Read Between the Lines: Detecting Tracking JavaScript with Bytecode Classification

Mohammad Ghasemisharif
University of Illinois Chicago
Chicago, Illinois, USA
mghas2@uic.edu

Jason Polakis
University of Illinois Chicago
Chicago, Illinois, USA
polakis@uic.edu

ABSTRACT

Browsers and extensions that aim to block online ads and tracking scripts predominantly rely on rules from filter lists for determining which resource requests must be blocked. These filter lists are often manually curated by a community of online users. However, due to the arms race between blockers and ad-supported websites, these rules must continuously get updated so as to adapt to novel bypassing techniques and modified requests, thus rendering the detection and rule-generation process cumbersome and reactive (which can result in major delays between propagation and detection). In this paper, we address the detection problem by proposing an automated pipeline that detects tracking and advertisement JavaScript resources with high accuracy, designed to incur minimal false positives and overhead. Our method models script detection as a text classification problem, where JavaScript resources are documents containing bytecode sequences. Since bytecode is directly obtained from the JavaScript interpreter, our technique is resilient against commonly used bypassing methods, such as URL randomization or code obfuscation. We experiment with both deep learning and traditional ML-based approaches for bytecode classification and show that our approach identifies ad/tracking scripts with 97.08% accuracy, significantly outperforming cutting-edge systems in terms of both precision and the level of required features. Our experimental analysis further highlights our system's capabilities, by demonstrating how it can augment filter lists by uncovering ad/tracking scripts that are currently unknown, as well as proactively detecting scripts that have been erroneously added by list curators.

CCS CONCEPTS

• **Security and privacy → Privacy protections**.

KEYWORDS

Measurement; Web security; Privacy; Ad/tracking blocking

ACM Reference Format:
Mohammad Ghasemisharif and Jason Polakis. 2023. Read Between the Lines: Detecting Tracking JavaScript with Bytecode Classification. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3576915.3616637

1 INTRODUCTION

The modern web has been marked by the proliferation of web trackers and advertisements. While certain websites include tracking scripts to support web analytics and use the data to customize their content and improve the user experience, other online entities supply sites with scripts for tracking users across different sites. This large-scale collection of data powers the ad ecosystem and allows companies to profit by selling users' data to the highest bidders. The *stateful* (e.g., using cookies [21, 61, 62] or other stored identifiers [67]) or *stateless* (e.g., using browser fingerprints [28, 42, 48, 66]) tracking of users' activities, and the sharing of data with other trackers via *cookie syncing* [49], have become common practice [26, 35, 56]. This not only diminishes the user experience by bloating the website [29], but also violates their privacy. Users are becoming increasingly aware of these privacy-invasive techniques and attempt to take actions for protecting their data [23].

Countermeasures typically rely on blocking invasive scripts. In more detail, blocking rules are designed to block external scripts as well as requests related to advertising or tracking, as third-party trackers primarily use JavaScript (e.g., browser APIs used for fingerprinting) to generate unique identifiers or serve ads, and HTTP requests for transferring these identifiers. Browser vendors such as Brave [20] and popular content-blocking extensions [2, 31] use manually curated and crowd-sourced filter lists, such as EasyList [5] and EasyPrivacy [6], to detect and block communications with the endpoints. Due to the ongoing arms race between trackers and content blockers [36], these filter lists need to be frequently updated through a rigorous *manual process* by a group of online users.

As a result, automated detection systems have been proposed to complement the manual effort. These systems typically utilize features extracted from the network, JavaScript, and HTML layers [17, 32, 34, 37, 38, 41, 41, 60, 74, 76]. In response, advertisers and trackers have employed evasion techniques such as CNAME cloaking [24], URL randomization [72], and JavaScript obfuscation [63] to avoid being detected. Accordingly, more advanced systems utilize multiple features to improve their robustness. Nonetheless, such techniques can still be vulnerable to adversarial evasion attacks based on how a given feature's weight influences the system's decision. For instance, AdGraph [37] employs various inputs like network information and content-based features. However, content features such as the URL length have a greater influence on the model's results and can be easily tampered with, making it more susceptible to manipulation of those features. Subsequently, methods that assign a higher priority to the request's origin (i.e., being a third-party) [32, 41, 73] may not be effective in identifying first-party trackers. Therefore, the effectiveness of a detection system depends on the robustness of the features the system operates on.

