DE GRUYTER OPEN Proceedings on Privacy Enhancing Technologies ; 2017 (1):79–99

Muhammad Ikram*, Hassan Jameel Asghar, Mohamed Ali Kaafar, Anirban Mahanti, and Balachandar Krishnamurthy

Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning

Abstract: Numerous tools have been developed to aggressively block the execution of popular JavaScript programs in Web browsers. Such blocking also affects functionality of webpages and impairs user experience. As a consequence, many privacy preserving tools that have been developed to limit online tracking, often executed via JavaScript programs, may suffer from poor performance and limited uptake. A mechanism that can isolate JavaScript programs necessary for proper functioning of the website from tracking JavaScript programs would thus be useful. Through the use of a manually labelled dataset composed of 2,612 JavaScript programs, we show how current privacy preserving tools are ineffective in finding the right balance between blocking tracking JavaScript programs and allowing functional JavaScript code. To the best of our knowledge, this is the first study to assess the performance of current web privacy preserving tools in determining tracking vs. functional JavaScript programs.

To improve this balance, we examine the two classes of JavaScript programs and hypothesize that tracking JavaScript programs share structural similarities that can be used to differentiate them from functional JavaScript programs. The rationale of our approach is that web developers often "borrow" and customize existing pieces of code in order to embed tracking (resp. functional) JavaScript programs into their webpages. We then propose one-class machine learning classifiers using syntactic and semantic features extracted from JavaScript programs. When trained only on samples of tracking JavaScript programs, our classifiers achieve accuracy of 99%, where the best of the privacy preserving tools achieve accuracy of 78%.

The performance of our classifiers is comparable to that of traditional two-class SVM. One-class classification, where a training set of only tracking JavaScript programs is used for learning, has the advantage that it requires fewer labelled examples that can be obtained via manual inspection of public lists of well-known trackers. We further test our classifiers and several popular privacy preserving tools on a larger corpus of 4,084 websites with 135,656 JavaScript programs. The output of our best classifier on this data is between 20 to 64% different from the tools under study. We manually analyse a sample of the JavaScript programs for which our classifier is in disagreement with all other privacy preserving tools, and show that our approach is not only able to enhance user web experience by correctly classifying more functional JavaScript programs, but also discovers previously unknown tracking services.

Keywords: Machine learning, one class SVM, pu-learning, measurements, JavaScripts, tracking, privacy, usability, security

DOI 10.1515/popets-2017-0006
Received 2016-05-31; revised 2016-09-01; accepted 2016-09-02.

1 Introduction

JavaScript programs are frequently used to track users and to tailor advertisements on websites to the browsing history and web activities of users [22, 47, 56]. Recently, a class of tools (including web browser plugins) have been developed in an attempt to preserve user privacy (e.g., NoScript [14], Ghostery [8], and Adblock Plus [1]), which block JavaScript programs and other components of a webpage that may compromise user privacy and enable tracking. However, aggressive blocking can hinder proper functioning of the website and impact user's browsing experience [19, 46] (an example of how a tool may block content necessary for proper functioning of a webpage is shown in Appendix A.5). A mechanism that can properly isolate JavaScript programs necessary for

***Corresponding Author: Muhammad Ikram:** Data61, CSIRO and UNSW, Sydney, Australia, E-mail: muhammad.ikram@data61.csiro.au
Hassan Jameel Asghar: Data61, CSIRO, Sydney, Australia, E-mail: hassan.asghar@data61.csiro.au
Mohamed Ali Kaafar: Data61, CSIRO, Sydney, Australia, E-mail: dali.kaafar@data61.csiro.au
Anirban Mahanti: Data61, CSIRO, Sydney, Australia, E-mail: anirban.mahanti@data61.csiro.au
Balachandar Krishnamurthy: ATT Research Lab, New York, USA, E-mail: bala@att.com

CC BY-NC-ND

"legitimate" functioning of the website from others that are likely to be privacy-intrusive would therefore be useful.

Web tracking in general happens through the use of numerous technologies, e.g., cookies, supercookies, Flash cookies, tracking pixels, ETag cookies, HTTP referrers and JavaScript programs [38, 50][1]. Third-party tracking, where "unauthorised" third parties retrieve information from the "first party" websites visited by users, enable a plethora of services including analytics, advertisement and online social interactions. While third-party tracking [41] may happen through various techniques, trackers frequently use JavaScript programs [22, 47, 52, 56] for tracking, and as such, most tracking can be avoided by controlling the execution of JavaScript programs on webpages. We refer to these JavaScript programs as *tracking*. Consequently, most privacy preserving tools (PP-Tools in short) are either based on pre-defined (black)lists of URLs (or patterns of URLs) of third-party trackers for which the execution of JavaScript code is blocked or simply rely on blocking any third-party JavaScript program.

However, not all JavaScript programs are used for tracking and many are *essential* for proper functioning of a website, e.g., JavaScript programs that enable media players to show an embedded video on a webpage. We refer to these JavaScript programs as *functional* JavaScript programs. In this paper, we show that the current generation of PP-Tools are unable to achieve a balance between blocking tracking and functional JavaScript programs, which we believe has contributed to their poor uptake with a recent study showing about 3 to 20% uptake among web users [42].

This motivates our goal to develop a machine learning classification framework that is more effective in separating tracking from functional JavaScript programs on a webpage. We impose two important design constraints. One is to avoid detection through regular expressions based on blacklists, which is used by existing PP-Tools and, as measured later in the paper, is ineffective. Another constraint is to enable classification using a small single class of JavaScript programs that are known to be either exclusively functional or tracking. Our motivation for this is that in the real-world one can expect to have knowledge of only a subset of tracking or functional JavaScript programs, i.e., we cannot expect to have an exhaustive list of either classes of programs. The key rationale of our proposed approach is that web developers often embed JavaScript code used by popular tracking libraries or re-use pieces of known JavaScript code which they customise. Likewise, code of several functional web components (including search buttons, media players embedding, shopping carts, content fetching, etc.) are generally borrowed from previously published code. From that perspective, our technique resembles approaches taken to detect code plagiarism and malware code signatures.

In this paper, we develop machine learning approaches that classify functional and tracking JavaScript programs based on syntactic and semantic features extracted from a number of JavaScript programs. We find that traditional two-class support vector machine (SVM) trained on labelled data from both functional and tracking JavaScript programs can accurately distinguish these JavaScript programs. More importantly, we show that one-class machine learning classifiers, namely one-class SVM and positive and unlabelled (PU) learning, trained using only tracking JavaScript programs can achieve performance comparable to two-class SVM. We believe the latter approach is more practical as it requires fewer labelled samples for training which can be obtained from well-known tracking and advertising services, e.g., through blacklists of PP-Tools (although manual effort would still be required to remove wrong labels, since, as we show later, not all JavaScript programs from known trackers are tracking JavaScript programs). The proposed scheme only works for tracking via JavaScript programs, other techniques such as pixel abased tracking are excluded. In summary, we make the following contributions:

– We propose two machine learning approaches for automatic classification of tracking and functional JavaScript programs that rely only on partial knowledge of the former class. Instead of using static code analysis or regular expression matching on blacklists, we use an automated way to extract features from JavaScript programs using syntactic and structural models proposed in the literature to quantitatively determine similarity between functional and tracking JavaScript programs. Our proposed approaches achieve accuracy[2] of up to 99%, well above the accuracy achievable by existing PP-Tools ($\leq$ 78%) as validated through our manually labelled dataset.

1 Tracking can also be generalised to the use of multiple user identifiers for host-tracking including IP addresses, login IDs and web browser user-agents. This is generally referred to as stateless tracking. Interested readers may refer to [55] for further details.

2 Accuracy is defined as the sum of true positives and negatives normalized by the population size (total number of JavaScript programs). In our study, the tracking JavaScript programs constitute the positive class.

- We evaluate the effectiveness of five major PP-Tools by comparing their output against a set of 2,612 manually labelled JavaScript programs extracted from 95 different domains. Among these five, Ghostery achieves the best balance between true and false positive rates, 0.65 to 0.08. Others who fare better in terms of the false positive rate (≤ 0.06) pay the penalty with a considerably lower true positive rate of 0.44 or less. NoScript achieves the highest true positive rate (0.78), at the expense of the poorest false positive rate (0.21). Our results indicate that existing PP-Tools need considerable improvement in finding an optimal balance between true and false positives. To the best of our knowledge, this is the first study that analyses and assesses the performance of current privacy preserving tools in terms of determining tracking vs. functional JavaScript programs.
- We run our classifiers on a larger dataset of 4,084 websites, representing 135,656 JavaScript programs and compare their output against the above mentioned PP-Tools to analyse their respective effectiveness in the *wild*. Choosing our best classifier as a benchmark, we observe that NoScript, Ghostery and Adblock Plus agree with the output of our classifier between 75 to 80%, whereas Disconnect and Privacy Badger showed an agreement of only 36% and 43%, respectively. Between 11% to 14% of the JavaScript programs labelled as functional by our classifier were contradictorily classified as tracking by the first three. The remaining two PP-Tools had this number less than 7%.
- From this larger dataset we randomly sample two subsets of 100 JavaScript programs each, corresponding to the two sets of JavaScript programs labelled as functional and tracking by our classifier in disagreement with *all* five PP-Tools. Through manual inspection, we found that our classifier was correct in classifying 75 out of 100 JavaScript programs it labelled tracking, and 81 out of 100 JavaScript programs it labelled functional, meaning that the PP-Tools were only correct in labelling 25 and 19 JavaScript programs in the two subsets, respectively. We discuss, with examples, the main reasons for misclassification by our classifier and PP-Tools. Notably, we further show how our classifier is capable of revealing previously unknown tracking services simply by relying on JavaScript code structure and semantic similarity from popular tracking services.

The rest of this paper is organised as follows. In Section 2, we give a background on JavaScript based web tracking and how the PP-Tools considered in our study attempt to mitigate this. We discuss our main objectives, methodology and data collection in Section 3. In Section 4, we evaluate the effectiveness of PP-Tools using our labelled data set. Details on our choice of classifiers and their validation are described in Section 5. We evaluate the effectiveness of our classifiers and PP-Tools in the wild in Section 6. In Section 7, we discuss possible limitations and challenges as well as avenues for improvement of our proposed scheme. We review the related work in Section 8 and provide some concluding remarks in Section 9.

2 Background

2.1 Web Tracking and JavaScript programs

A typical webpage consists of several web-components, e.g., JavaScript codes, Flash-content, images, CSS, etc. When a user opens a website in a web browser, the fetched webpage typically generates several other HTTP(S) connections for downloading additional components of the webpage. These components can be downloaded from the website visited by the user (referred to as first-party domain) or downloaded from other third-party domains. Here, we focus on one type of web-component, namely JavaScript codes, which is loaded both from first- and third-party domains. JavaScript programs are widely used by ad networks, content distribution networks (CDNs), tracking services, analytics platforms, and online social networks (e.g., Facebook uses them to implement plugins).

Figure 1 illustrates a typical scenario of web tracking via JavaScript codes. Upon fetching a webpage from first-party domains (steps 1 & 2), the user's web browser interprets the HTML tags and executes JavaScript programs within the HTML `script` tags. JavaScript code execution enables the web browser to send requests to retrieve additional content from third-party domains (step 3). Depending on the implemented functionalities, the JavaScript programs can be considered as useful (functional), e.g., fetching content from a CDN, or as tracking. In the latter case, when the webpage is completely rendered (step 4), the JavaScript codes track user's activities on the webpage [52], write to or read from the cookie database [35, 46] (steps 5 & 6), or reconstruct user identifiers [22, 45]. Tracking JavaScript programs may also be used to transfer private and sensitive information to third-party domains (step 7) [38].

