

(1) *Number of trackers* [16]: how many distinct trackers can be exposed in the wild and what is the average number of trackers per website?
(2) *Prevalence of web tracking* [16]: how many different websites send some tracking requests and what are the most popular trackers observed in the wild?
(3) *Effectiveness of filter lists* [18]: how many tracking requests can be blocked by popular filter lists like Disconnect?

Our goal is determining how much the choice of specific approaches to the detection of tracking requests impacts on the high-level privacy inferences supported by the measures reported above. In particular, we use the set of tracking requests identified by syntactic matching and taint tracking as available in our dataset, including

Table 4: Comparing different approaches to web privacy measurements. S = Syntactic, S_{NR} = SyntacticNR (No Refuted), S_C = SyntacticC (Confirmed), T = Taint.

Measure	S	S_{NR}	S_C	T	$S_C \cup T$
Total number of tracking req.	33,584	28,726	25,440	23,109	34,358
└ In Disconnect	25,975	22,357	19,909	19,020	28,449
Average number of tracking req.	4.41	3.77	3.34	3.04	4.51
└ In Disconnect	3.41	2.94	2.61	2.50	3.74
Total number of trackers	2,823	2,296	1,813	1,623	2,183
└ In Disconnect	1,255	1,010	760	728	988
Average number of trackers	2.02	1.73	1.52	1.47	2.06
└ In Disconnect	1.51	1.31	1.16	1.17	1.67
Number of sites with a tracker	3,642	3,538	3,396	4,292	4,604
└ In Disconnect	2,980	2,902	2,756	3,970	4,254

possible variants and combinations of the two approaches, to determine how the analysis of the web tracking ecosystem would change in terms of measurable research inferences.

We clarify that in the following we use the term *tracker* to refer to any domain that is the target of a tracking request according to the data collected using a given measurement technique. Not all such trackers are necessarily associated with actual web tracking practices, they are just exposed by some measurement technique.

5.2 Variants of Syntactic Matching

We compare three flavors of syntactic matching: the classic approach presented in Section 3.3 and two variants based on the automated validation technique presented in Section 4.2. The first variant (SyntacticNR) filters out the refuted requests, i.e., those that have been marked as false positives by our validation procedure; the second variant (SyntacticC) instead only keeps the confirmed requests, i.e., those have been marked as true positives. The difference in the two variants lies in their treatment of unconfirmed requests that have not been classified as either true positives or false positives: these requests are considered actual tracking requests by SyntacticNR, while they are conservatively filtered out by SyntacticC. In terms of findings, the three approaches are increasingly more refined. The classic syntactic matching approach is coarse-grained, but based on standard research practices. It suffers from many false positives due to spurious matches, which are removed in SyntacticNR. SyntacticNR, in turn, is less conservative than SyntacticC, because it assumes that unconfirmed tracking requests may be used for tracking.

Table 4 reports different quantitative measures established using the three techniques under study. As we can see, the choice of a specific syntactic matching technique has a significant impact on some of the reported measures. For example, syntactic matching identified 33,584 tracking requests, while SyntacticNR identified just 28,726 tracking requests (-14%) and SyntacticC detected only 25,440 tracking requests (-24%). Similarly, the total number of distinct trackers ranges from 1,813 for SyntacticC to 2,823 (+56%) for classic syntactic matching.

Since we do not have a ground truth for web tracking, we cannot conclusively determine which of the three techniques yields the most accurate results in practice. We tried to make an educated guess by building on top of the Disconnect list for web tracking protection. Although Disconnect is far from a comprehensive list of all the trackers on the web [14], it is fair to assume that most of the domains included therein are associated with actual web trackers. We can then use the prevalence of Disconnect domains within our dataset of tracking requests as a proxy of the ability of a detection technique to expose relevant tracking behavior. What we observe is that, if we recompute our quantitative measures after matching tracking requests against the Disconnect list, the absolute numbers drop as expected, but there is no clear signal that can help us identify the most accurate technique. For example, the percentage of tracking requests in the Disconnect list ranges from 77% to 78% based on the chosen syntactic matching technique, while the percentage of identified trackers occurring in the Disconnect list ranges from 42% to 44%. These differences are too small to conclude anything meaningful about specific detection techniques.

It is instructive to look into which trackers are exposed by which technique. A first interesting observation we make is that the top ten trackers identified by syntactic matching, SyntacticNR and SyntacticC are very similar (see Table 6). Also their popularity is quite stable, with the notable exception of pagead2.googlesyndication.com: this tracker is detected in 360 websites using the classic syntactic matching approach, while its popularity drops to 158 for SyntacticNR and even lower for SyntacticC. The reason is that this domain uses multiple endpoints: in particular, the `/ccm/collect` endpoint does not appear associated with stateful tracking practices, because the detected match for the corresponding tracking requests is for the domain name of the publisher website. These requests are marked as tracking by syntactic matching, but they are correctly filtered out by using SyntacticNR and SyntacticC. Other endpoints of pagead2.googlesyndication.com, like `/pagead/ads`, instead appear to be used for tracking, hence requests sent to these endpoints are not removed by our improved heuristics.