Certain websites may intentionally, or inadvertently, tie the website's functionality to blocked resources, which can result in website breakage when these resources are blocked. Such malfunctions are often reported to the filter list maintainers and are fixed by including exception rules that allow such requests to bypass the content blockers. For example, EasyPrivacy blocks `googletagmanager.com/gtm.js` in most websites, but allows it to bypass tracking blockers on certain domains. Systems that only rely on URL-based features fail to distinguish between the two identical requests, rendering systems ineffective in detecting allowlisted resources. While breakage analysis through manual inspection [37, 38, 43, 60] can evaluate the impact of incorrectly blocked functional resources, it does not assess the models' ability to detect allowlisted resources, especially at scale, which is an essential part of the update cycle for filter lists.

In this work, we propose a method for accurately identifying external JavaScript resources that exhibit advertising and tracking characteristics. Ultimately, our goal is to address the deficiencies of previous detection methods by utilizing features that cannot be trivially altered. Our method is origin-agnostic (i.e., it can be applied to both first-party and third-party), does not rely on URL features, and is independent of static features in the code. Instead of extracting specific features such as Web API calls [41, 70, 73] using dynamic analysis, our system utilizes the information obtained directly from the JavaScript interpreter as a representation of the execution flow, for classifying the script's behavior. This makes our method resilient against code obfuscation and URL modification, and also allows us to classify allowlisted scripts based on their different execution patterns (i.e., being categorized as AT on one website and benign on another).

To that end, we instrument Chromium's V8 engine to append necessary additional information to the bytecode sequences we obtain, and use it to crawl, collect, and label scripts from the top 50K websites according to Tranco [52]. We then approach bytecode classification as a text classification task and evaluate three different models including both supervised machine learning and deep learning models that differ in generalizability, performance, and resource requirements. Our extensive experimental evaluation shows that these classifiers can detect ad and tracking scripts with 97.08% accuracy, demonstrating their effectiveness in automating the time-consuming task of creating filter lists. We compare our bytecode classification technique to prior cutting-edge approaches and find that bytecode classification outperforms the existing detection systems by 6.37-11.22% in accuracy. Crucially, our longitudinal analysis of filter lists shows that our models can identify scripts that are undetected by current filter lists, incorrectly detected as trackers and later removed or allowlisted by the lists.

In summary, our work makes the following contributions:

(1) We design a novel bytecode-classification-based approach for identifying ad and tracking scripts that is origin-agnostic, considers code dependencies, and is resilient to evasion techniques like URL and code obfuscation.
(2) We extensively explore the performance of bytecode classification using both traditional machine learning and deep learning models. Our comprehensive evaluation shows that bytecode classification outperforms state-of-the-art systems.
(3) To enable reproducibility, facilitate additional research, and assist filter list maintainers in automating the filter list lifecycle process, we will publicly release our code and data https://github.com/byte-learn/byte-learn.git.

2 BACKGROUND & RELATED WORK

2.1 Preliminaries

The crux of this work is classifying *(external) JavaScript resources*, which are files that have a `script` resource type and are fetched as users visit websites. Each script contains JavaScript executable source code, and as the scripts are parsed and syntax trees are created the interpreter generates the *bytecode* representation for the executable code; in this work, we use the bytecode to represent each script and predict its behavior. Throughout this paper we refer to ad or tracking JavaScript resources as *malicious scripts* or, for brevity, **AT**. We also refer to non-tracking/ad scripts as *non-malicious* or *benign*. These scripts are often essential for websites' functionality and do not carry out AT actions. However, there are some scripts that exhibit both AT and benign behavior. We discuss the implication of such cases in §6.

We consider two content blocking methods: *online* and *offline*. In the online approach, features are extracted and rules are generated in real-time during the web page visit. Implementing these methods commonly requires in-browser instrumentation, which can add overhead to the page load. Offline approaches, or blocklist-based techniques, rely on lists that contain rules that provide instructions for blocking browser requests. EasyList [5] and EasyPrivacy [6] are prevalent examples of such lists. In contrast to online methods, these rules are typically manually crafted beforehand, and updated over time by a group of web users in a tedious crowd-sourced effort.

We consider three main steps in the filter list lifecycle: 1) detection, 2) rule generation, 3) update and/or correction. Improving the detection step involves finding a set of features that can reliably identify AT requests, which is the primary focus of prior work in this space. The rule generation step uses the information obtained during the detection process to create a set of efficient rules. Lastly, the generated rules are updated as the ongoing arms race between advertising/tracking entities and content blockers continues. While step (3) has been mostly overlooked by prior AT detection studies, our work is an *offline* approach that addresses both (1) and (3).