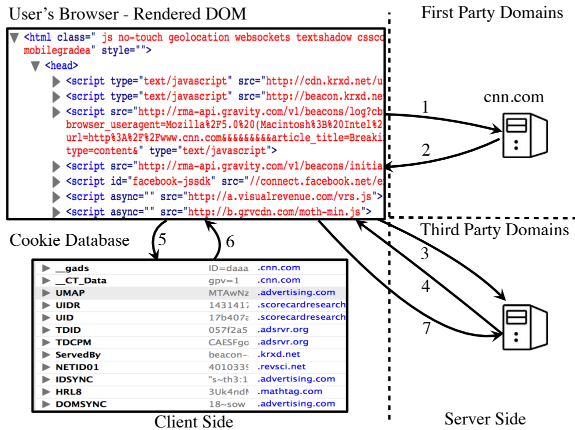


Fig. 1. Overview of a webpage rendering process and web tracking. Websites (in this case `cnn.com`) use third-party domains for content provisions and analytics services.

2.2 Privacy Preserving Tools (PP-Tools)

In the following we briefly introduce five common PP-Tools considered in our study.

NoScript (NS)[3] blocks JavaScript programs, Java programs, Silverlight, Flash and other executable content on a webpage that may undermine security including tracking [14]. The default behaviour is to deny, thus only allowing content that the user has explicitly permitted (whitelists). This however requires frequent user intervention and may cause usability issues.

Adblock Plus (AP) is primarily an advertisement blocking tool based on blacklists [1]. It provides the option to choose from different blacklists, e.g., *EasyList* [5], to block unwanted advertisements. It searches the rendered HTML page (DOM tree) through regular expressions and blocks the downloading of web-components such as web bugs, ads or JavaScript programs that belong to blacklisted tracking and advertising services.

Disconnect (DC) is also a blacklist based tool [3], which mainly blocks third-party tracking cookies and JavaScript programs from social networks such as Facebook and Twitter.

Ghostery (GT) finds and disables cookies and scripts that are used for tracking [8]. It also searches the DOM tree through regular expressions for advertisers and trackers identified in a predefined blacklist. It provides feedback to the user to selectively unblock tracking domains.

Privacy Badger (PB) uses a blacklist [16] of third-party tracking cookies that track users on multiple first-party domains. It further uses a heuristic algorithm that blocks content (JavaScript programs and ads) from third-party domains who either read high entropy cookies, and read cookies on multiple (at least 3) first-party domains, and do not comply with an acceptable "Do Not Track" policy statement [7].

3 Although NoScript is primarily a security add-on, it can also be viewed as an extremely aggressive PP-Tool which blocks third-party advertisements, analytics, and trackers.

3 Methodology

3.1 Objectives and Overview of our Work

Our objectives can be summarised as follows:

1. Assess the effectiveness of existing PP-Tools in terms of correctly classifying tracking and functional JavaScript programs.
2. Propose an effective machine learning approach to classify tracking and functional JavaScript programs, trained on a subset of tracking JavaScript programs only.
3. Analyse PP-Tools and our classifier(s) in the wild and identify any trackers missed by the PP-Tools.

For the first objective, we use a labelled dataset, composed of JavaScript programs, to evaluate the outcome of each of the aforementioned five PP-Tools as observed by a user navigating through a list of webpages. We used a set of 95 websites and extracted 2,612 unique JavaScript programs which were manually inspected and labelled as either tracking or functional according to a set of pre-identified rules (cf. Table 1). This dataset is referred to as the *labelled dataset* (Section 3.3).

For the second objective, we build classifiers trained only on partial knowledge, i.e., on knowledge of labels from a subset of tracking JavaScript programs only. These tracking JavaScript programs were extracted from the above mentioned labelled dataset. The accuracy of our classifiers is also validated using the labelled dataset, and compared against a traditional two-class classifier trained on both functional and tracking JavaScript programs from the labelled dataset.

We assess the effectiveness of our classifiers and the PP-Tools *in the wild*, i.e., our third objective, by further extracting 135,656 JavaScript programs from a set of 4,084 websites. We call this dataset, the *wild dataset* (Section 6.1). We apply and compare results of the PP-Tools and our classifiers on this wild dataset. We also manually inspect two random samples of 100 JavaScript programs each on which our classifier and all the other PP-Tools disagree. As a result, we identify trackers that are missed by the PP-Tools and reveal several functional JavaScript programs that are being mistakenly blocked by the PP-Tools.

3.2 Obtaining JavaScript programs

We developed a crawler based on the Selenium webdriver [21] for automated downloading of target webpages. The Selenium framework allows us to retrieve the whole content of a rendered webpage (DOM tree). We configured our crawler to wait 180 seconds for each target webpage to be rendered. The yielded, rendered webpages were cached, i.e., stored locally. In a post-processing step, we checked if the rendering of any webpage stalled. If so, we reran the crawler on the webpage by increasing the waiting time until the webpage is rendered. Our crawling process consisted of the following steps: 1) use Firefox's Selenium webdriver to fetch the landing page from a given list of webpages, 2) dump the DOM tree and save it locally, 3) parse all `script` tags to find in-page JavaScript programs, i.e., programs embedded within the HTML `script` tag, and save them in files, and 4) download all external JavaScript programs, i.e., programs that are linked via external URLs.

For extracting JavaScript programs when the PP-Tools are turned off, referred to as PP-Tools *off*, we obtain JavaScript programs for both labelled and wild datasets as above. For the labelled dataset, these JavaScript programs are additionally assigned a label (tracking or functional) using Firefox and Firebug as detailed in Section 3.3. Likewise, when PP-Tools are turned on (referred to as PP-Tool *on*), we simultaneously obtain the set of JavaScript programs for each of the PP-Tools under consideration, by creating a separate Firefox profile for each tool.

Due to their dynamic nature, the content of webpages is likely to change when accessed at different times. This means that the JavaScript programs, which load dynamic content such as ads, may also change on a webpage. To ensure that the number and type of downloaded JavaScript programs were consistent, we executed the five PP-Tools profiles (i.e., PP-Tools *on*) and the PP-Tools *off* setting in parallel to access the target webpages (with a wait time of 180 seconds per webpage) simultaneously using separate VMs running on a single host. However, this still leaves open the question whether the JavaScript programs thus obtained were the same across different profiles. To check this, we simultaneously accessed a set of 100 target websites with 6 crawlers (all with PP-Tools *off*) running in parallel on separate machines on the same host and stored the resulting JavaScript programs. The in-page JavaScript programs were saved using the domain name of the webpage containing them followed by the sequence in which they appear in the rendered webpage, whereas external JavaScript programs were saved using the domain name associated with them.

As a first check, the number of JavaScript programs obtained through each crawler was the same, i.e., 2,049. We then used Python's diff utility, i.e., `difflib` [2], to determine differences between 6 copies of the same JavaScript program by taking the first crawler's JavaScript programs as reference. By inspecting the output of the diff utility, we found that 95% of these programs were completely identical. The rest, i.e., 5%, were different only in the values of the constants assigned to variables. A further examination of these codes revealed that 75% of the constants were cookie values whilst the remaining 25% were ad tag values read by the JavaScript codes. For instance, JavaScript codes like Google Tag Manager (`gtm.js`) and Facebook "connect" (`sdk.js`) read tags and cookie values that were different in our simultaneous crawlers. However, note that differences in the values of constants do not change the syntax or structure of JavaScript programs, nor their nature (functional or tracking). Thus, our methodology ensures consistent view of JavaScript programs for all profiles.

3.3 The Labelled Dataset

We defined a set of 12 rules to guide our manual classification of JavaScript programs as tracking or functional. These rules were applied to the JavaScript programs present in the stored (cached) copies of rendered webpages obtained through the method described in Section 3.2. The rules were derived after detailed review of example JavaScript programs and discussion among the authors. Guided by the derived labelling rules, a single expert analysed the JavaScript programs present in the target websites, i.e., labelled dataset, and carried out the actual labelling. There may be discrepancies in judging the nature of JavaScript programs in case of two or more reviewers. However, our classification methods (discussed in Section 5) are robust since the classifiers can be trained on whatever the view of functional/tracking JavaScript codes is agreed upon. Note that labelling was done prior to applying our classifiers to ensure that it was not biased in giving our classifiers any advantage over PP-Tools. As we discuss in Section 7.2.3, the labelling process could be crowd-sourced to resolve any issues, where the label of JavaScript programs may be decided based on majority decision or any other agreed upon method. Table 1 summarises the classification rules we used. Rules R1 and R2 in Table 1 label all JavaScript programs that create panels and set

Rule	JS	#	Description
R1	✗	216	All JS that create panels and set margins for ads
R2	✗	115	All JS that access and display ads
R3	✗	45	All social media widgets
R4	✗	324	All in-page JS that include external JS from third-party analytics and advertisers
R5	✗	353	All external JS from third-party analytics and advertisers
R6	✗	180	All cookie enablers, readers or writers
R7	✓	542	All external JS that provide useful functionality such as navigation menus, search and login
R8	✓	509	All in-page JS that provide useful functionality
R9	✓	132	All JS that fetch content from first-party content domains or third-party CDNs
R10	✗	103	All JS in hidden iframe that belong to third-party analytics, advertisers and social media
R11	✗	40	All JS in hidden iframe that enable, read or modify cookies
R12	✓	53	All JS that track mouse or keyboard events

Table 1. Rules for labelling JavaScript programs - R stands for Rule; JS stands for JavaScript program; # denotes the number of JavaScript programs satisfying the corresponding rule in the labelled dataset; ✗ represents tracking JavaScript programs and ✓ represents functional JavaScript programs.

margins for ads within a webpage or the ones that fetch ads and display them, as tracking. Likewise we also label all programs that enable social media widgets, such as Facebook 'Likes' and Twitter 'Follow', as tracking (rule R3). We went through all the `script` tags, i.e., in-page JavaScript programs, and labelled them as tracking if they include an external JavaScript code (within the body) belonging to a known third-party advertiser or analytic service (R4). It follows that all *external* JavaScript programs contained in a webpage that belong to a known third-party advertiser or analytics are labelled tracking (R5).

JavaScript programs					
External	In-page	Average	Total	Tracking	Functional
1,353	1,256	27.5	2,612	57%	43%

Table 2. Characteristics of JavaScript programs from 95 websites in our labelled dataset.

To uniquely identify users, trackers enable, read, write or update cookies on the users' machine. We used Firebug's 'Cookies' panel to obtain a list of cookies. This list contains the name and ID of a cookie as well as the domain the cookie is stored for. We then searched the DOM tree by the cookie name and ID to find the JavaScript that enables, reads or modifies this cookie. Note that the JavaScript corresponding to the cookie contains its name and ID in its source code. If the domain for which the cookie is stored is a known analytic or advertiser then we mark the JavaScript as tracking; otherwise the JavaScript is marked as functional (R6). We are aware that, in principle, cookie names can be dynamically generated. However, during manual labelling we did not encounter this except for session cookies (which may or may not be set by JavaScript programs).

Similarly, we used Firebug's 'Network' panel to identify invisible iframes that belong to third-party analytics, advertisers, or social media. We label the JavaScript programs belonging to these iframes as tracking JavaScript programs (R10). We also label the JavaScript programs inside iframes that enable, read, or modify cookies as tracking JavaScript programs (cf. rules R10 and R11). Analytics and advertisers employ hidden iframes by specifying the height and width of the iframe to zero, one, or by positioning it so that it is out of the visible area on a webpage. A hidden iframe is positioned so that when a user interacts with a certain component of a webpage, his action and potentially the information contained in corresponding cookie(s) are redirected to the advertiser's or analytic's networks.