In terms of trackers that are filtered out by the use of different approaches, we observe that the transition from classic syntactic matching to SyntacticNR removes 527 trackers (245 in Disconnect), with the most popular ones being tags.creativecdn.com (73 websites), bat.bing.net (66 websites), api.livechatinc.com (43 websites), secure.livechatinc.com (41 websites) and analytics.twitter.com (40 websites). Except for bat.bing.net, all these domains are included in the Disconnect list. We manually inspected the following cases and confirmed that tracking requests sent to them are false positives for the described motivations.

tags.creativecdn.com: The match is found in a path segment, which includes a domain-specific constant string instead of a variable client identifier.

bat.bing.net: The match is found over the `p` query parameter, which includes the domain name of the publisher website.

api.livechatinc.com / secure.livechatinc.com: The match is found over the `organization_id` query parameter, which includes a domain-specific constant string.

analytics.twitter.com: Two matches are found over different query parameters, including respectively the domain name and the landing page title associated with the publisher website.

This shows that the use of SyntacticNR over the classic syntactic matching approach is effective at removing false positives that have relevant effects in terms of important privacy inferences.

We observe a similar picture for SyntacticC. The adoption of SyntacticC over classic syntactic matching removes 1,010 trackers (495 in Disconnect), including www.googletagmanager.com (153 websites), fundingchoicesmessages.google.com (98 websites), connect.facebook.net (85 websites) and tags.creativecdn.com (73 websites). With the exception of www.googletagmanager.com, all of these are in Disconnect. We already discussed that the tracking requests sent to tags.creativecdn.com are false positives, so we manually inspected the remaining cases and concluded the corresponding requests to be false positives as well. Specifically, for www.googletagmanager.com, two types of matches are found over different query parameters, each including a domain-specific constant string and the domain name of the publisher website. Regarding fundingchoicesmessages.google.com and connect.facebook.net, the match is found over the `fccs` and `domain` query parameters, respectively, which include the domain name of the publisher.

Note that SyntacticC also removes the false positives removed by SyntacticNR that we discussed above, because it performs a more aggressive filtering. As such, we showed that both the additional refinements are helpful to avoid over-reporting.

At the end of the day, refining syntactic matching with additional heuristics is definitely useful. When measuring top trackers alone, all three syntactic matching variants are roughly equivalent, however relevant differences take place when moving to other low-popularity trackers included in Disconnect. For these cases, we observe that syntactic matching shows a clear trend towards over-reporting, which can be significantly mitigated by introducing additional refinements like SyntacticNR and SyntacticC.

5.3 Syntactic Matching vs. Taint Tracking

We now compare syntactic matching and taint tracking (see again Table 4). Taint tracking identified 23,109 tracking requests, which is a -31% reduction with respect to syntactic matching (-20% w.r.t. SyntacticNR, -9% w.r.t. SyntacticC). An interesting observation is that 82% of these requests are sent to actual trackers included in Disconnect, which is higher than the percentage estimated for syntactic matching (at most 78%). This may be associated with the limited number of false positives observed in our experimental evaluation. In terms of total trackers, taint tracking exposed 1,623 distinct trackers, which is a -43% reduction with respect to syntactic matching (-29% w.r.t. SyntacticNR, -10% w.r.t. SyntacticC). The reduction also affected many domains in the Disconnect list: out of the 1,200 removed trackers, 527 occur in Disconnect. The percentage of detected trackers in the Disconnect list is around 45% for taint tracking, which is slightly above what we estimated for syntactic matching (42% - 44%). Collectively, these numbers confirm that taint tracking is more precise than syntactic matching.

Moreover, the analysis of trackers that have been detected only by one between syntactic matching and taint tracking reveals important insights. If we focus on the trackers that have been detected just by taint tracking, we observe that all of them have low popularity, because they are included in two websites at most. Examples include subdomains of fls.doubleclick.net and evergage.com, both of which are included in Disconnect and are associated with web tracking practices. This shows that taint tracking can generalize to trackers that are less widespread in the wild and escape syntactic matching. As for the trackers that have been detected only by syntactic matching, we instead observe a few prominent entries from Disconnect like accounts.google.com (226 websites), fundingchoicesmessages.google.com (82 websites) and connect.facebook.net (85 websites). After manual investigation, we observed that all these cases are not associated with web tracking practices. This confirms that syntactic matching alone suffers from too many false positives.

The picture is more interesting if we compare the trackers that have been detected only by either SyntacticC or taint tracking, because we showed that SyntacticC has the lowest number of false positives among syntactic matching variants. Both sets of trackers include relevant entries corresponding to actual web trackers that are missed by one of the two approaches. Trackers that are only detected by taint tracking include a.mrktmtrcs.net (34 websites), t.dtscout.com (34 websites) and gaua.hit.gemius.pl (17 websites), all of which are present in Disconnect. For these cases, SyntacticC cannot confirm the tracking request due to the absence of a matching request in the second visit to the website. Instead, SyntacticC can detect the presence of trackers like rtb.openx.net (71 websites), yandex.ru (69 websites) and widget.as.criteo.com (40 websites).