The update cycle also includes adding exception rules, where certain blocked resources are *allowlisted* to prevent breakage in websites. Listing 1 illustrates an allowlisted example from EasyPrivacy. The first rule (line 1) instructs content blockers to block requests to any URL that contains the domain `googletagmanager.com`, whereas the second rule (line 2) makes an exception for specific requests towards `googletagmanager.com/gtm.js` when the originating domain is `okwave.jp` because blocking it causes breakage [1]. We use *allowlisted* to describe these exception rules (indicated by the "@@" prefix) and provide more details in Appendix A.1.

2.2 Feature Selection & Challenges

Existing methods use JavaScript-based properties, request-based features, or a combination of both, to identify and block advertising

Listing 1: An example of filter syntax in EasyPrivacy

```
||googletagmanager.com^
@@||googletagmanager.com/gtm.js$script,domain=okwave.jp
```

and tracking related resources on webpages in mobile [58, 59, 75] or desktop browsers [34, 37, 41, 60]. These properties are often used to train classifiers for identifying AT requests and/or facilitate the creation of filter lists in an offline approach.

Request-based features include the HTTP request/response structure and URL or domain characteristics, such as whether domains belong to third-parties [32, 41], URLs contain special keywords [17], or requests are sent consecutively to a domain [38]. A combination of these features is used to determine whether a request belongs to AT. However, adversarial evasion techniques can bypass request-based detection, such as randomizing the hostname and path [72], moving a domain to a different origin, or simply removing keywords from URLs [65]. Third-party trackers have been a focus of many studies, while first-party trackers are often overlooked and considered inconsequential because they do not track users across different websites. As a result, detection systems that rely on the request's origin are also vulnerable to evasive techniques, like CNAME cloaking [24, 25]. More importantly, URL features alone are not sufficient for detecting allowlisted resources, as the exact same URL can be allowed for one domain and blocked for another. These shortcomings highlight the importance of selecting features that cannot be easily circumvented.

JavaScript-based features typically refer to syntactical or structural attributes of JS code, such as identifying the script's behavior based on a sequence of API calls [41] or whether it produces predefined signatures in the event loop [22]. Such features are extracted statically or dynamically. However, relying on *static features* may fail to detect dynamically generated code, or incorrectly identify code that is not actually being executed. For instance, features obtained from a function that is defined in the code but never called would be incorrectly taken into account. More importantly, static analysis is vulnerable to JavaScript obfuscation [27, 63]. To bypass obfuscation, extraction methods can collect such features directly from the JavaScript engine through instrumentation.

2.3 Motivation

Given the limitations of relying on request-based and JavaScript features, a resilient detection system must utilize features that do not require prior knowledge about the AT method, and are not easily evaded. If a set of *known* features is used, it is possible that features that participated in the AT scripts will be overlooked, due to not being known at the time of signature creation. For instance, a classifier that uses a set of known Web API calls to determine a script's behavior will be outdated when such features are removed or replaced and require additional investigation to examine the effectiveness of newly added APIs. Moreover, features that have bigger weights in a classifier's decision can be modified by adversaries for bypassing detection systems [37, 60]. To tackle these shortcomings, we consider the task of script classification as a sequence classification problem. To do this, we represent scripts using sequences of bytecode generated by the interpreter and evaluate different classifiers to predict each script's behavior. Using bytecode presents multiple advantages:

Table 1: Comparison of our JavaScript bytecode classification method in detecting tracking and/or ad scripts to related work. We label data samples as (S)mall (below 10k), (M)edium (10k-100k), and (L)arge (above 100k).

Method	**Features**		**Focus**		**Party**		**Susceptible**		**Allowlist**	**Sample**
	Req	JS	Track	Ad	1st	3rd	Req man	JS obf	**Analysis**	**Size**
Bhagavatula et al. [17]	✓	✗	✓	✓	✓	✓	✓	✗	✗	M
Gugelmann et al. [32]	✓	✗	✓	✓	✗	✓	✓	✗	✗	S
Wu et al. [73]	✗	✓	✓	✗	✗	✓	✗	✗	✗	S
Kaizer and Gupta [41]	✓	✓	✓	✗	✗	✓	✓	✗	✗	M
Ikram et al. [34]	✗	✓	✓	✗	✓	✓	✗	✓	✗	S
Sun et al. [70]	✗	✓	✓	✗	✗	✓	✗	✗	✗	S
WebGraph [60]	✓	✓	✓	✓	✓	✓	✗	✗	✓/✗*	L
Bytecode Classification	✗	✓	✓	✓	✓	✓	✗	✗	✓	L

* Their analysis focuses on website breakage.