All JavaScript programs that facilitate access to contents and services related to the target (visited) webpage are labeled as functional. For instance, webpages contain JavaScript programs that enable search boxes, accessibility options, authentication services, shopping carts, prompts, navigation menu and breadcrumbs (rules R7 and R8). Similarly, some JavaScript programs are used to track mouse and keyboard events, such as right click or caps lock on or off (R12) while others are used to retrieve content from either first-party content domains or third-party CDNs like Akamai (R9). We created a manual list of well-known third-party CDNs to differentiate them from other content providers.

While we have tried to remain as objective as possible with our definition of functional JavaScript programs, there might still be instances in which a JavaScript program satisfies both tracking and function rules. In such a case, we consider it as tracking. For instance, we considered social widgets to be privacy-intrusive as they allow social networks to track users [24]; however, these could potentially be perceived as providing functional features as they allow users to interact with their social network. Notably, mouse or keyboard related JavaScript programs are only considered functional if they do not send information to third-party servers (unlike JavaScript programs that belong to e.g., 'Moat'[13] that track users and send collected data to third-party servers). Likewise, JavaScript programs that track user's comments and send them to an

external server (e.g., 'Disqus'[4]) were labelled as tracking JavaScript programs.

We selected 95 web domains such that 50 of them were the top 50 Alexa websites, and the remaining 45 were randomly chosen from websites with Alexa rank in the range 5,000-45,000. In total, we collected 2,612 JavaScript programs, which consisted of 1,376 tracking and 1,236 functional JavaScript programs according to our labelling. This constitutes our labelled dataset. Note that the 2,612 JavaScript programs obtained correspond to the PP-Tool *off* setting in our terminology. Table 2 summarizes the key characteristics of the JavaScript programs in our labelled dataset. Note that 43% of the JavaScript programs are functional indicating that the scale of the problem we are addressing is significant. Misclassifying even a small part of this can significantly impair user's web experience.

4 Effectiveness of PP-Tools

We first introduce metrics used in our evaluation.

4.1 Aggressiveness of PP-Tools and Surrogate JavaScript Programs

Let H denote the set of DOM trees (webpages) in a given dataset (in our case, H could be the labelled or the wild dataset). Let J denote the set of all JavaScript programs from H. Given a DOM tree $h \in H$, $\mathsf{js}(h)$ denotes the set of JavaScript programs contained in h. This corresponds to the PP-Tools' *off* setting. Let p denote a given PP-Tool. Then $p(h)$ denotes the DOM tree obtained after applying p to h, and $\mathsf{js}(p(h))$ denotes the set of JavaScript programs in the DOM tree $p(h)$. This corresponds to the PP-Tools' *on* setting. We have the condition that $\mathsf{js}(p(h)) \subseteq \mathsf{js}(h)$, i.e., the set of JavaScript programs obtained when a PP-Tool is *on* will always be a subset of the set of JavaScript programs obtained when the PP-Tool is *off*.

However, in our experiments, we found that some PP-Tools, such as Ghostery and NoScript, replace some JavaScript programs by new JavaScript programs, called *surrogate* JavaScript programs, to ensure smooth viewing of the webpage [9, 18]. For instance, Ghostery replaces the JavaScript code enabling Twitter 'Follow' button with a surrogate JavaScript code that displays a Ghostery button, while still blocking the functionality of the Twitter button. When a surrogate JavaScript codes is present in the DOM tree $p(h)$, we consider the replaced JavaScript code to be *blocked* by the PP-Tool p, and as a result we do not include the surrogate JavaScript code in the overall count. That is, we consider this situation as if the surrogate JavaScript program is absent and the original JavaScript program is blocked, thus ensuring that the PP-Tool is not penalised. This is consistent with our manual inspection of the surrogate lists of Ghostery and NoScript, as all the surrogates turned out to be functional (according to our labelling rules).

We define the *aggressiveness* of a privacy-preserving tool p as

$$a(h) = 1 - \frac{|\mathsf{js}(p(h))|}{|\mathsf{js}(h)|},$$

which is the fraction of JavaScript programs blocked by a PP-Tool p from the DOM tree h. In settings with PP-Tools *off*, aggressiveness is 0. A PP-Tool is said to be *aggressive* if it has high aggressiveness (i.e., closer to 1) for a high portion of webpages from H.

4.2 Evaluation Results

We configured the PP-Tools so that their definitions of tracking and functional JavaScript programs are consistent with our labelling rules in Table 1. Specifically, we configured NoScript so that it does not allow iframes, consistent with rules R10 and R11 in Table 1. Likewise, we disabled the default option "Allow all non-intrusive ads" for Adblock Plus and we used The EasyList [5] and Fanboy's Social Blocking List [15] as blacklists for Adblock Plus. We enabled "Blocking all Trackers and Cookies" option for Ghostery. We used the default settings for Privacy Badger and Disconnect since they do not provide the user with any configuration options. Summary of the set up used for PP-Tools is shown in Table 9 in Appendix A.1.

To account for any "heating" phase of the PP-Tools, we ran each of the tools over another set of 100 random webpages (with Alexa rank between 5,000 and 50,000, *excluding* the webpages present in the labelled and wild datasets). We call these *heat-up* webpages. Using the same rules for labelling tracking and functional JavaScript programs described in Section 3.3 (and summarized in Table 1), we determined the types and the distribution of trackers in these heat-up webpages. We focused on external JavaScript programs since internal JavaScript programs are not associated with domain names of trackers. The external JavaScript programs were saved with the domain names associated with them. We observed that the tracking JavaScript programs belonged to a total of 130 unique trackers

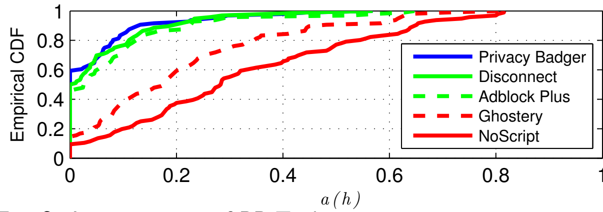


Fig. 2. Aggressiveness of PP-Tools.

(domains). Only 8% of the trackers were found in only one webpage, meaning that the overwhelming majority (92%) of the trackers were present in at least two or more webpages. Thus, we expect all analysed PP-Tools, and in particular Privacy Badger (note that part of the heuristic used by Privacy Badger needs the tracker to be present in at least three webpages), to block JavaScript programs belonging to most trackers from this list. Increasing the number of heat-up webpages further may positively affect the output of PP-Tools, especially Privacy Badger. However, we believe that 100 webpages is sufficient since we expect it to be way above the number of domains an average user would visit over a reasonably short period of time. For instance, according to the marketing research company, Nielsen, the average number of domains visited by a US citizen over a month ranged between 89 and 99 between the years 2010 to 2013, inclusive [26]. Figures 4 and 5 in the appendix show the distribution of trackers in the heat-up webpages.

JS	PP-Tools On				
	NS	GT	AP	DC	PB
External	570	813	1,141	1,206	1,230
In-page	1,118	1,173	1,197	1,218	1,208
Total	1,688	1,986	2,338	2,424	2,438
Average per webpage	17.6	20.1	24.4	25.3	25.4
Blocked (%)	35.38	24.0	10.5	7.2	6.7
Allowed (%)	64.6	76.0	89.5	92.8	93.3

Table 3. Characteristics of JavaScript programs (JS) from the labelled dataset marked as functional (allowed) by PP-Tools.

Table 3 shows the view of JavaScript programs from the 95 webpages in the labelled dataset when the different PP-Tools were *on*. The top-half of the table shows the number of JavaScript programs (in-page and external) that are allowed when a particular PP-Tool is used. The last two rows of the table show the number of JavaScript programs that are blocked and allowed as a percentage of the total number of JavaScript programs in our dataset (2,612).

We further analyse the performance of the PP-Tools by measuring their aggressiveness, i.e., $a(h)$. Figure 2 shows the cumulative distribution function of $a(h)$ for the five PP-Tools when applied on the 95 websites. We observe that NoScript's aggressiveness is more than 0.2 for about 60% of the web domains whilst Ghostery's aggressiveness is more than 0.2 for about 40%. In contrast, Adblock Plus, Disconnect, and Privacy Badger have an aggressiveness of more than 0.2 for only about 10% of domains, which indicates that they are comparatively less aggressive in blocking JavaScript programs.

PP-Tool	Tracking		Functional	
	Blocked	Allowed	Blocked	Allowed
NoScript	0.78	0.22	0.21	0.79
Ghostery	0.65	0.35	0.08	0.92
Adblock Plus	0.44	0.56	0.06	0.94
Disconnect	0.40	0.60	0.06	0.94
Privacy Badger	0.37	0.63	0.06	0.94

Table 4. Comparison of the output of PP-Tools against our labelled set of tracking and functional JavaScript programs. true positives and negatives, false positives and negatives.

We are also interested to know whether there is an inverse relation between aggressiveness and effectiveness of PP-Tools, i.e., an aggressive PP-Tool breaks useful functionality in a webpage (by incorrectly blocking functional JavaScript programs), and a less aggressive PP-Tool allows more tracking JavaScript programs go undetected. Effectiveness is defined as the balance between correctly blocking tracking JavaScript programs (true positives) and incorrectly blocking functional JavaScript programs (false positives).

For this, we measure the true positive and false positive rates of each of the PP-Tools. Our results are shown in Table 4. We find that PP-Tools' true positive rates vary from 37% to 78% and false positives range from 6% to 21%. Not surprisingly, NoScript has the highest true positive rate of 78% at the expense of the poorest false positive rate of 21%. Adblock Plus[4], Disconnect, and Privacy Badger fair better in terms of false positive rate (6%) but pay the penalty with considerably lower true positives rates of 44%, 40% and 37%, respectively. Both Ghostery and NoScript achieve the lowest average error rate (AER) of 0.215, where AER is defined as the average of false positive and negative rates. However, Ghostery is better in terms of allowing functional JavaScript programs, achieving a false positive rate of only 8% with a lower true positive rate (65%) than NoScript.

To summarise, these results suggest that current PP-Tools are ineffective in terms of striking a good balance between limiting tracking and adversely affect-

4 Adblock Plus can be configured using additional complimentary lists such as EasyPrivacy [6]. This may improve its true positive rate but it is likely to result in a higher false positive rate. We leave it as future work to try different combinations of complimentary lists [12] on Adblock Plus.

ing useful functionalities on webpages. There are several possible reasons for this ineffectiveness. For instance current PP-Tools use rather elementary techniques such as manually maintained blacklists (whose maintenance is hard amidst the rapid growth of trackers), regular expression matching only on URLs within the `script` tag or even completely blocking the use of JavaScript programs. In our work, we go further by inspecting JavaScript code itself.

5 Classification and Validation

In a real setting, we may only expect to collect a small subset of JavaScript programs known to be functional or tracking due to the sheer prevalence of JavaScript programs on the web. We hypothesise that it is sufficient to have partial knowledge of only the tracking JavaScript class. Our intuition is that tracking JavaScript programs potentially share similar characteristics and these characteristics can be leveraged in a one-class classification framework. In what follows, we first introduce the various models to extract JavaScript code features and then present our machine learning approaches.

5.1 Feature Models

The intuition. Consider the cookie setting code snippets from Google Analytics [10] and Visual Revenue [20] shown as Trackers 1 and 2, respectively. Notice that the two are functionally and structurally similar, with differences in variable names. More technically, the snippets result in similar canonical representations which we shall explain in Section 5.1.2. Similar examples indicate that a similarity measure based on a feature space composed of semantic or syntactic *tokens* from these JavaScript programs should be effective in differentiating between functional and tracking JavaScript programs.