Finally, Table 6 also reports the most popular trackers identified by taint tracking. Although the key market players are largely the same observed for all variants of syntactic matching, we notice that tracking requests to region1.google-analytics.com are detected on 1,923 websites, which is almost four times the prevalence estimated using syntactic matching (511 websites). In general, taint tracking appears more effective than syntactic matching at detecting tracking requests sent to different domains operated by Google Analytics, as we discussed in Section 4.4. This is reflected by the number of websites with a tracker, which increases from 3,642 for syntactic matching to 4,292 (+18%) for taint tracking.

5.4 Combining Techniques

In our analysis, we showed that both SyntacticC and taint tracking are precise, producing significantly less false positives than traditional syntactic matching. However, we also established that they suffer from a number of false negatives. To compensate for that, we experiment with a way to combine SyntacticC and taint tracking by considering the union of their findings. This way, we combine results from two independent techniques with high precision in the attempt to compensate for their shortcomings.

As it turns out, the union of the results lead to the highest number of detected tracking requests: 34,358, which is a +49% increase over taint tracking and a +35% increase over SyntacticC. Remarkably, 28,449 of these requests are sent to domains included in Disconnect (83%). This percentage is higher than the 82% computed for taint tracking and the 78% computed for SyntacticC. The total number of identified trackers is 2,183, which is a +35% increase over taint tracking and a +20% increase over SyntacticC. The percentage of identified trackers included in the Disconnect list is 45%, which is in line with taint tracking and slightly higher than SyntacticC. The number of websites where we observe the presence of trackers is 4,604, which is a +7% increase over taint tracking and a +36% increase over SyntacticC. The significant difference in increase for the two cases can be explained by the improved visibility of taint tracking into the tracking behavior of Google Analytics, which is

Table 5: Effectiveness of the Disconnect list against the tracking requests detected by SyntacticC (S_C) and taint tracking (T). −Disconnect denotes the numbers computed after filtering out the requests sent to domains included in Disconnect.

Measure	S_C	−Disconnect	T	−Disconnect
Total number of tracking req.	25,440	5,531	23,109	4,089
Average number of tracking req.	3.34	0.73	3.04	0.54
Total number of trackers	1,813	1,053	1,623	895
Average number of trackers	1.52	0.37	1.47	0.29
Number of sites with a tracker	3,396	1,659	4,292	1,469

very popular in the wild and features many domains included in the Disconnect list.

Given that both SyntacticC and taint tracking are precise, the observed increase in numbers when using the union of their results confirms the prevalence of false negatives for the two approaches and suggests the need for a combination of multiple detection techniques in robust web privacy measurements.

5.5 Effectiveness of Filter Lists

Finally, we estimate the effectiveness of the Disconnect filter list based on the tracking requests collected by SyntacticC and taint tracking. The effectiveness of filter lists is an important research topic, which was primarily tackled through empirical studies [18]. Here we estimate the privacy improvements supported by Disconnect by recomputing our quantitative measures in presence of a filter that rules out all the tracking requests sent to a domain in Disconnect. Table 5 reports the numbers. The reduction in the number of tracking requests is estimated as -78% for SyntacticC and -82% for taint tracking, which are roughly comparable estimates. As for the number of websites embedding a tracker, we estimate a reduction of -51% for SyntacticC and -66% for taint tracking. Here, the more significant difference for taint tracking can be easily explained by the inclusion in Disconnect of several domains operated by Google Analytics, a tracker whose popularity is more accurately estimated by taint tracking, as we explained.

To further investigate the impact of the Disconnect filter list, we report in Table 7 the top trackers observed for syntactic matching and taint tracking after filtering out the domains occurring in Disconnect. The table highlights significant similarities in terms of the remaining detected trackers, with the first four positions being occupied by the same trackers, and three additional common elements in the two sets. However, our analysis also shows important differences between the residual tracking potential observed when using either SyntacticC or taint tracking in our web measurement. In particular, we can compute the set of trackers not included in Disconnect that are detected using just one of the two measurement techniques. We observe that Disconnect does not block communication with several popular trackers detected by SyntacticC alone, including b30612d89.a910d264c4.com (22 websites), bid.sparteo.com (20 websites) and log.cookieyes.com (14 websites). All these cases are interesting: b30612d89.a910d264c4.com has a random name and is likely associated with web tracking practices; bid.sparteo.com appears to be owned by a French tracker not included in Disconnect; and log.cookieyes.com appears related to a consent management platform. In turn, taint tracking alone reveals the existence of relevant trackers not included in Disconnect, including go.skmada.com (8 websites), smetrics.rcsmetrics.it (5 websites) and events.getsitectrl.com (4 websites). Specifically, go.skmada.com appears to be linked to a minor tracking infrastructure; smetrics.rcsmetrics.it performs ad-related tracking on websites belonging to an Italian publishing group; and events.getsitectrl.com likely performs tracking for email marketing.

6 Discussion

We here summarize the key insights of our study and discuss the main limitations of our investigation.