- **Representativeness:** V8's lazy parsing generates bytecode for the invoked functions, offering a representation of the JavaScript code that was *actually* executed and excluding unexecuted functions, which impact techniques that rely on static analysis and can lead to misclassification (false positives/negatives). Leveraging bytecode enables us to take the entire execution into account and classify its behavior, compared to other dynamic analysis methods that rely on predefined signatures. In §5, we compare bytecode classification's performance with prior work that uses predefined features including Web API calls. We find that such preset features can result in misclassifications; for instance, `hCaptcha` [10] being misclassified as AT while our bytecode classification successfully captures the appropriate context and correctly classifies it. Conceptually, our bytecode-based approach is akin to classifying text based on its meaning rather than merely relying on the presence of specific keywords.
- **Resilience:** Bytecode is *resilient* against obfuscation techniques, such as code obfuscation and URL randomization, as it is obtained *directly* from the JavaScript engine. While similar scripts can generate different bytecode sequences (e.g., invoke different functions), in contrast to approaches that rely on predefined signatures, such nuances can be mitigated by a context-aware text classification model that captures the syntactic and semantic relationships between bytecode sequences.
- **Efficiency:** Bytecode can be quickly extracted from a large collection of scripts with minimal overhead. Having a large set of samples can closely represent the input space compared to smaller sets used in prior work (due to the need for manual inspection or instrumentation overhead). Even though training such a classifier has an upfront cost, the inference cost is minimal. We provide a more in-depth discussion in §6.
- **Accuracy:** Our technique can uncover blocked scripts that should be removed or allowlisted in addition to new AT scripts that current filter lists have not yet detected.

2.4 Related Work

Due to the painstaking effort and challenging nature of manually curating rules, previous studies have demonstrated that errors are unfortunately common in popular filter lists [65]. This is especially problematic as over half the websites could suffer from false positive errors lasting more than a month, and obsolete rules can take two to three months to be removed [16]. While a plethora of prior studies have focused on content blocking, our approach overcomes the

Listing 2: An example of code dependecy

```
1 <!-- external script where tracking() function is declared -->
2 <!-- function tracking(){ ... } -->
3 <script src="https://domain.com/tracking.js"></script>
4
5 <!-- inline script calls the function from external script -->
6 <script>
7     var uid = tracking();
8 </script>
```

Listing 3: Declared function in tracking.js

```
function tracking(){
    // no bytecode generated for unused_func()
    function unused_func(){ ... }
    // tracking()'s bytecode from here (if invoked)
    var uid = ...;
    return uid;
}
```

pitfalls of prior work by using features that are more robust to evasion and achieves higher accuracy in both the detection and update phases of the filter list lifecycle, and also achieving early detection of both false positives and false negatives.

Online techniques. Such approaches often rely on a special tool (e.g., instrumented browser) that enables their features to be computed on-the-fly, and blocking decisions are made in real-time. AdGraph [37] creates a graph representation of the website during page load and uses the features extracted from the graph to decide which requests should be blocked. The top features used in AdGraph are content-based features, such as URL length or the script's origin, which are prone to adversarial evasions. Khaleesi [38] is another online approach that focuses on detecting advertising and tracking *request chains* using request and response features, and sequential features such as the number of unique domains in the chain.

Offline methods. In practice, popular AT blockers (e.g., Ad Block Plus [2]) and browsers (e.g., Brave [20]) use offline techniques that rely on pre-computed filter lists. Many existing studies have focused on improving (and automating) the rule-generation process, which relies on human intervention, because incorrect rules can cause breakage and have unwanted consequences. These studies use different feature sets and train supervised and unsupervised classification models to identify potential AT scripts. Their methods vary in terms of the type of features they use, their classification technique, and the sample size. Bhagavatula et al. [17] used URL features such as the presence of ad-related keywords or particular parameters in query strings to train a machine learning model. The authors in [70] used Web API calls to classify JavaScript execution, while [22] focused on the behavior of scripts during the JavaScript event loop to identify AT behavior. Kaizer and Gupta [41] leveraged request-based features such as the presence of `referer` and cookies in HTTP headers in addition to JavaScript features such as web API calls. WebGraph [60] is an offline implementation of AdGraph that was introduced to improve the robustness of the graph-based approach and reduce AdGraph's susceptibility to evasion. In contrast to our work, WebGraph employs features extracted from the network, storage, JavaScript, and HTML layers via instrumentation, which greatly increases the workload. We compare the core aspects of our work to prior studies in Table 1.