Tracker 1. Google Analytics Cookie Setting

```
var _gaq = _gaq || [];
_gaq.push(['_setAccount', 'UA-1627489-1']);
_gaq.push(['_setDomainName', 'geo.tv']);
_gaq.push(['_trackPageview']);
```

Tracker 2. Visual Revenue Cookie Setting

```
var _vrq = _vrq || [],
_vrqIsOnHP = (document.body.className ||
  '').search('pg-section') >=0 ? true : false;
_vrq.push(['id', 396]);
_vrq.push(['automate', _vrqIsOnHP]);
_vrq.push(['track', function() {}]);
```

To exemplify the existence of such similarity, we create three distinct sets of 500 JavaScript codes (tracking-

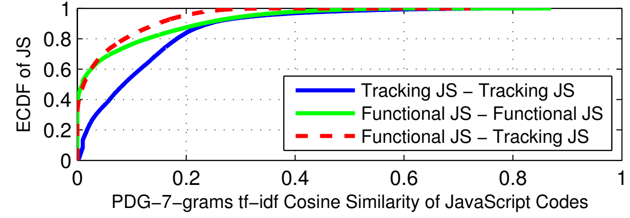


Fig. 3. Similarity between disjoint sets of functional and tracking JavaScript codes (JS).

only, functional-only, and tracking and functional) from our labelled dataset and calculate the *term frequency with inverse document frequency* (tf-idf) based *cosine* similarity values based on our "PDG-7-grams" feature model, to be explained shortly in Section 5.1.2. Note that, being a naïve approach, the cosine similarity metric is shown here for illustrative purposes only. We shall use our classifiers later (Section 5.2) to more efficiently classify tracking and functional JavaScript codes.

Figure 3 plots the cumulative distribution function (CDF) of the similarity values obtained. The long tail we observe for "tracking vs. tracking" and "functional vs. tracking" as compared to "functional vs. functional" suggests that tracking (resp. functional) components do indeed have higher intra-similarity. We provide examples of code similarity for tracking JavaScript codes in [34].

In the rest of this section, we give details of the syntactic and semantic models in use. But first, we introduce some notation.

Notation and Problem Formulation. Let J represent all JavaScript programs in the corpus, and let J' be a small subset of them that are labelled. The labels only belong to tracking JavaScript programs, which we call positive labels. The totality of the negative labels, i.e., functional JavaScript programs, are not known beforehand. The JavaScript programs from $J - J'$ can be either functional or tracking. The goal of the classifier is to obtain a correct labelling of the JavaScript programs in J. Assume that we have a feature vector $\mathbf{j}$ corresponding to some JavaScript code j obtained through one of the feature extraction models, to be discussed shortly. Let y represent the class $\mathbf{j}$ belongs to. If $\mathbf{j}$ is a tracking JavaScript, then $y = +1$; otherwise $y = -1$. Let us also introduce the label flag l. If $\mathbf{j}$ is labelled, i.e., is assigned a class via $y = +1$ or -1, then $l = 1$; otherwise $l = 0$.

To quantitatively measure similarity between different JavaScript programs, we use the tf-idf measure. Central to this measure is the term t. How the term t is defined gives rise to the different semantic and syntactic models to be discussed shortly. For now let us assume

that each JavaScript program $j \in J$ is composed of one or more terms. We then use the boolean term frequency (tf) measure such that $\mathsf{tf}(t, j) = 1$ if $t \in j$ and 0 otherwise. The inverse document frequency measure idf is defined as: $\mathsf{idf}(t, j, J) = \log \frac{|J|}{|\{j \in J : t \in j\}|}$. Finally, we obtain tf with idf for each $t \in j$ over the corpus J, as: $\mathsf{tf\text{-}idf}(t, j, J) = \mathsf{tf}(t, j) \cdot \mathsf{idf}(t, j, J)$. The $\mathsf{tf\text{-}idf}$ thus transforms our corpus J to the feature vector space, where the feature vector of a JavaScript program j is $\mathbf{j}$. The ith component of $\mathbf{j}$, denoted $\mathbf{j}[i]$ corresponds to the term t_i and is equal to $\mathsf{tf\text{-}idf}(t_i, j, J)$.

5.1.1 Syntactic Model

In the syntactic model, the term t is defined as a string of characters representing *one line* of JavaScript code.[5] Due to the large number of JavaScript codes in J the dimension of the feature space needed to be "capped" to avoid making it impractically large. We experimented with different sizes of the feature space by using the top 200, 500, 1,000, 1,500 and 2,500 terms as ranked by their $\mathsf{tf\text{-}idf}$ scores, in turn. We found no considerable improvement in classification over the smallest size, i.e., 200, and since this size was also the most computationally efficient, we chose it as a cap on the feature vector space.

5.1.2 Semantic Models

For a richer model that captures the structure of JavaScript codes, we use the n-gram model introduced in [32], which identifies *tokens* in programs. An example of a token is one line of JavaScript code, as used in the syntactic model. An n-gram, where $n \geq 1$, refers to n tokens which are dependent on each other. To construct n-grams, the approach taken in [32] converts the code into a *canonical form*. The canonical form is an intermediate representation of a JavaScript code where variable names and loop specifics are abstracted away in an effort to apply natural language processing-like techniques on the resulting code representation. From the resultant canonical form, two different models can be derived based on how the n-grams are constructed. The difference lies in how different lines of the canonical code are perceived to be dependent on each other. One model relies on sequential dependency, wherein a line of code is dependent on the execution of the previous line, and so on. Thus, an n-gram of a statement in the canonical code is a sequence of n lines. We call this the *sequential n-gram model*.

Alternatively, since consecutive lines in source code are often unrelated to each other, instead of computing n-grams via this trivial ordering, program dependency graphs (PDGs) can be employed [32, 43]. In such a graph, a node represents a single statement from the canonical form. A node a of the graph is dependent on a node b if its execution depends on the result of b or if it takes as input a value written by b. If this is true, node a is connected to b via a directed edge in the direction of the latter. An n-gram of a particular gram x is then the subgraph of the program dependency graph consisting of all (backward) paths of length $n-1$ starting from x. This represents the second model and is referred to as the *PDG n-gram model*. See Appendix A.2 for an example of canonicalization and the resulting PDG.

Once n-grams have been constructed via the sequential or the PDG model, we use them in our $\mathsf{tf\text{-}idf}$ measure to calculate their relative importance [32], where the term t is an n-gram. We can have different models depending on the value of n in n-gram. For this study, we use 4-gram and 7-gram variants. We tried several values of n in our n-gram models, but found no significant improvement beyond $n = 7$.

5.2 Machine Learning Classifiers

We use two machine learning approaches, *one-class SVM* [51] and *positive and unlabelled (PU) learning* [28, 40] that apply to our problem of binary classification (tracking or functional JavaScript code) with only partially known instances from one of the classes used in training [33]. These two classifiers only require samples from the positive class, i.e., tracking JavaScript programs, in the training phase. One-class SVM (OCSVM) maps the feature vectors belonging to the training data to a higher dimensional space through the use of an appropriate kernel, and then finds the hyperplane whose margin from the origin is maximized. One can view one-class SVM as a regular two-class SVM with the difference that the origin represents the only member of the negative class [51]. The PU learning technique constructs a probabilistic classifier that decides whether a JavaScript code is tracking or functional from a probabilistic classifier that decides whether a JavaScript code is labelled or unlabelled. A brief introduction to PU learning is given in Appendix A.3.

5 Given that many JavaScript codes are compressed by removing whitespaces (sometimes into a single line), we used the `jsbeautifier` Python library to unpack JavaScript codes before applying our feature extraction models.

5.3 Validation

We use the traditional supervised two-class support vector machine (SSVM) [44, 49] as a benchmark for the performance of our one-class classifiers. We run the three classifiers (one-class SVM, PU-Learning and SSVM) fed by syntactic and semantic features extracted from the JavaScript programs in the labelled dataset. For the PDG n-gram model, we construct the program dependency graph from the canonical code by analysing the abstract syntax trees produced by the V8 JavaScript engine. We used the software shared by authors of [32] to transform JavaScript programs into canonical forms and to construct PDGs from the canonical forms. As mentioned before, feature vectors for the syntactic model were constructed by considering the top 200 terms ranked by their tf-idf score. On the other hand, no "cap" was used for the sequential n-gram and PDG n-gram models, as the feature vector size was already around 200.

For PU-learning and one-class SVM, we use 80% of the tracking JavaScript programs from the labelled dataset to constitute the training set (i.e., the training set only contains members of the positive class). We mixed the remaining 20% of tracking JavaScript programs with functional JavaScript programs in the labelled dataset for the testing of these two classifiers. For SSVM, we use 80% and 20% of JavaScript programs (both functional and tracking) from the labelled dataset for training and testing, respectively. We empirically find the appropriate values for γ, a parameter for *radial basis function* kernel [51], and ν, a parameter for SVMs by performing a grid search on the ranges $2^{-15} \leq \gamma \leq 2^{0}$ and $2^{-10} \leq \nu \leq 2^{0}$ with 5-fold cross-validation on each training group. We use `scikit-learn` [48], an open source machine learning library for Python that includes a modified version of LIBSVM [25].

5.3.1 Performance of Classifiers

Table 5 shows the performance of our classifiers. Note that for each feature model, we use the same training set for PU-learning and one-class SVM. We observe that, regardless of the feature model in use, PU-learning and one-class SVM exhibit similar performance. They also perform similar to supervised SVM in terms of false and true negative rates (related to functional JavaScript programs). In general, except for the syntactic feature model where supervised SVM outperforms our one-class classifiers in terms of false and true positives, the three classifiers achieve very similar rates, with true positive and negative rates of up to 0.99 and false positive and negative rates of only 0.01.

Feature Model	Classifier	Tracking		Functional	
		Blocked	Allowed	Blocked	Allowed
Syntactic	SSVM	0.93	0.07	0.01	0.99
	OCSVM	0.88	0.12	0.02	0.98
	PU	0.86	0.14	0.02	0.98
PDG 4-gram	SSVM	0.96	0.04	0.03	0.97
	OCSVM	0.95	0.05	0.03	0.97
	PU	0.93	0.07	0.04	0.96
Sequential 4-gram	SSVM	0.98	0.02	0.01	0.99
	OCSVM	0.98	0.02	0.02	0.98
	PU	0.96	0.04	0.03	0.97
PDG 7-gram	SSVM	0.99	0.01	0.01	0.99
	OCSVM	0.99	0.01	0.01	0.99
	PU	0.98	0.02	0.02	0.98
Sequential 7-gram	SSVM	0.99	0.01	0.01	0.99
	OCSVM	0.99	0.01	0.01	0.99
	PU	0.98	0.02	0.02	0.98

Table 5. Performance of the classifiers against the labelled dataset of tracking and functional JavaScript programs. ■ true positives and negatives, ■ false positives and negatives.

In comparison with the tested PP-Tools (cf. Table 4), this shows an improvement in the true positive rate by 21% to 62% and in the false positive rate by 5% to 20%[6]. Not only do our classifiers outperform the PP-Tools in effectively detecting tracking JavaScript programs, but they also do not suffer from high misclassification of functional JavaScript programs which would result in poor user web experience.

5.3.2 Effect of Feature Models

Table 5 suggests that the feature models have an effect on the classification accuracy. The syntactic model has the worst performance for all three classifiers. The PDG 7-gram and sequential 7-gram models in contrast show the best results for all the classifiers. We improve the false negative rate by 11-12% in the case of the one-class classifiers by using the 7-gram models. Interestingly, the performance of the classifiers for classifying functional JavaScript programs is similar across all feature models, which suggests that perhaps functional JavaScript programs have more inter-similarity than the inter-similarity between tracking JavaScript programs.