6.1 Recommendation and Take-Aways

Recent work on web measurements shows that they are potentially fragile, because specific and often under-specified implementation details may have a significant impact on the established research inferences [11, 12, 20]. Our work further reinforces this claim. Measuring web tracking by means of classic syntactic matching heuristics is subject to both false positives and false negatives. In particular, we estimated the amount of false positives to be around 16% - 19% for the tracking requests exposed by syntactic matching. These false positives are not just measurement artifacts, but negatively affect the reliability of important privacy inferences, e.g., by revealing a wide prevalence of trackers which are not actually performing any form of stateful tracking.

Luckily, we can use principled approaches to significantly improve over the reported shortcomings. We proposed effective testing techniques to corroborate the findings of syntactic matching, leading to new heuristics which are more precise because they explicitly confirm the existence of communication patterns associated with web tracking practices. Our most conservative heuristic, called SyntacticC, is effective at ruling out false positives. As a reference, SyntacticC identified a -24% reduction in the number of tracking requests and a -36% reduction in the number of distinct trackers with respect to classic syntactic matching. A careful analysis of the filtered requests confirmed that they are not associated with stateful web tracking.

That said, SyntacticC inherits all the key limitations of syntactic matching: since it has no visibility of the JavaScript logic, it can easily miss tracking requests and suffers from a significant number of false negatives. Taint tracking is a more principled alternative than syntactic matching, because it performs a dynamic analysis of JavaScript to draw more precise conclusions. Still, a state-of-the-art implementation of browser-side taint tracking like Foxhound still suffers from a high number of false negatives. The reasons for this are manifold, including the restriction of its taint propagation logic to string operations, the incomplete support for all the complexity of the JavaScript semantics (e.g., storing an identifier as a property key and subsequently reading it back with `Object.keys`), and the adoption of tracking practices which are oblivious to JavaScript (e.g., HTTP redirects). As such, taint tracking is not (yet) ready to fully replace syntactic matching and we advocate for a combination of multiple detection approaches to mitigate on their independently detected shortcomings. Extending Foxhound to support taint propagation across additional data types would in principle be feasible, but it would require substantial modifications to the taint tracking

engine, in the order of thousands of lines of code. These changes may have a major impact on the performance of taint tracking and the precision of the analysis. Our study shows that the current focus of Foxhound on strings alone is insufficient for effective web tracking detection, and researchers who are willing to use Foxhound for privacy analyses should be aware of this limitation. A major overhaul of Foxhound goes beyond the scope of our study.

As for the role of filter lists, we observe that a quantitative evaluation of the effectiveness of the Disconnect list yields similar results when using SyntacticC and taint tracking. Still, the loopholes identified in Disconnect are different when using these two techniques, which further confirms the benefits of using multiple measurement tools. Also, we observe that the blind adoption of the filter lists may lead to over-reporting the prevalence of stateful tracking, because many requests sent to domains included in these lists do not contain any client identifier or are not associated with any stateful tracking practice as reported in previous work [18, 38].

6.2 Limitations

A limitation of our study is that it focuses on specific implementations of syntactic matching and taint tracking. Nevertheless, we carefully designed our experiments to ensure their representativeness and generality. We implemented our syntactic matching heuristics based on previous web privacy measurements presented at reputable conferences. We tried to capture the essence of syntactic matching by supporting multiple identifier transformations and techniques for request parsing, essentially building a collection of sound ideas already presented in the literature. As for taint tracking, we reused a state-of-the-art tool like Foxhound and we implemented just the necessary changes required to enable a fair comparison with syntactic matching, like byte-level taint tracking, to be able to identify which character of a specific cookie or local storage item was read by JavaScript. Foxhound is just one specific implementation of a taint tracker, but it is also the only reliable one for JavaScript according to a recent study [7] and has already been used in prior web privacy measurements [3, 6]. We believe that several of our findings are generalizable: no matter how syntactic matching is implemented, it will always suffer from false positives and false negatives given its lack of visibility of the JavaScript code. Our work quantifies the prevalence of false positives and false negatives based on a reasonable implementation built by taking into account the state of the art, and proposes possible ways forward.

We also acknowledge that our work does not cover all possible ways to measure web tracking in the wild. In particular, our analysis does not cover machine learning-based approaches to web tracking detection, like AdGraph [22] and CookieGraph [29], or other behavior-based detection methods [24, 36, 43, 46]. Extending our comparison to this additional class of tracking detection methods would be an interesting avenue for future work. In this study, we privileged syntactic matching and taint tracking for multiple reasons. Notably, they are widely adopted techniques and have been used in many web measurement studies, including recent ones [3, 4, 6, 34]. Moreover, syntactic matching and taint tracking are interesting in their own rights as two representative baselines, with the former completely lacking visibility of JavaScript and the latter having access to the entire JavaScript logic. This contrast is particularly meaningful, as the two approaches operate at essentially opposite ends of the observability spectrum.