JS bytecode. To the best of our knowledge no prior work has used JavaScript bytecode sequences as a predictor for AT scripts. The closest study was conducted by Rozi et al. [55], which uses

Chromium

```
[generated bytecode for function: (0x2bda7b5bb8e9
<SharedFunctionInfo>)]
Parameter count 7
Register count 33
Frame size 264
0x2bda7b5bcf96 @ 0  : 84 00 4b CreateFunctionContext [0],[75]
0x2bda7b5bcf99 @ 3  : 16 e7    PushContext r20
0x2bda7b5bcf9b @ 5  : 25 06    Ldar a1
0x2bda7b5bcf9d @ 7  : 1d 05    StaCurrentContextSlot [5]
0x2bda7b5bcf9f @ 9  : 25 04    Ldar a3
0x2bda7b5bcfa1 @ 11 : 1d 04    StaCurrentContextSlot [4]
0x2bda7b5bcfa3 @ 13 : 0f       LdaTheHole
0x2bda7b5bcfa4 @ 14 : 1d 06    StaCurrentContextSlot [6]
0x2bda7b5bcfa6 @ 16 : 0f       LdaTheHole
0x2bda7b5bcfa7 @ 17 : 1d 07    StaCurrentContextSlot [7]
0x2bda7b5bcfa9 @ 19 : 0f       LdaTheHole
...
```

Instrumented Chromium

```
[generated bytecode for function: (0x2bda7b5bb8e9
<SharedFunctionInfo>)]
Parameter count 7
Register count 33
Frame size 264
Bytecode: [CreateFunctionContext,PushContext,Ldar,
StaCurrentContextSlot,Ldar,StaCurrentContextSlot,LdaTheHole,
StaCurrentContextSlot,LdaTheHole,StaCurrentContextSlot,
LdaTheHole,...]
Source: function (){...}
URL: https://domain.com/resource.js
...
```

Figure 1: Bytecode generated for a function using unmodified Chromium (top) and our instrumented Chromium (bottom) which only keeps bytecode needed for our method and adds the script URL of this function.

bytecode classification to identify *malware*. There are several major differences in our work. First, we focus on an entirely different class of *malicious* behaviors. In their work AT and functional scripts belong to the same class, whereas our objective is to identify JavaScript files that exhibit AT behavior. Moreover, blocking AT indiscriminately can cause breakage. Thus, an effective model must also distinguish between disruptive and non-disruptive AT, highlighting the unique challenge. More importantly, they execute JavaScript files in the `jsdom` [12] environment which is slow, injects unrelated (`jsdom`) bytecode sequences into the dataset, and does not work in the web ecosystem, where dependent scripts loaded from different locations are needed for correct and complete execution. This is important due to how V8 generates bytecode. Since parsing and generating bytecode for all functions can be costly, V8's lazy parsing[13] approach parses and generates bytecode when the functions are invoked. The functions can be invoked within the scripts they were declared or from other scripts. Listing 2 illustrates an example of an inline script (line 7) that calls a function declared in an external script (line 3). In this example, loading the external script in `jsdom`, without invoking the `tracking()` function will not generate a bytecode sequence due to the lazy parsing. Since we do not have a-priori knowledge about such code dependencies, we cannot know which scripts (inline or otherwise) must be included prior to execution for `jsdom` to generate bytecode that is equivalent to visiting the website. To accurately capture the script's bytecode, we need to render the web page in a browser, obtain the bytecode and create references from bytecode sequences to their origin, which is not a capability provided by V8. To that end, we instrument V8 to obtain bytecode faster with less noise (i.e., unrelated bytecode) and the ability to capture code dependencies. It is particularly important to ignore unrelated bytecode where the sequences can be extremely long, especially for training neural networks, as GPU memory is expensive and limited.

3 METHODOLOGY

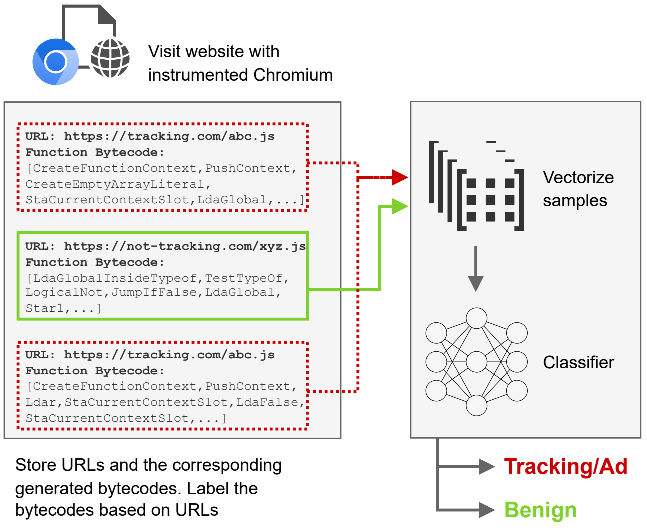
Here we describe our pipeline for collecting, pre-processing and labeling the data prior to training. First, we present our instrumented browser for collecting data. Next, we describe the process of generating, cleaning and storing bytecode sequences. Third, we provide an overview of obtaining word and document embeddings. Finally, we describe our bytecode classification models.