Our results show that one-class classifiers, as non-expensive learning techniques, perform similarly. Now we aim to apply our one-class SVM and PU-learning classifiers in the wild and compare their output to PP-Tools. For this purpose, we choose the best and the worst performers of the lot: sequential 7-gram and the syntactic model, respectively. Note that although, per-

6 We reiterate that our comparison is fair since we configured these PP-Tools to be consistent with our rules in Table 1.

formance wise, sequential 7-gram and PDG 7-gram are similar, we chose the former as it requires less preprocessing (no construction of PDGs).

6 Evaluation in the Wild

We evaluate and compare the output of our classifiers and the PP-Tools on a set of JavaScript programs collected from a large number of websites, called the *wild dataset*.

6.1 The Wild Dataset

The wild dataset consists of JavaScript programs extracted from the landing page of 4,084 websites. These websites were selected such that 3,500 of them were top Alexa websites ranked between 51 to 3,550 inclusive (since the top 50 websites were already used in the labelled dataset), with the remaining 584 having a rank in excess of 5,000 (excluding the 45 websites with rank in this range used in the labelled dataset (Section 3.3)). A total of 135,656 JavaScript programs were present in these websites. The composition of JavaScript programs into external and in-page JavaScript programs is shown in Table 6 (under the column labelled PP-Tools *off*). The table also shows the number of in-page and external JavaScript programs allowed by each PP-Tool considered in our work. Again, we observe that NoScript is the most aggressive tool, blocking on average 37.7% of the JavaScript programs per website, closely followed by Ghostery (34.8%) and AdBlock Plus (26%). In comparison, Privacy Badger and Disconnect block 11.5% of JavaScript programs per website.

	PP-Tools	PP-Tools *on*				
JS	*off*	NS	GT	AP	DC	PB
External	71,582	29,345	38,492	48,191	59,488	60,817
In-page	64,074	54,972	49,952	51,389	60,546	59,250
Total	135,656	84,428	88,444	99,580	120,034	120,067
Average per webpage	33.2	20.7	21.7	24.4	29.4	29.4
Blocked (%)	-	37.7	34.8	26.6	11.5	11.5
Allowed (%)	-	63.3	65.2	73.4	88.5	88.5

Table 6. Characteristics of JavaScript programs (JS) collected from 4,084 websites with PP-Tools *on* and *off* as viewed from a Firefox Selenium-controlled browser.

6.2 Comparing PP-Tools and our Classifiers

We trained our PU-Learning and one-class SVM classifiers using only the tracking JavaScript programs from the labelled dataset. The trained classifiers were then run on the wild dataset. Next, we assess the extent to which our two classifiers agree or disagree with each PP-Tool.

6.2.1 Agreement Ratio

Denote by T_c and F_c the set of tracking and functional JavaScript programs, respectively, in the wild dataset as classified by a classifier c (i.e., one-class SVM or PU learning). Likewise, T_p, resp. F_p, denotes tracking, resp. functional, JavaScript programs as marked by the PP-Tool p. Then, the ratio of agreement between c and p on tracking JavaScript programs is $|T_c \cap T_p|/|J|$, where J is the set of JavaScript programs in the wild dataset. Similarly, the ratio of agreement on the functional JavaScript programs is $|F_c \cap F_p|/|J|$. The ratio of agreement between a classifier c and a PP-Tool p is simply $(|T_c \cap T_p| + |F_c \cap F_p|)/|J|$. To avoid excessive notation, we shall denote the ratios simply by their constituent sets. For instance, the agreement ratio for tracking JavaScript programs will simply be written as $T_c \cap T_p$. Table 7 lists the values of agreement and the corresponding disagreement ratios (defined as 1 minus the agreement ratio). Figure 7 in Appendix A.4 illustrates these ratios graphically.

6.2.2 Performance of Classifiers

From Table 7, we first note that the two classifiers are similar in terms of their agreement/disagreement with PP-Tools, yielding very high disagreement ratio in the syntactic model (ranging from 39% to 58%) and significant disagreement in the sequential 7-gram model (20% to 64%). Overall, the two classifiers are relatively more in agreement with NoScript, followed closely by Ghostery and Adblock Plus. This agreement, however, is mostly in terms of classifying tracking JavaScript programs.

In terms of agreement on functional JavaScript programs, the two classifiers are more in tune with Disconnect and Privacy Badger. These two PP-Tools however are the least in agreement with our classifiers in terms of tracking JavaScript programs (ranging between 13% and 27%). One possible explanation for this, is that Disconnect mostly blocks social plugins and buttons such as Facebook Likes and Twitter Follow. This leaves a host of other trackers allowed.

Another possible explanation for the disagreements with PP-Tools is related to R4 in Table 1. If an In-page JavaScript code that does not write or read cookies but loads a tracking JavaScript code, PP-Tools may block only the tracking JavaScript code to effectively prevent tracking. However, following R4 (cf. Table 1), our mechanism classify them as tracking JavaScript codes. Moreover, the results for Privacy Badger are similar because it mainly blocks JavaScript codes that track users across

Feature Model	Classifier	PP-Tool	$T_c \cap T_p$	$T_c \cap F_p$	$F_c \cap T_p$	$F_c \cap F_p$	Agreement	Disagreement
Syntactic	OCSVM	NoScript	0.56	0.10	0.29	0.05	0.61	0.39
		Ghostery	0.54	0.13	0.27	0.06	0.60	0.40
		Adblock Plus	0.47	0.20	0.25	0.09	0.56	0.44
		Privacy Badger	0.23	0.44	0.11	0.22	0.45	0.55
		Disconnect	0.17	0.50	0.08	0.25	0.42	0.58
Sequential 7-gram	OCSVM	NoScript	0.71	0.06	0.14	0.09	0.80	0.20
		Ghostery	0.67	0.10	0.15	0.08	0.75	0.25
		Adblock Plus	0.62	0.15	0.11	0.13	0.75	0.25
		Privacy Badger	0.27	0.5	0.07	0.16	0.43	0.57
		Disconnect	0.19	0.58	0.06	0.17	0.36	0.64
Syntactic	PU	NoScript	0.50	0.07	0.36	0.07	0.57	0.43
		Ghostery	0.47	0.10	0.35	0.08	0.55	0.45
		Adblock Plus	0.43	0.14	0.30	0.13	0.56	0.44
		Privacy Badger	0.18	0.38	0.15	0.28	0.46	0.54
		Disconnect	0.13	0.44	0.12	0.31	0.44	0.56
Sequential 7-gram	PU	NoScript	0.70	0.05	0.16	0.09	0.79	0.21
		Ghostery	0.65	0.10	0.16	0.09	0.74	0.26
		Adblock Plus	0.61	0.14	0.12	0.13	0.74	0.26
		Privacy Badger	0.18	0.57	0.07	0.18	0.36	0.64
		Disconnect	0.26	0.49	0.07	0.18	0.44	0.56

Table 7. Agreement and disagreement in classification of tracking and functional JavaScript programs between our classifiers and PP-Tools on the wild dataset; agreement, disagreement; T_p and F_p represent JavaScript programs classified as tracking and functional, respectively, by the PP-Tool p, and T_c and F_c represent JavaScript programs classified as tracking and functional, respectively, by the classifier c.

multiple websites through cookies, which is routinely done by social widgets.

6.2.3 Effect of Feature Models

Table 7 also shows that the sequential 7-gram model of the two classifiers is more in agreement with NoScript, Ghostery and Adblock Plus as compared to the syntactic model, by around 20%. However, the difference is nominal for Disconnect and Privacy Badger, with our classifiers agreeing more with these two PP-Tools in the syntactic model. In the following, we further analyse the observed disagreement by using the sequential 7-gram model of the one-class SVM as it showed superior results during our validation experiments (Section 5.3).

6.3 Analysing Disagreements

We delve into the set of JavaScript programs on which *all* PP-Tools and our one-class SVM with sequential 7-grams classifier disagree. We are interested in the two facets of disagreement: JavaScript programs that our classifier considers tracking but all the PP-Tools consider functional, i.e., the set $T_c \cap_p F_p$, and JavaScript programs that our classifier deems tracking while all PP-Tools consider functional, i.e., $F_c \cap_p T_p$. Note that subscripts c and p stand for our classifier and a PP-Tool, respectively.

The number of JavaScript programs for which our classifier and *all* PP-Tools are in disagreement is 9,071, representing 6% of the total number of JavaScript programs in the wild dataset. These JavaScript programs are split as 4,610 for $T_c \cap_p F_p$ and 4,461 for $F_c \cap_p T_p$. Inspecting these sets of disagreement would shed light on the main reasons for disagreement. Unfortunately, manually inspecting thousands of JavaScript programs (using the process used for producing our labelled dataset) is a tedious and time consuming process. We instead took a pragmatic approach, where we randomly sampled 100 JavaScript programs each from the two sets of disagreement. We then manually inspected each JavaScript from the two samples and classified them as tracking or functional following the rules and methodology described in Section 3.3. The results of this manual process are shown in Table 8.

Our classifier is correct in its labelling of 75 out of the 100 JavaScript programs it considered tracking. All these JavaScript programs are marked as functional by all the PP-Tools, implying that the PP-Tools are correct in labelling only 25 of these JavaScript programs. Moreover, our classifier correctly deemed 81 out of the 100 JavaScript programs as functional, implying that the PP-Tools correctly labelled only 19 of the JavaScript programs in the random sample. Note that these numbers should not be taken as reflecting the overall classification performance of our classifier, which was validated in Section 5.3. These samples merely represent the cor-

Disagreement	Total	Sample	Manual Labelling	
			Tracking	Functional
$T_c \cap_p F_p$	4,610	100	75	25
$F_c \cap_p T_p$	4,461	100	19	81

Table 8. Comparison of random samples of disagreement between our classifier and *All* PP-Tools. ■ manual labelling agrees with classifier and disagrees with PP-Tools, ■ manual labelling disagrees with classifier and agrees with PP-Tools.

ner cases of complete disagreement with *all* other PP-tools. In other words, these numbers do not directly give us the true and false positive rates of our classification methodology.

We first look at the 75 JavaScript programs correctly labelled as tracking by our classifier, and incorrectly considered as functional by the PP-Tools. Table 10 in Appendix A.4 shows 10 representative JavaScript programs from this sample. We identify two typical reasons the PP-Tools miss blocking these JavaScript programs:

1. PP-Tools do not perform regular expression matching on the body of JavaScript programs to identify known trackers. Examples from these "misses" are JavaScript programs #2 and #3 in Table 10, which are allowed by all PP-Tools even though the referred domain `doubleclick.net` is included in their blacklists; this is because these JavaScript programs refer to this domain in their body, and the PP-Tools perform a regular expression match only on the URL of the JavaScript within the `script` tag.
2. As expected PP-Tools are unable to block trackers that are not in the blacklist. An example is JavaScript #9 in the table which we manually check to be a social widget allowing users to 'like' comments on the webpage while tracking the user activity which is then transmitted to the first party domain. All PP-Tools miss this JavaScript because it does not belong to a popular social media domain. Similarly, JavaScript #4 in the table belongs to a Russian tracking and advertising service domain `i-vengo.com`, but it is not in the blacklists of PP-Tools.

Our classifier correctly marked these JavaScript programs as tracking as these scripts were syntactically and structurally similar to the tracking JavaScript programs used for training our classifiers. We stress that our classifiers do not need to know about all tracking scripts a priori; in fact, our classifiers are able to find new tracking scripts leveraging the syntactic and semantic similarity between known tracking scripts and previously unknown tracking JavaScript programs.