7 Related Work

Stateful tracking has been extensively studied in prior work, leading to the development of multiple detection techniques. In this paper, we present a comparative analysis of dynamic taint tracking and syntactic matching, two approaches that are widely adopted in web privacy research. To the best of our knowledge, previous studies have evaluated these techniques largely in isolation, without providing a systematic comparison on their effectiveness.

Previous work employed dynamic taint tracking to quantify privacy-violating information flows at scale [23, 39], uncover information leakage enabled by browser extensions [10, 44, 45], analyze tracking practices involving cookies and web storage [3, 9], and perform comparative privacy analyses [2].

Conversely, prior applications of syntactic matching heuristics include studies on third-party tracking detection [32], tracking persistence [1], large-scale tracking measurements [16], extension-enabled privacy leakage [40], cookie syncing [30], invisible pixel-based tracking [18], characterization of the cookie ecosystem [34], and first-party cookie jar sandboxing [4].

A third relevant category of techniques is based on machine learning. These approaches train classifiers to detect tracking at different abstraction layers: request-level features such as URLs and network metadata [5, 19], JavaScript structural and behavioral features [21, 24, 43], and graph-based representations of page execution and resource interactions [22, 29, 36, 46]. Including these approaches in the comparison would be an interesting subject for future work.

Last, the traditional method for blocking ads and trackers are filter lists, of which Disconnect [13] and EasyList / EasyPrivacy [15] are popular examples. Since filter lists serve as the primary data source for popular ad blockers, web privacy research commonly uses them as a baseline when evaluating novel detection techniques. In this work, we leverage Disconnect [13] to assess the privacy implications of using a certain detection technique.

8 Conclusion

Syntactic matching and taint tracking are two widely adopted techniques to measure the prevalence of web tracking in the wild. In this work, we examined these approaches as two extremes in terms of the JavaScript observability spectrum, ranging from complete opacity to full visibility into runtime information flows. Our analysis shows that neither technique is suitable to be deployed out of the box without additional validation and controls, as each exhibits relevant blind spots that affect detection accuracy. At the same time, their complementary strengths suggest that combining syntactic matching with taint tracking, when done with care, can provide a more robust and nuanced assessment of web tracking practices.

As future work, we would like to extend our evaluation to machine learning-based approaches to web tracking detection [22, 29, 36, 46]. Moreover, we would like to generalize our findings to additional filter lists besides Disconnect and to further contribute to the implementation of the Foxhound project to improve its amenability as a robust measurement tool for web tracking practices.

Acknowledgments

This research was supported by the Vienna Science and Technology Fund (WWTF) and the City of Vienna [Grant ID: 10.47379/ICT22060], Cyber Security Austria (CSA), and project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU. The project benefited from the high-performance computing infrastructure developed under the project CONVECS, funded by the PR Veneto FESR 2021-2027 program, Priority 1 – Specific Objective 1.1 – Action 1.1.2.

The authors used generative AI-based tools to revise the text, improve flow and correct any typos, grammatical errors, and awkward phrasing. Moreover, they declare that these tools were not used to generate any content in this manuscript.

References

[1] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Díaz. 2014. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 674–689. https://doi.org/10.1145/2660267.2660347

[2] Zubair Ahmad, Stefano Calzavara, Samuele Casarin, and Ben Stock. 2024. Information flow control for comparative privacy analyses. *Int. J. Inf. Sec.* 23, 5 (2024), 3199–3216. https://doi.org/10.1007/S10207-024-00886-0

[3] Zubair Ahmad, Samuele Casarin, and Stefano Calzavara. 2024. An Empirical Analysis of Web Storage and Its Applications to Web Tracking. *ACM Trans. Web* 18, 1 (2024), 7:1–7:28. https://doi.org/10.1145/3623382

[4] Pouneh Nikkhah Bahrami, Aurore Fass, and Zubair Shafiq. 2025. CookieGuard: Characterizing and Isolating the First-Party Cookie Jar. In *Proceedings of the 2025 ACM Internet Measurement Conference, IMC 2025, Madison, WI, USA, 28-31 October 2025*. ACM, 645–661. https://doi.org/10.1145/3730567.3764490

[5] Sruti Bhagavatula, Christopher W. Dunn, Chris Kanich, Minaxi Gupta, and Brian D. Ziebart. 2014. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop, AISec 2014, Scottsdale, AZ, USA, November 7, 2014*. ACM, 95–102. https://doi.org/10.1145/2666652.2666662

[6] Soumaya Boussaha, Lukas Hock, Miguel Bermejo, Rubén Cuevas Rumín, Ángel Cuevas Rumín, David Klein, Martin Johns, Luca Compagna, Daniele Antonioli, and Thomas Barber. 2024. FP-tracer: Fine-grained Browser Fingerprinting Detection via Taint-tracking and Entropy-based Thresholds. *Proc. Priv. Enhancing Technol.* 2024, 3 (2024), 540–560. https://doi.org/10.56553/POPETS-2024-0092

[7] Stefano Calzavara, Samuele Casarin, and Riccardo Focardi. 2025. Dynamic Security Analysis of JavaScript: Are We There Yet?. In *Proceedings of the ACM on Web Conference 2025, WWW 2025, Sydney, NSW, Australia, 28 April 2025- 2 May 2025*. ACM, 1105–1115. https://doi.org/10.1145/3696410.3714614

[8] Stefano Calzavara, Samuele Casarin, Marco Squarcina, and Matteo Maffei. 2026. Repository containing all of the artifacts related to this paper. http://purl.org/tracking-detection-paper.