3.1 V8 Instrumentation

V8 is a JavaScript engine used in Node.js and Chromium. To execute a piece of JavaScript code, the JavaScript source code is first parsed into an Abstract Syntax Tree and then the interpreter generates bytecode from it. The bytecode will be used to generate optimized machine code. To obtain bytecode from the interpreter, one can run Node.js or Chromium with the `-print-bytecode` flag. Figure 1 (top) shows an example of the printed bytecode for a function. A script can have several functions, but the generated bytecode for each function is printed separately and can be attached together to represent a script. Note that the printed bytecode for a function represents the parts that will be executed. An example of this is shown in Listing 3 where `unused_func()` does not generate bytecode even though it is declared inside an invoked function. However, V8 does not include the source code or the URL of the script. Therefore, when a website is visited, a stream of bytecode sequences is generated, but there are no corresponding links or source code to indicate which bytecode sequence belongs to which script. We instrument the `DoFinalizeJobImpl` function in V8's interpreter in Chromium to obtain the script's source code and its URL (if available) from the `SharedFunctionInfo` object and print them along with the bytecode sequences. Since bytecode sequences can be lengthy, we also modify the `Disassemble` and `Decode` functions to only print the bytecode sequence used by our system. In total, we added 40 lines of code for including source code and URL, and modified/removed about 130 lines to change the output format. We anticipate that this code will be easily adaptable for future versions of V8. Figure 1 (bottom) shows an example of the generated bytecode from our instrumented V8. Adding the script URL allows us to collect the function bytecode and label it based on the originating URLs. For brevity, throughout the paper, we refer to the Chromium browser with our instrumented V8 engine as "instrumented Chromium".

3.2 Pre-processing

Generating Bytecode. Training our classification models requires a large number of scripts with their corresponding binary labels (AT/benign). To collect the scripts, we use our instrumented Chromium and developed a tool using Puppeteer [14] to crawl the homepage of the top 50K websites from the Tranco list [52]. Our tool visits a website, listens on `requestfinished` network events (i.e., successful


Visit website with instrumented Chromium
Vectorize samples
Classifier
Store URLs and the corresponding generated bytecodes. Label the bytecodes based on URLs
Tracking/Ad
Benign

```
URL: https://tracking.com/abc.js
Function Bytecode:
[CreateFunctionContext,PushContext,
CreateEmptyArrayLiteral,
StaCurrentContextSlot,LdaGlobal,...]
```

```
URL: https://not-tracking.com/xyz.js
Function Bytecode:
[LdaGlobalInsideTypeof,TestTypeOf,
LogicalNot,JumpIfFalse,LdaGlobal,
Star1,...]
```

```
URL: https://tracking.com/abc.js
Function Bytecode:
[CreateFunctionContext,PushContext,
Ldar,StaCurrentContextSlot,LdaFalse,
StaCurrentContextSlot,...]
```

Figure 2: The process of collecting and classifying bytecode. Note that AT and benign bytecode sequences are combined and then labeled based on their associated URLs prior to training. The red dotted lines illustrate AT related bytecode whereas the green line specifies that the bytecode belongs to a benign script.

requests) and if the request's resource type is a `script` it collects the request's URL and the script's content. We also incorporate Adblock-rs [19] to parse the Adblock Plus syntax and label each request as AT (true) or not (false). We use EasyList [5], EasyPrivacy [6] and Fanboy's Enhanced Tracking [7] rules as ground truth for classifying requests. We label allowlisted resources as false since filter lists detect them as benign. After visiting the website, our pipeline saves the printed bytecode to a file. Another script extracts the bytecode from the file and stores it in a database. Visiting a website and storing the generated bytecode are the initial steps in our pipeline, and illustrated on the left side of Figure 2. We note that bytecode sequences, such as those shown in Figure 2's example, are not meaningful to humans and do *not* have visually distinguishable features. In other words, one cannot distinguish AT from benign scripts by simply perusing and contrasting their bytecodes. As such, there is a need for a machine-learning-based approach for extracting and identifying patterns in bytecode sequences that concretely capture AT functionality in scripts. After storing all function bytecode generated from visiting a website, a Python script parses the output, combines the function bytecode of each script based on its URL, and stores them in the database. Each row in the database contains information about the website's rank and hostname, the script's URL, its source code and the associated label.