Next, we look at the 81 JavaScript programs correctly labelled as functional by our classifier, and incorrectly considered as tracking by the PP-Tools. Table 11 in Appendix A.4 shows 10 JavaScript programs from this sample. The predominant reason for the PP-Tools mistakenly blocking these JavaScript programs is because they belong to a tracking domain, even though the JavaScript itself performs a useful functionality. A typical example is JavaScript #10 in the table, which fetches content from the first party domain `buzzfeed.com` without sending or collecting user information.

Lastly, we believe that the main reason our classifier misclassified 25 functional JavaScript programs and 19 tracking JavaScript programs is due to their structural similarity with representatives of the opposite class. For instance, the JavaScript `jquery.cookie.js` in the website `pnc.com` modifies cookies for this non-tracking domain. The PP-Tools rightly allow this JavaScript because `pnc.com` is not a tracking domain. But, due to the structural similarity of this JavaScript with JavaScript programs that modify cookies for tracking domains, our classifier deemed it as tracking. Similarly, our classifier misclassified the JavaScript `count.js` that gathers comment statistics on the website `listverse.com` and sends this information to the domain `disqus.com`, which is listed as a tracker by the PP-Tools. Our classifier misread this due to its similarity with JavaScript programs that maintain comments on a webpage but do not send this information through to third party trackers. For brevity, we do not enlist samples of these two categories of JavaScript programs misclassified by our classifier.

7 Discussion

In the following, we discuss possible uses and limitations of our approach.

7.1 Possible Uses

We envision at least two different uses of our technique:

7.1.1 Browser Extension

A natural application of our technique is a client-based browser extension to evade trackers. We are currently developing a Firefox browser extension which extracts the JavaScript programs while a webpage is being loaded (prior to rendering) and calculates the similarities of the observed JavaScript programs against the training model which is kept locally. As discussed later in this section, we aim to periodically update the training model using a semi-supervised learning technique [23]. We believe that such an extension is practical, as our current system classifies in the order of milliseconds per website. Development of the browser extension

would also enable us to directly compare the actual performance of our system in terms of time and memory with existing PP-Tools.

7.1.2 Updating Blacklists and Whitelists

Another possible use of our technique is to improve the accuracy of existing PP-Tools by updating their filtering lists. These tools could submit sets of JavaScript programs embedded in webpages (randomly chosen via a web scanner) to the classifier which would identify them as functional or tracking. The domains corresponding to the URLs linking these JavaScript programs can then be deemed tracking or tracking-free. The newly identified URLs can be used to update the PP-tool's blacklists (generally locally stored on the user browser) or to refine whitelists used by some tools such as NoScript. However, it is important to note that the generated URLs might lead to errors, potential false positives and negatives, as the same domain may produce both functional and tracking JavaScript programs, and thus the decision to mark the domain tracking or tracking-free lacks the contextual information used by the classifier. One trivial example is `http://static.bbci.co.uk` domain which can be observed in both functional and tracking JavaScript programs. Another important caveat is that the blacklist based approach inherently does not filter in-page JavaScript programs, and as such the PP-Tools considered in this paper do not block their execution, with the exception of NoScript (which, as we have seen, aggressively blocks execution of JavaScript codes).

7.2 Limitations and Possible Improvements

7.2.1 Classification Arms Race - Feature-exploit Case

In principle, machine learning based detection is prone to exploits that introduce features absent from the training set [27]. In our case, the tracker could introduce some unique or rare piece of tracking code in the JavaScript code. Due to the uniqueness of the resulting JavaScript code, its feature set is unlikely to be present in the training model, and is therefore likely to go undetected. This can particularly be the case with non-pervasive "handcrafted" trackers. However, as these pieces of code become ubiquitous and with periodic re-training of the classifier, this exploit can be circumvented.

7.2.2 Classification Arms Race - Code Obfuscation Case

A tracker might also evade detection by obfuscating a tracking JavaScript code either by renaming it or by making changes to its code. In the first case, since our approach is based on code similarity of JavaScript codes, renaming does not affect the efficiency of our classifiers (unlike blacklist based PP-Tools). In the latter case, the attacker might *(i)* rename function or variable names, *(ii)* add or remove whitespaces, *(iii)* add zero-impact or dummy code, *(iv)* or encode the JavaScript code [54]. In the semantic feature models, our classifiers are resilient against the first two types of JavaScript code obfuscation strategies. We illustrate this via an example in Appendix A.6. On the other hand, the attacker might evade detection by applying the last two types of JavaScript code obfuscation techniques. We believe that this presents the classical arms race issue which we elaborate (Appendix A.6), as a limitation of our work.

7.2.3 The manual labelling challenge

In this work, we used a relatively small set of labelled tracking and functional JavaScript programs (2,612 to be precise). Our choice was dictated largely by the time consuming nature of the labelling process. Obviously, the performance of the classifier can be improved by increasing this number. One approach is to rely on crowdsourcing and recruit "tech-savvy" users as reviewers of a JavaScript code. This is not a trivial task as it requires considerable effort in providing guidelines (i.e., Table 1) to the reviewers, a platform for interaction, and the need to resolve conflicts in labelling due to variable technical expertise of reviewers.

Alternatively, hybrid schemes such as semi-supervised learning [23] can be used. The basic idea behind semi-supervised learning is to use unlabelled data to improve the training model that has been previously built using only labelled data. Semi-supervised learning has previously been successfully applied to real-time traffic classification problems [29], showing that automated identification of re-training points is possible. Note that our model does not have to be re-trained as frequently as updating the blacklists of PP-Tools. Blacklists need to be frequently updated so that PP-Tools can keep track of new tracker URLs. In our case, re-training is only required in case new tracking JavaScript codes with unconventional code structure emerge. This is expected to occur far less frequently.

8 Related Work

In recent years, there has been much research on privacy implications of web tracking [22, 30, 35–38, 45, 52, 53]. For a recent and comprehensive survey on web tracking techniques and tracking prevention policies, see [41].

Specific to automated detection of trackers, the work of Gugelmann et al. [31] is related to ours in the

sense that they use a machine-learning approach to detect tracking JavaScript codes. However, there are fundamental differences: (1) they use HTTP traffic traces instead of code analysis (which is done in our case). The resulting features are therefore traffic-based features as opposed to our approach where we use JavaScript syntax and structure based features. The features are extracted by inspecting HTTP requests sent from the users' browsers. The HTTP request statistics can be hard to gather and can be privacy intrusive themselves as is acknowledged by the authors themselves [31]; (2) Gugelmann et al.'s approach requires collaboration between different network nodes and services to gather traffic traces to apply machine learning algorithms on them (also, once again, raising privacy issues). In comparison, our approach can be used via a client-side browser extension where the machine learning algorithms are applied on the webpages directly. Thus, our approach relies on data treated locally from one node only (the client's browser), whereas the approach in [31] relies on data as monitored and accumulated from the network; (3) Finally, their approach is focused on aiding and complementing a list of trackers from the blacklist of Adblock Plus, whereas our main objective is to build a standalone detection of tracking entities.

Similarly, the closest work to ours is by Orr et al. [47] who proposed a machine learning approach to detect JavaScript-based advertisements. However, [47] detects only ad-related JavaScript codes, while we target and identify a wider class of tracking JavaScript codes, which on top of advertisements, includes analytics, cookie readers/writers, social media widgets, etc. In addition, Orr et al. use static code analysis of ad related JavaScript codes to come up with a set of 20 features to discriminate between ad related and non-ad related JavaScript codes. Since we consider the class of trackers to be much more diverse, static code analysis of such a diverse class is not only infeasible, but it might not result in a common set of features for all different classes of trackers. Therefore, the approach used in our paper is to generate dynamic features from the code syntax and structure of JavaScript codes through a tf-idf approach. This approach is fundamentally different from static code analysis, as these features are based on syntax and structure rather than manual extraction of features, which makes our approach more feasible and generic. Finally, Orr et al. do not evaluate their classifiers in the wild and it is unclear how the classifier performs against PP-Tools that specifically block advertisements. Hence, it is difficult to determine the advantage of their approach over ad-blocking PP-Tools. In our paper, we compare the performance of our classifiers against existing PP-Tools and show classification performance improvement.

Tran et al. [52] use tainting for dynamic analysis of JavaScript execution to assess private information leakage. They end up detecting *one* tracking website in addition to the services listed by Ghostery. In contrast, our framework reveals more than 4,000 new JavaScript codes showing tracker-like behaviour, and based on a representative sample we estimate that 75% indeed offer traditional advertisement and analytics services.

In [32], authors focus on plagiarism detection of codes. We inherit the n-grams technique from [32] to identify similarity between functional and tracking JavaScript codes. However, for the purpose of our classification problem, we need also to distinguish between two classes of programs, functional and tracking, for which we use more specialized approaches: one-class SVM and PU learning.

Lastly, we remark that our work on the effectiveness of PP-Tools is complementary to prior work on their performance aspects, such as [39] which focus on the user-friendliness of PP-Tools.

9 Concluding Remarks

This paper presented a new automated mechanism to effectively filter tracking JavaScript programs from functional JavaScript programs, both embedded and externally linked, within the DOM trees of webpages. We first study the (in)effectiveness of popular privacy-preserving tools with respect to balancing blocking of tracking JavaScript programs and allowing functional JavaScript programs. We then postulate and verify that one-class machine learning techniques that utilize similarities between tracking JavaScript programs based on syntactic and semantic features can effectively improve user's web experience. One key aspect of our work is the ability of our classifiers to discover previously unseen tracking JavaScript programs. We stress that our methodology is generic and can be adapted to more conservative choices of what are considered functional JavaScript codes (e.g., JavaScript codes related to social media). Our current labelling is strict on what is considered tracking. Our classifiers can easily be tuned by simply adding or removing few instances of JavaScript programs from the tracking set used in training. In the future, we aim a characterizing JavaScript programs' obfuscation techniques to improve the resiliency of our proposed scheme. Finally, we also aim to evaluate our scheme against browser fingerprinting JavaScript programs.