[9] Quan Chen, Panagiotis Ilia, Michalis Polychronakis, and Alexandros Kapravelos. 2021. Cookie Swap Party: Abusing First-Party Cookies for Web Tracking. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*. ACM / IW3C2, 2117–2129. https://doi.org/10.1145/3442381.3449837

[10] Quan Chen and Alexandros Kapravelos. 2018. Mystique: Uncovering Information Leakage from Browser Extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 1687–1700. https://doi.org/10.1145/3243734.3243823

[11] Nurullah Demir, Matteo Große-Kampmann, Tobias Urban, Christian Wressnegger, Thorsten Holz, and Norbert Pohlmann. 2022. Reproducibility and Replicability of Web Measurement Studies. In *WWW '22: The ACM Web Conference 2022, Virtual Event, Lyon, France, April 25 - 29, 2022*. ACM, 533–544. https://doi.org/10.1145/3485447.3512214

[12] Nurullah Demir, Jan Hörnemann, Matteo Große-Kampmann, Tobias Urban, Norbert Pohlmann, Thorsten Holz, and Christian Wressnegger. 2023. On the Similarity of Web Measurements Under Different Experimental Setups. In *Proceedings of the 2023 ACM on Internet Measurement Conference, IMC 2023, Montreal, QC, Canada, October 24-26, 2023*. ACM, 356–369. https://doi.org/10.1145/3618257.3624795

[13] Disconnect. 2026. Disconnect - Tracker Protection lists. https://disconnect.me/trackerprotection. Accessed: 2026-02-27.

[14] Disconnect. 2026. Tracker Descriptions for Selected Fingerprinters and Cryptominers. https://github.com/disconnectme/disconnect-tracking-protection/blob/master/descriptions.md. Accessed: 2026-02-27.

[15] EasyList. 2026. EasyList - Overview. https://easylist.to/. Accessed: 2026-02-27.

[16] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 1388–1401. https://doi.org/10.1145/2976749.2978313

[17] Steven Englehardt, Dillon Reisman, Christian Eubank, Peter Zimmerman, Jonathan R. Mayer, Arvind Narayanan, and Edward W. Felten. 2015. Cookies That Give You Away: The Surveillance Implications of Web Tracking. In *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*. ACM, 289–299. https://doi.org/10.1145/2736277.2741679

[18] Imane Fouad, Nataliia Bielova, Arnaud Legout, and Natasa Sarafijanovic-Djukic. 2020. Missed by Filter Lists: Detecting Unknown Third-Party Trackers with Invisible Pixels. *Proc. Priv. Enhancing Technol.* 2020, 2 (2020), 499–518. https://doi.org/10.2478/POPETS-2020-0038

[19] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. 2015. An Automated Approach for Complementing Ad Blockers' Blacklists. *Proc. Priv. Enhancing Technol.* 2015, 2 (2015), 282–298. https://doi.org/10.1515/POPETS-2015-0018

[20] Florian Hantke, Peter Snyder, Hamed Haddadi, and Ben Stock. 2025. Web Execution Bundles: Reproducible, Accurate, and Archivable Web Measurements. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*. USENIX Association, 8235–8253. https://www.usenix.org/conference/usenixsecurity25/presentation/hantke

[21] Muhammad Ikram, Hassan Jameel Asghar, Mohamed Ali Kâafar, Anirban Mahanti, and Balachander Krishnamurthy. 2017. Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning. *Proc. Priv. Enhancing Technol.* 2017, 1 (2017), 79–99. https://doi.org/10.1515/POPETS-2017-0006

[22] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2020. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 763–776. https://doi.org/10.1109/SP40000.2020.00005

[23] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2010. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*. ACM, 270–283. https://doi.org/10.1145/1866307.1866339

[24] Andrew J. Kaizer and Minaxi Gupta. 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@CODASPY 2016, New Orleans, LA, USA, March 11, 2016*. ACM, 33–40. https://doi.org/10.1145/2875475.2875479

[25] Soheil Khodayari, Thomas Barber, and Giancarlo Pellegrino. 2024. The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*. IEEE, 166–184. https://doi.org/10.1109/SP54263.2024.00098

[26] David Klein, Thomas Barber, Souphiane Bensalim, Ben Stock, and Martin Johns. 2022. Hand Sanitizers in the Wild: A Large-scale Study of Custom JavaScript Sanitizer Functions. In *7th IEEE European Symposium on Security and Privacy, EuroS&P 2022, Genoa, Italy, June 6-10, 2022*. IEEE, 236–250. https://doi.org/10.1109/EUROSP53844.2022.00023