Data Cleaning. Each script generates a stream of bytecode outputs that are separated per function. The top item in Figure 1 shows a sample output for a function generated by a Chromium browser. The bottom item shows the "cleaned" bytecode generated using our instrumented Chromium. To clean the data, we ignore most of the output and only extract the bytecode sequences without register information. We also ignore any bytecode that does not have a script

URL, or that belongs to Puppeteer (indicated by `_puppeteer_`), or has no source code associated with it. We consider each bytecode as a separate word and each function as a separate sentence. After parsing and cleaning, the end result for each script forms a list of sentences in a document. Finally, since different scripts can generate the same bytecode sequences, we locate and remove duplicate bytecode to avoid any pollution during our experiments and ensure that all test data will be unseen.

Data Compression. The bytecode sequences can be very long with repetitive words, which takes up unnecessary space when stored. We opt to compress data by replacing each bytecode with a shorter word. Each bytecode in our dataset has a length between 3-42 characters. The distribution of bytecode sequence length in our dataset is illustrated in Figure 5. In total, we gathered 524 bytecode from both Chromium's source code (488) and traversing and identifying unique bytecode in the collected data (36). Therefore, every bytecode in our data can be represented with a 2-character word. We created a word mapping dictionary by assigning each bytecode to a randomly generated 2-character string and substituted all instances of that bytecode in the documents. The total number of unique bytecode observed in our data is 339. The compressed bytecode sequences required 76.32% (58 GB) less storage space compared to the original bytecode.

3.3 Word Embedding

In order to train our model, we first need to create a representation for each word; this task is shown in Figure 2 as the sample vectorization part of the pipeline. There are various methods for converting the documents to vectors (e.g., count vectors), which vary in computation time and the amount of information they can capture in terms of context and semantics. We use two methods, Word2vec [47] and FastText [18], to learn a distributed word (i.e., bytecode) representation in a vector space and create the word embeddings. Prior work has shown that using unsupervised embeddings in DPCNN improves performance [39]. Word2vec is an unsupervised algorithm that can be used to create a vector for each word that keeps the semantic closeness of the words by learning which words the target word more often occurs with. Word2vec has two architectures: Continuous Bag of Words (CBOW) and Skip-Gram. The CBOW architecture predicts the word given the neighboring words (context) while Skip-Gram predicts the neighboring words in a certain range before and after a given word. FastText follows a similar approach but focuses on character n-grams: a word representation using the sum of character n-grams. We provide additional details about FastText in the following subsection.

The output layer of the Skip-Gram model is a softmax layer with $1 \times V$ dimensions, where V is the size of the vocabulary and each value in this probability vector represents the probability score of the word in that position. However, we only need the hidden layer's output for creating word embeddings. After training on the corpus, the weight matrix in the hidden layer is used to calculate the vector representation of each bytecode, and the vectorized bytecode sequences are used to train our model. We note that while increasing the word range in the Skip-Gram architecture increases the prediction accuracy, it also increases the computational complexity [47]. Therefore, tuning the parameters to achieve the best accuracy with reasonable performance is crucial, as we detail in §4.1.

Table 2: Number of all and unique script samples collected. We use unique bytecode sequences for training and testing.

Dataset	Label	#Samples
All JavaScript	Benign	545,223
	Ad/tracking	368,667
Unique JavaScript	Benign	265,861
	Ad/tracking	130,509

Document Embedding. After training, Word2vec generates a vector representation for each word in the vocabulary. We directly use this as the embedding layer of DPCNN, which is a look-up table for vectors of each word obtained from Word2vec. Since Word2vec only provides word embeddings we need to use another method to represent the document (i.e., script) vector which will be used in traditional machine learning methods. One popular method is calculating the mean of all the word vectors in the scripts to represent each script as a vector [44, 57]. We use this approach to prepare our data for training a random forest classifier and compare our ML-based classifier with our deep learning model.

3.4 Classification

Next, we provide an overview of our classifiers, which differ in terms of performance and training costs as we detail in later sections.

DPCNN. The Deep Pyramid Convolutional Neural Network [39] is a text classification model that can efficiently capture global information and long-range associations in the text [39]. The model begins with a text region embedding with optional unsupervised embeddings (for improving accuracy). We train Word2vec on our dataset and use it to transform bytecode sequences into vectors for this layer. The first feature extraction layer consists of two convolutional layers with a max-pooling layer and a skip connection with pre-activation. This can be written as $z + f(z)$ [33], where $f(z)$ are the two convolutional skipped layers with pre-activation and the activation function $\sigma(.)$ is the rectifier $\sigma(x) = max(0, x)$. After each convolution block, there is a max-pooling layer with pooling stride 2 and kernel size 3 which reduces the size of the representation by half. The repeated *halving* reduces the size of the document's internal representation and forms a *pyramid*.