References

[1] Adblock Plus. https://www.adblockplus.org.
[2] difflib – Helpers for computing deltas. Python 2.7.11 documentation, https://docs.python.org/2.7/library/difflib.html.
[3] Disconnect. https://www.disconnect.me/.
[4] Disqus. http://www.disqus.com/.
[5] EasyList. https://easylist-downloads.adblockplus.org/easylist.txt.
[6] Easyprivacy. https://easylist.to/easylist/easyprivacy.txt.
[7] EFF DNT Policy. https://www.eff.org/dnt-policy.
[8] Ghostery. https://www.ghostery.com.
[9] Ghostery 5.4.7 is ready to run! https://www.ghostery.com/intelligence/consumer-blog/product-releases/ghostery-547-is-ready-to-run/.
[10] Google Analytics. https://www.google.com/analytics/.
[11] Javascript Obfuscator. Dan's tools, http://www.danstools.com/javascript-obfuscate/.
[12] Known Adblock Plus Subscriptions. https://adblockplus.org/subscriptions.
[13] Moat. http://www.moat.com/.
[14] NoScript. https://www.noscript.net.
[15] PFanboyList. https://easylist-downloads.adblockplus.org/fanboy-social.txt.
[16] Privacy Badger Firefox. https://github.com/EFForg/privacybadgerfirefox/blob/master/data/cookieblocklist.txt.
[17] Real-Time Digital Advertising That Works | Criteo. http://www.criteo.com.
[18] Script Surrogates Quick Reference. https://hackademix.net/2011/09/29/script-surrogates-quick-reference/.
[19] Search Results in Ghostery - 381 Topics Found for Breaking. https://getsatisfaction.com/ghostery/searches?query=breaking&x=15&y=10&style=topics.
[20] Visual Revenue. http://www.visualrevenue.com.
[21] What is Selenium? http://www.seleniumhq.org/.
[22] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. CCS, 2014.
[23] S. Basu, M. Bilenko, and R. J. Mooney. A Probabilistic Framework for Semi-supervised Clustering. KDD, 2004.
[24] A. Chaabane, M. A. Kaafar, and R. Boreli. Big Friend is Watching You: Analyzing Online Social Networks Tracking Capabilities. WOSN, 2012.
[25] C.-C. Chang and C.-J. Lin. LIBSVM: A Library for Support Vector Machines. *ACM ToIST*, 2011.
[26] N. Corporation. January 2013: Top U.S. Entertainment Sites and Web Brands. Visited 23 May 2016, http://www.nielsen.com/us/en/insights/news/2013/january-2013--top-u-s--entertainment-sites-and-web-brands.html, 2013.
[27] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. ZOZZLE: Fast and Precise In-browser JavaScript Malware Detection. USENIX SEC, 2011.
[28] C. Elkan and K. Noto. Learning Classifiers from Only Positive and Unlabeled Data. KDD, 2008.
[29] J. Erman, A. Mahanti, M. Arlitt, I. Cohen, and C. Williamson. Offline/Realtime Traffic Classification Using Semi-supervised Learning. *Perform. Eval.*, 2007.
[30] P. Gill, V. Erramilli, A. Chaintreau, B. Krishnamurthy, K. Papagiannaki, and P. Rodriguez. Follow the Money: Understanding Economics of Online Aggregation and Advertising. IMC, 2013.
[31] D. Gugelmann, B. Ager, and V. Lenders. An Automated Approach for Complementing Ad Blockers' Blacklists. PETs, 2015.
[32] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy. Using Web Corpus Statistics for Program Analysis. OOPSLA, 2014.
[33] M. Ikram, H. Asghar, M. A. Kaafar, and A. Mahanti. On the Intrusiveness of JavaScript on the Web. CoNEXT, Student Workshop, 2014.
[34] M. Ikram, H. J. Asghar, M. A. Kaafar, B. Krishnamurthy, and A. Mahanti. Towards seamless tracking-free web: Improved detection of trackers via one-class learning. *arXiv (pre-print), doi: 603.06289*.
[35] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An Empirical Study of Privacy-violating Information Flows in JavaScript Web Applications. CCS, 2010.
[36] B. Krishnamurthy. I Know What You Will Do Next Summer. *SIGCOMM CCR*, 2010.
[37] B. Krishnamurthy, D. Malandrino, and C. E. Wills. Measuring Privacy Loss and the Impact of Privacy Protection in Web Browsing. SOUPS, 2007.
[38] B. Krishnamurthy, K. Naryshkin, and C. Wills. Privacy leakage vs. Protection measures: The Growing Disconnect. W2SP, 2011.
[39] P. Leon, B. Ur, R. Shay, Y. Wang, R. Balebako, and L. Cranor. Why Johnny Can'T Opt out: A Usability Evaluation of Tools to Limit Online Behavioral Advertising. In *SIGCHI*, 2012.
[40] B. Liu, Y. Dai, X. Li, W. S. Lee, and P. S. Yu. Building Text Classifiers Using Positive and Unlabeled Examples. ICDM, 2003.
[41] J. R. Mayer and J. C. Mitchell. Third-Party Web Tracking: Policy and Technology. IEEE S&P, 2012.
[42] H. Metwalley, S. Traverso, and M. Mellia. The Online Tracking Horde: a View from Passive Measurements. TMA, 2015.
[43] S. S. Muchnick. *Advanced Compiler Design and Implementation*. MK Publishers Inc., 1997.
[44] K. R. Muller, S. Mika, G. Ratsch, K. Tsuda, and B. Scholkopf. An Introduction to Kernel-based Learning Algorithms. *ToNN*, 2001.
[45] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless Monster: Exploring the Ecosystem of Web-Based Device Fingerprinting. IEEE S&P, 2013.
[46] L. Olejnik, C. Castelluccia, and A. Janc. Why Johnny Can't Browse in Peace: On the Uniqueness of Web Browsing History Patterns. HotPETs, 2012.
[47] C. R. Orr, A. Chauhan, M. Gupta, C. J. Frisz, and C. W. Dunn. An Approach for Identifying JavaScript-loaded Advertisements Through Static Program Analysis. WPES, 2012.
[48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.*, 2011.
[49] Rieck, Konrad and Krueger, Tammo and Dewald, Andreas. Cujo: Efficient detection and prevention of drive-by-download attacks. ACSAC, 2010.

PP-Tool	Filtering Method	Setting
NS (v2.6.9.11)	Block all JS	Default blocking mode with iframes blocking option on
GT (v5.4.1)	Blacklist	Enabled 'Blocking all tracker and cookies' option
AP (v2.6.7)	Blacklist	*EasyList* and *Fanboy's* list and disabled 'Allow non-intrusive ads' option
DC (v3.14.0)	Blacklist	Default
PB (v0.1.4)	Heuristics and cookies blacklist	Default

Table 9. PP-Tools' settings used with Firefox v32.0.

[50] F. Roesner, T. Kohno, and D. Wetherall. Detecting and Defending Against Third-party Tracking on the Web. NSDI, 2012.

[51] B. Schölkopf, J. C. Platt, J. C. Shawe-Taylor, A. J. Smola, and R. C. Williamson. Estimating the Support of a High-Dimensional Distribution. *NC*, 2001.

[52] M. Tran, X. Dong, Z. Liang, and X. Jiang. Tracking the Trackers: Fast and Scalable Dynamic Analysis of Web Content for Privacy Violations. ACNS, 2012.

[53] C. Wills and D. Uzunoglu. What Ad Blockers Are (and Are Not) Doing. WPI-CS-TR-16-02, 2016.

[54] W. Xu, F. Zhang, and S. Zhu. The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study. MALWARE, 2012.

[55] T.-F. Yen, Y. Xie, F. Yu, R. P. Yu, and M. Abadi. Host Fingerprinting and Tracking on the Web: Privacy and Security Implications. NDSS, 2012.

[56] C. Yue and H. Wang. Characterizing Insecure Javascript Practices on the Web. WWW, 2009.

A Appendix

A.1 PP-Tools Settings And Trackers Distribution in Heat-up Webpages

Table 9 presents the PP-Tools and their settings used in our experiments to collect the datasets as discussed in Section 3.2. Figure 4 and Figure 5 show the type and distribution of trackers, respectively, in heat-up webpages as explained in Section 4.2.

A.2 Canonicalization of JavaScript programs

The canonical form is an intermediate representation of the JavaScript program where variable names and loop specifics are abstracted away in an effort to apply natural language processing-like techniques on the resulting code representation. To explain the canonical form and PDG of a JavaScript program, consider the following (toy) 'equalTest' Javascript function:

Listing 1. An Example of JavaScript Program

```
function equalTest(a, b){
  if(a == b){
    return true;}
  return false;}
```

The canonical form of this routine is:

Listing 2. Canonical form of JavaScript code in Listing # 3

```
function equalTest(a, b){
1:     begin;
2:     $0 = a === b;
3:     if($0){
4:       return true;}
5:     return false;
6:     end;}
```

One line of the canonical form consists of a binary operation, its operands or an assignment. The PDG of this routine is shown in Figure 6a. The 2 and 3-grams of line 3 above is shown in Figure 6b. For a more detailed example of these concepts, see [32].

A.3 Positive and Unlabelled (PU) Learning

The PU learning technique, translated to our problem space, constructs a probabilistic classifier that decides whether a JavaScript code is tracking or functional from a probabilistic classifier that decides whether a JavaScript program is labelled or unlabelled. More precisely, it constructs the classifier $f(\mathbf{j}) = \Pr[y = +1 \,|\, \mathbf{j}]$ from the classifier $g(\mathbf{j}) = \Pr[l = 1 \,|\, \mathbf{j}]$. The two classes in $g(\mathbf{j})$ are the labelled and unlabelled JavaScript programs, whereas the two classes in $f(\mathbf{j})$ are positive (tracking) and negative (functional) JavaScript programs. To understand the concept behind PU learning, notice that the assumptions (a) only positive examples (tracking JavaScript programs) are labelled, (b) the set of labelled examples is chosen uniformly at random from all positive examples, lead to the result $\Pr[l = 1 \,|\, \mathbf{j}, y = -1] = 0$ and $\Pr[l = 1 \,|\, \mathbf{j}, y = +1] = \Pr[l = 1 \,|\, y = +1]$. The probability $\Pr[l = 1 \,|\, y = +1]$ is the constant probability that a positive example is labelled (as it is independent of $\mathbf{j}$). Now we can have $\Pr[l = 1 \,|\, \mathbf{j}]$ as

$$\begin{aligned}\Pr[l = 1 \,|\, \mathbf{j}] &= \Pr[l = 1 \,|\, \mathbf{j}, y = -1] \Pr[y = -1 \,|\, \mathbf{j}] \\ &\quad + \Pr[l = 1 \,|\, \mathbf{j}, y = +1] \Pr[y = +1 \,|\, \mathbf{j}] \\ &= \Pr[l = 1 \,|\, y = +1] \Pr[y = +1 \,|\, \mathbf{j}].\end{aligned}$$

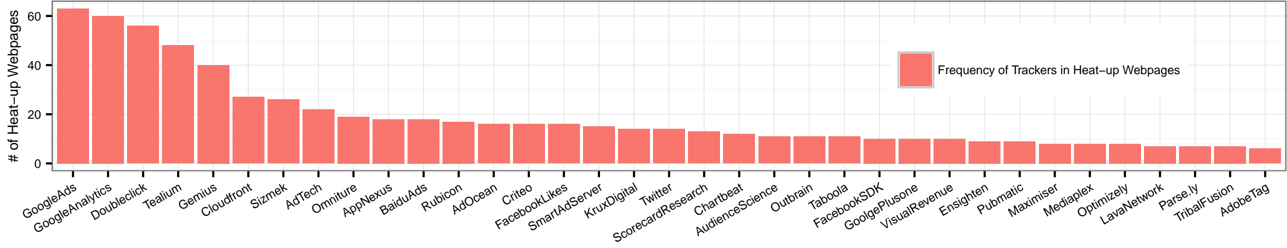


Fig. 4. Top 35 trackers (x-axis) in heat-up webpages as discribed in Section 4.2

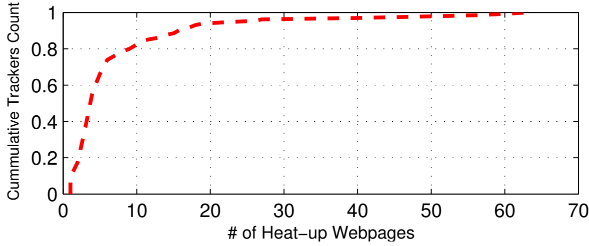


Fig. 5. Distribution of trackers in heat-up webpages as discribed in Section 4.2

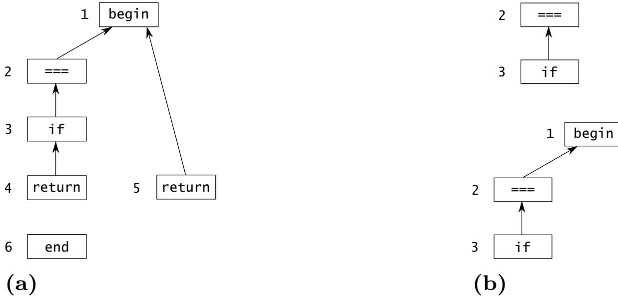


Fig. 6. (a) Program dependency graph of the `equalTest` canonical form. (b) 2-gram (top) and 3-gram (bottom) of line 3 of the program dependency graph.