[27] Tianyi Li, Xiaofeng Zheng, Kaiwen Shen, and Xinhui Han. 2021. FPFlow: Detect and Prevent Browser Fingerprinting with Dynamic Taint Analysis. In *Cyber Security - 18th China Annual Conference, CNCERT 2021, Beijing, China, July 20-21, 2021, Revised Selected Papers (Communications in Computer and Information Science, Vol. 1506)*. Springer, 51–67. https://doi.org/10.1007/978-981-16-9229-1_4

[28] Jonathan R. Mayer and John C. Mitchell. 2012. Third-Party Web Tracking: Policy and Technology. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*. IEEE Computer Society, 413–427. https://doi.org/10.1109/SP.2012.47

[29] Shaoor Munir, Sandra Deepthy Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. 2023. CookieGraph: Understanding and Detecting First-Party Tracking Cookies. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*. ACM, 3490–3504. https://doi.org/10.1145/3576915.3616586

[30] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos P. Markatos. 2019. Cookie Synchronization: Everything You Always Wanted to Know But Were Afraid to Ask. In *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*. ACM, 1432–1442. https://doi.org/10.1145/3308558.3313542

[31] Victor Le Pochat, Tom van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/tranco-a-research-oriented-top-sites-ranking-hardened-against-manipulation/

[32] Franziska Roesner, Tadayoshi Kohno, and David Wetherall. 2012. Detecting and Defending Against Third-Party Tracking on the Web. In *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI*

2012, San Jose, CA, USA, April 25-27, 2012. USENIX Association, 155–168. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/roesner

[33] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19. https://doi.org/10.1109/JSAC.2002.806121

[34] Iskander Sánchez-Rola, Matteo Dell'Amico, Davide Balzarotti, Pierre-Antoine Vervier, and Leyla Bilge. 2022. Journey to the Center of the Cookie Ecosystem: Unraveling Actors' Roles and Relationships. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1990–2004. https://doi.org/10.1109/SP40001.2021.9796062

[35] Iskander Sánchez-Rola, Matteo Dell'Amico, Platon Kotzias, Davide Balzarotti, Leyla Bilge, Pierre-Antoine Vervier, and Igor Santos. 2019. Can I Opt Out Yet?: GDPR and the Global Illusion of Cookie Control. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*. ACM, 340–351. https://doi.org/10.1145/3321705.3329806

[36] Sandra Deepthy Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. 2022. WebGraph: Capturing Advertising and Tracking Information Flows for Robust Blocking. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*. USENIX Association, 2875–2892. https://www.usenix.org/conference/usenixsecurity22/presentation/siby

[37] Alexander Sjösten, Daniel Hedin, and Andrei Sabelfeld. 2021. EssentialFP: Exposing the Essence of Browser Fingerprinting. In *IEEE European Symposium on Security and Privacy Workshops, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*. IEEE, 32–48. https://doi.org/10.1109/EUROSPW54576.2021.00011

[38] Peter Snyder, Antoine Vastel, and Ben Livshits. 2020. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 2 (2020), 26:1–26:24. https://doi.org/10.1145/3392144

[39] Cristian-Alexandru Staicu, Daniel Schoepe, Musard Balliu, Michael Pradel, and Andrei Sabelfeld. 2019. An Empirical Study of Information Flows in Real-World JavaScript. In *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security, CCS 2019, London, United Kingdom, November 11-15, 2019*. ACM, 45–59. https://doi.org/10.1145/3338504.3357339

[40] Oleksii Starov and Nick Nikiforakis. 2017. Extended Tracking Powers: Measuring the Privacy Diffusion Enabled by Browser Extensions. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3-7, 2017*. ACM, 1481–1490. https://doi.org/10.1145/3038912.3052596

[41] Yash Vekaria, Yohan Beugin, Shaoor Munir, Gunes Acar, Nataliia Bielova, Steven Englehardt, Umar Iqbal, Alexandros Kapravelos, Pierre Laperdrix, Nick Nikiforakis, Jason Polakis, Franziska Roesner, Zubair Shafiq, and Sebastian Zimmeck. 2025. SoK: Advances and Open Problems in Web Tracking. *CoRR* abs/2506.14057 (2025). https://doi.org/10.48550/ARXIV.2506.14057 arXiv:2506.14057

[42] Daniel Lowe Wheeler. 2016. zxcvbn: Low-Budget Password Strength Estimation. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016*. USENIX Association, 157–173. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/wheeler

[43] Qianru Wu, Qixu Liu, Yuqing Zhang, Peng Liu, and Guanxing Wen. 2016. A Machine Learning Approach for Detecting Third-Party Trackers on the Web. In *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9878)*. Springer, 238–258. https://doi.org/10.1007/978-3-319-45744-4_12

[44] Mengfei Xie, Jianming Fu, Jia He, Chenke Luo, and Guojun Peng. 2020. JTaint: Finding Privacy-Leakage in Chrome Extensions. In *Information Security and Privacy - 25th Australasian Conference, ACISP 2020, Perth, WA, Australia, November 30 - December 2, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12248)*. Springer, 563–583. https://doi.org/10.1007/978-3-030-55304-3_29

[45] Qinge Xie, Manoj Vignesh Kasi Murali, Paul Pearce, and Frank Li. 2024. Arcanum: Detecting and Evaluating the Privacy Risks of Browser Extensions on Web Pages and Web Content. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/xie-qinge

[46] Zhiju Yang, Weiping Pei, Monchu Chen, and Chuan Yue. 2022. WTAGRAPH: Web Tracking and Advertising Detection using Graph Neural Networks. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*. IEEE, 1540–1557. https://doi.org/10.1109/SP46214.2022.9833670

A Breakage Evaluation

Our approach for the automated validation of tracking requests, described in Section 4.2, tests whether an identifier set in client-side storage reappears in a matching request once modified into a canary. However, this active tampering with client-side storage may disrupt the normal operation of some trackers, systematically suppressing certain classes of tracking requests. In particular, we observed that around 7% of the requests in our dataset do not have any matching request after the introduction of the canary.

To evaluate the potential systematic breakage introduced by our testing procedure, we measure the prevalence of tracking requests without any matching request after setting the canary across different trackers. Specifically, we extract all trackers with at least 20 requests (to ensure a meaningful sample size) and for which more than 50% of requests have no matching counterparts. For example, if some tracker receives 30 requests and more than 15 of them have no matching requests, it may exhibit systematic breakage. From this experiment, it emerges that 13 trackers satisfy this condition, corresponding to 524 requests without matching counterparts. This is 21% of the total of such requests and less than 2% of the requests that underwent our testing procedure.

Among the trackers that may exhibit systematic breakage according to our evaluation, we observe that fundingchoicesmessages.google.com receives requests whose URLs are generated by a preceding, non-tracking request to the same service and include a path segment that changes on each page visit. Since our template only generalizes path segments when a syntactic match is detected, and no such match is found in this case, each request is treated as distinct. This specific case thus does not appear to be correlated with breakage, but is rather an artifact of our matching strategy. This is a positive finding, because it reveals a shortcoming of our matching heuristics rather than a systematic breakage introduced by our automated approach. A manual analysis of the other cases did not reveal particularly useful insights about breakage, which may also be explained by the just occasional lack of matching requests in our dataset.

B Top Trackers

This appendix provides supplementary tables supporting the results presented in Section 5. Table 6 compares the most popular trackers exposed by each variant of syntactic matching and taint tracking. This comparison highlights where different techniques agree on the detection of specific trackers and where their outcomes diverge. Supplementally, Table 7 reports the top trackers detected by the most precise techniques, namely SyntacticC and taint tracking, which are not present in Disconnect [13]. This analysis highlights the agreement between these techniques in identifying potentially new trackers missed by reputable filter lists.

Table 6: Top trackers detected using different analysis techniques.

Syntactic (S)	#	**SyntacticNR (S_{NR})**	#	**SyntacticC (S_C)**	#	**Taint (T)**	#
www.facebook.com	799	www.facebook.com	793	www.facebook.com	792	region1.google-analytics.com	1,923
region1.google-analytics.com	511	region1.google-analytics.com	502	region1.google-analytics.com	495	region1.analytics.google.com	1,356
region1.analytics.google.com	441	region1.analytics.google.com	429	region1.analytics.google.com	410	www.facebook.com	648
mc.yandex.com	374	mc.yandex.com	373	mc.yandex.com	373	www.google-analytics.com	504
pagead2.googlesyndication.com	360	www.google.com	303	www.google.com	301	mc.yandex.com	372
www.google.com	310	accounts.google.com	224	accounts.google.com	224	analytics.tiktok.com	203
accounts.google.com	226	analytics.tiktok.com	214	analytics.tiktok.com	214	analytics-ipv6.tiktokw.us	196
analytics.tiktok.com	214	analytics-ipv6.tiktokw.us	206	analytics-ipv6.tiktokw.us	206	googleads.g.doubleclick.net	173
analytics-ipv6.tiktokw.us	206	www.google-analytics.com	199	www.google-analytics.com	184	www.google.com	156
www.google-analytics.com	204	pagead2.googlesyndication.com	158	gum.criteo.com	156	www.google.at	138

Table 7: Top-10 trackers detected by SyntacticC (S_C) and Taint Tracking (T) after Disconnect filtering (−Disconnect).

S_C **−Disconnect**	#	T **−Disconnect**	#
analytics-ipv6.tiktokw.us	206	analytics-ipv6.tiktokw.us	196
fondleaegipan.shop	63	fondleaegipan.shop	63
ua.yektanet.com	41	ua.yektanet.com	40
flushpersist.com	29	flushpersist.com	28
api.mediaad.org	25	b7510.com	25
b7510.com	25	mpc-prod-25-s6uit34pua-wl.a.run.app	23
mpc-prod-25-s6uit34pua-wl.a.run.app	23	mpc-prod-24-s6uit34pua-uw.a.run.app	21
3b30612d89.a910d264c4.com	22	mpc-prod-27-s6uit34pua-uk.a.run.app	19
mpc-prod-24-s6uit34pua-uw.a.run.app	22	jamssp.yektanet.com	19
bid.sparteo.com	20	ingestion.smartocto.com	17