As $\Pr[l = 1 \,|\, y = +1]$ is a constant, we get the classifier $f(\mathbf{j})$ from $g(\mathbf{j})$. We need only to estimate $\Pr[l = 1 \,|\, y = +1]$. For this, a validation set consisting of only labelled examples, say P, can be used. Note that, according to the assumption above, the labelled examples are all positive. Therefore, in the above equation, for $\mathbf{j} \in P$, the term $\Pr[y = +1 \,|\, \mathbf{j}]$ is 1. This means that $g(\mathbf{j})$ is equal to the constant $\Pr[l = 1 \,|\, y = +1]$ for the validation set P. Thus, we can use the *trained* classifier $g(\mathbf{j})$ on the validation set P to estimate this constant probability by $\frac{1}{|P|} \sum_{\mathbf{j} \in P} g(\mathbf{j})$. In this work, we choose SVM as the trained classifier, i.e. $g(\mathbf{j})$. The reader can refer to [28, 40] for a more comprehensive treatment on PU learning.

A.4 Analyzing Disagreement : Examples

Examples where there is disagreement between our classifier and PP-Tools are shown in Table 10 and Table 11. In essence, we show examples of JavaScript codes that are classified as tracking JavaScript codes (resp. functional) while classified as functional (resp. tracking) by the ensemble of PP-Tools. We discuss these details in Section 6.3. For reference, the ratio of agreement and disagreement between our classifier and PP-Tools on the wild dataset is also illustrated in Figure 7.

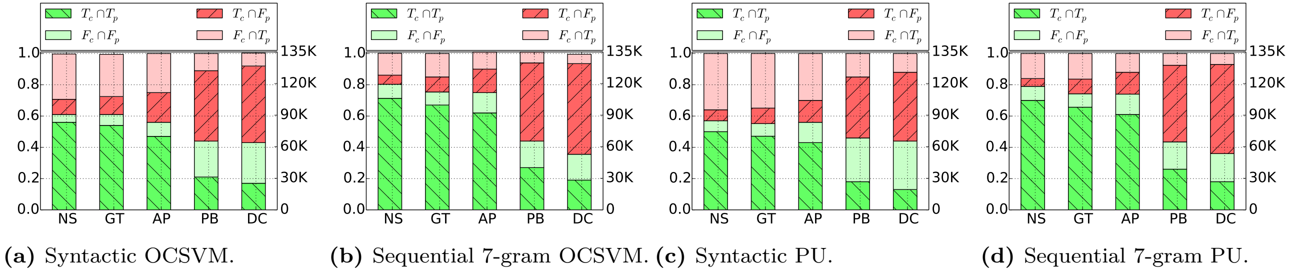
A.5 Surrogate JavaScript Programs

While using PP-Tools, certain content might not be working properly [19]. This is known as *broken webpages*. This happens when certain web-components are blocked on a website which might be necessary for smooth browsing. In order to tackle broken pages, exceptions and errors, PP-Tools often inject snippets of non-tracking JavaScript programs, also called *surrogate* scripts, when they block content from loading. Through manual inspection, we observed that both Ghostery and NoScript inject surrogate scripts. We investigated Ghostery and NoScript source codes to derive a comprehensive list of surrogates.

Interestingly, however, we noticed that using its 'block all trackers' setting, certain Ghostery surrogate scripts do not necessarily facilitate smooth browsing. Instead, they block useful content. For instance, Figure 8 shows an example where Ghostery injects a surrogate script for the `brightcove` widget. The resulting surrogate script blocks the video content on the webpage, which is arguably a useful functionality.

A.6 JavaScript Code Obfuscation

In this section, we further illuminate on the resiliency of our classifier against JavaScript code obfuscation



(a) Syntactic OCSVM. **(b)** Sequential 7-gram OCSVM. **(c)** Syntactic PU. **(d)** Sequential 7-gram PU.

Fig. 7. Agreement (■ & ■) and disagreement (■ & ■) in classification of tracking and functional JavaScript programs between our classifiers and PP-Tools on the wild dataset. T_p and F_p represent JavaScript programs classified as tracking and functional, respectively, by the PP-Tool p, and T_c and F_c represent JavaScript programs classified as tracking and functional, respectively, by the classifier c; NS, GT, AP, PB, and DC stand for NoScript, Ghostery, Adblock Plus, Privacy Badger, and Disconnect, respectively.

#	Website	JavaScript Program	Referred Domain	Function Performed
1	examiner.com	cdn2-b.examiner.com/.../ex_omniture/s_code.js	omniture.com	Analytics
2	bbc.com	static.bbci.co.uk/bbcdotcom/.../adverts.js	pubads.g.doubleclick.net	Analytics + Ads
3	telegraph.co.uk	telegraph.co.uk/template/ver1-0/js/gpt.js	pubads.g.doubleclick.net	Analytics + Ads
4	vesti.ru	s.i-vengo.com/js/ivengo.min.js	www.i-vengo.com	Analytics + Ads
5	climatempo.com.br	http://s1.trrsf.com/metrics/inc/br/201411250000d.js	scorecardresearch.com	Analytics
6	amc.com	amc.com/wp-content/plugins/amcn-common-analytics/js/common-analytics.js	omniture.com	Track user activities
7	lancer.com	static.lancers.jp/js/ga_social_tracking.js	google.com	Tracker user activities
8	iqiyi.com	static.iqiyi.com/js/pingback/qa.js	pps.tv, baidu.com, 71.com	Tracker user activities
9	babyblog.ru	act.babyblog.ru/static844/likes.js	babyblog.ru	Social widgets
10	autoscout.de	s.autoscout24.net/unifiedtracking/gtm.js	autoscout.de	Tracks user activities

Table 10. Ten JavaScript programs our classifier correctly classified as tracking and all the PP-Tools wrongly classified as functional verified through manual labelling.

#	Website	JavaScript Program	Referred Domain	Function Performed
1	crateandbarrel.com	j.c-b.co/js/account_1505080410.js	crateandbarrel.com	Creates user accounts
2	suomi24.fi	kiwi27.leiki.com/focus/mwidget.js	leiki.com	Magnifying widget
3	nhl.com	b3.mookie1.com/2/LB/3115965742.js	cdn-akamai.mookie1.com	Fetches content
4	michael.com	edgesuite.net/js/picturefill.min.js	-	Fetches content
5	ing.nl	ensighten.com/ing/NL-ingnl-prod/code/fc_90aaa8fc7.js	ing.nl	Fetches content
6	worldoftanks.ru	mc.wargaming.net/tsweb.js	wargaming.net	Sets session cookie
7	divyabhaskar.co.in	nr.taboola.com/newsroom/bhaskar-divyabhaskar/getaction.js	taboola.com	Enables user interaction
8	15min.lt	15minadlt.hit.gemius.pl/_1431091788674/redot.js	squarespace.com	Timestamps user login
9	abovetopsecret.com	casalemedia.com/j.js	abovetorespect.com	Fetches content
10	buzzfeed.com	ct-ak.buzzfeed.com/wd/UserWidget.js	s3.amazonaws.com	Fetches content

Table 11. Ten JavaScript programs our classifier correctly classified as functional and all the PP-Tools incorrectly classified as tracking verified through manual labelling.

(Section 7.2.2). In the semantic feature models, our classifiers are resilient against the first two types of JavaScript code obfuscation strategies: *(i)* rename function or variable names and *(ii)* add or remove whitespaces. For instance, Criteo[17] sets its tracking cookies at `dailymotion.com` with the JavaScript code shown in Listing 3. By using an online obfuscation tool [11], the obfuscated version of this code is shown in Listing 1, with variable names replaced and white spaces removed (i.e., type *(i)* and *(ii)* obfuscation as mentioned in Section 7.2.2). The canonical form (cf. Appendix A.2) of both the original code and the obfuscated code is the

(a) (b)

Fig. 8. Ghostery's filters (a) video content by blocking `brightcove` widget on www.9news.com.au. Once unblocked, `brightcove` loads (b) video content on the web-page.

same, as shown in Listing 2, meaning that our semantic feature models based classifiers will still detect the obfuscated, tracking JavaScript codes.

Listing 3. An Example of JavaScript Program

```
var crtg_nid="1822";
var crtg_cookiename="co_au";
var crtg_varname="crtg_content";
function crtg_getCookie(c_name){
var i,x,y,ARRCookies=document.
                   cookie.split(";");
  for(i=0;i<ARRCookies.length;i++)
  {
     x=ARRCookies[i].substr(0,ARRCookies[i].
            indexOf("="));
     y=ARRCookies[i].substr(ARRCookies[i].
            indexOf("=")+1);
     x=x.replace(/^\s+|\s+$/g,"");
     if(x==c_name){
   return unescape(y);
   }
   }
   return "";
}
```

Listing 1. Type *(i)* and *(ii)* obfuscation.

```
var a="1822";var b="co_au";var c="crtg_
content";function crtg_getCookie(e){var i,x,y,
d=document.cookie.split(";");while(i<d.length)
{x=d[i].substr(0,d[i].indexOf("="));y=d[i].sub
str(d[i].indexOf("=")+1);x=x.replace(/^\s+|\s+
$/g,"");if(x==e){return unescape(y)}i++}
return""}
```

Listing 2. Canoncial form of original and obfuscated code.

```
function crtg_getCookie=
         function crtg_getCookie(e){
  begin;$0 = document.cookie;
  d = $0.split(";");
  $1 = d.length; $2 = i < $1;
  while ($2) { $3 = d[i];$4 = d[i];
    $5 = $4.indexOf("=");
    x = $3.substr(0, $5);
    $6 = d[i];$7 = d[i];
    $8 = $7.indexOf("=");
    $9 = $8 + 1;
    y = $6.substr($9);
    x = x.replace(RegExp("^\s+|\s+$","g"), "");
    $10 = x == e;
    if ($10) { $11 = unescape(y);
      return $11;}
    $12 = i; i = i + 1;
    $13 = d.length;
    $2 = i < $13;}
  return "";
  end;
  };
  $14 = %InitializeVarGlobal("a", 0, "1822");
  $15 = %InitializeVarGlobal("b", 0, "co_au");
  $16 = %InitializeVarGlobal("c", 0,
               "crtg_content");
  end;
```

On the other hand, the attacker (i.e., tracker) might evade detection by applying the *(iii)* and *(iv)* types of JavaScript code obfuscation techniques (cf. Section 7.2.2). We believe that this is the classical arms race issue, to which all machine learning based detection methods are prone, in which the trackers pay higher cost in trying to obfuscate their code to evade detection, since (a) as discussed, simple obfuscation fails, and (b) moreover, the obfuscation will need the additional guarantee of being detection proof as our machine learning techniques can re-learn newly introduced tracking JavaScript code (if enough trackers decide to obfuscate and the obfuscated JavaScript code still have structural similarities). Thus, trackers would need to resort to even more sophisticated forms of obfuscation. It is also important to mention that blacklist based approaches still persist in spite of the fact that trackers can change their URLs to evade popular patterns, or simply rename JavaScript programs to defeat regular expression matching. Note that sophisticated form of obfuscation remains a challenge for other detection systems as well, e.g., malware detection. Despite the availability of obfuscation tools, our classifiers achieve higher efficiency and detect trackers missed by contemporary PP-Tools. We nevertheless regard this as a limitation and believe it to be an interesting area of future research.

In the future, we aim to characterize and study JavaScript code obfuscation techniques employed by trackers. In essence, we aim to investigate JavaScript's dynamic code generation and run-time evaluation functions, e.g., `eval()` and `document.write()`. We believe that the investigation of the arguments supplied to these functions can be leveraged in the re-training of our classifiers and in the detection of possible obfuscated tracking and malicious JavaScript programs.