A typical text document has a large vocabulary of words with a few repetitive neighboring words. However, the nature of our dataset is different: each document has a relatively small vocabulary with very long sequences. The document length distribution is shown in Figure 5. Therefore, we need to change the default parameters that were originally proposed by [39] to improve the model's performance in this specific domain. We use a PyTorch implementation of DPCNN [53] to train and test the model.

FastText. FastText [18] attempts to improve Word2vec by extending the Skip-Gram model to include character-level information and represent the internal structure of the words. This model is particularly useful when a word can have a large number of morphological forms. In this model, words are bag-of-character *n-grams*. For instance, the word "track" with $n = 3$ will be decomposed into <tr, tra, rac, ack, ck> as well as the special sequence <track>, where < and > are special characters added at the beginning and end of the words. Each n-gram is then represented by a vector, therefore the word will be represented by the sum of these

Table 3: Top 10 FQDNs that served AT JavaScript files. The reported counts for all collected scripts and scripts with unique bytecodes are separated.

Unique Scripts		All Scripts	
FQDN	Count	FQDN	Count
www.googletagmanager.com	33,234 (25.46%)	www.googletagmanager.com	41,869 (11.36%)
www.google-analytics.com	11,933 (9.14%)	www.google-analytics.com	26,812 (7.27%)
connect.facebook.net	6,652 (5.1%)	connect.facebook.net	23,481 (6.37%)
securepubads.g.doubleclick.net	5,351 (4.1%)	googleads.g.doubleclick.net	12,397 (3.36%)
tags.tiqcdn.com	3,834 (2.94%)	tpc.googlesyndication.com	11,935 (3.24%)
pagead2.googlesyndication.com	3,558 (2.73%)	www.googleadservices.com	9,935 (2.69%)
www.clarity.ms	2,624 (2.01%)	pagead2.googlesyndication.com	8,781 (2.38%)
www.googletagservices.com	2,383 (1.83%)	securepubads.g.doubleclick.net	7,646 (2.07%)
script.hotjar.com	2,224 (1.7%)	assets.adobedtm.com	7,529 (2.04%)
mc.yandex.ru	1,399 (1.07%)	tags.tiqcdn.com	7,145 (1.94%)

character n-gram vectors. FastText can be used for both *supervised* and *unsupervised* learning. FastText's supervised learning for text classification [40] uses document/sentence vectors as features by averaging the word/n-gram vectors and uses multinomial logistic regression to perform text classification. For obtaining word embeddings, we use Gensim's [54] implementation of FastText's unsupervised learning since we use the same library for Word2vec. For the classification task, we use [8]'s implementation.

Random Forest (RF). We use a random forest classifier to compare the performance of a classic machine learning technique, which also requires less time and resources to train. Similar to the DPCNN model, we need to represent our inputs as vectors. While in DPCNN the word embedding acted as a lookup table for word vectors, here we need to find a representation for each script document. We use the same vectors generated by Word2vec and FastText with the difference being that for each script we create a document/sentence vector v by replacing each bytecode (word) with its corresponding b_i vector (obtained from Word2vec) and then averaging all the bytecode vectors:

$$v = \frac{1}{n} \sum_{i=1}^{n} b_i$$

where n is the number of bytecode (words) in the script and v's dimension is the same as the bytecode vectors generated by Word2vec. We follow the same method with vectors created by FastText to compare its performance with Word2vec's vectors. Finally, for the classification task, we use the `scikit-learn` [51] implementation of a random forest classifier.

4 EXPERIMENTAL SETUP

Here we provide details about the experimental setup used to collect data and train our models. We also provide performance comparison between different bytecode classification models and additional details about our models' parameter configuration process.

Hardware. We performed all data collection, as well as training and testing our models, on an Ubuntu 18.04 server with an Intel(R) Xeon(R) Silver 4110 CPU, three GeForce GTX 1080 Ti GPU and 64GB RAM. Data collection was performed using Puppeteer and our instrumented Chromium browser (version 100.0.4889.2) from a university network with an IP address located in the US. We used GNU Parallel [71] to run tasks in parallel.

Dataset breakdown. We collected 913,890 scripts from the top 50K websites and, after processing, obtained 396,370 scripts that generated unique bytecode sequences. The division between AT and

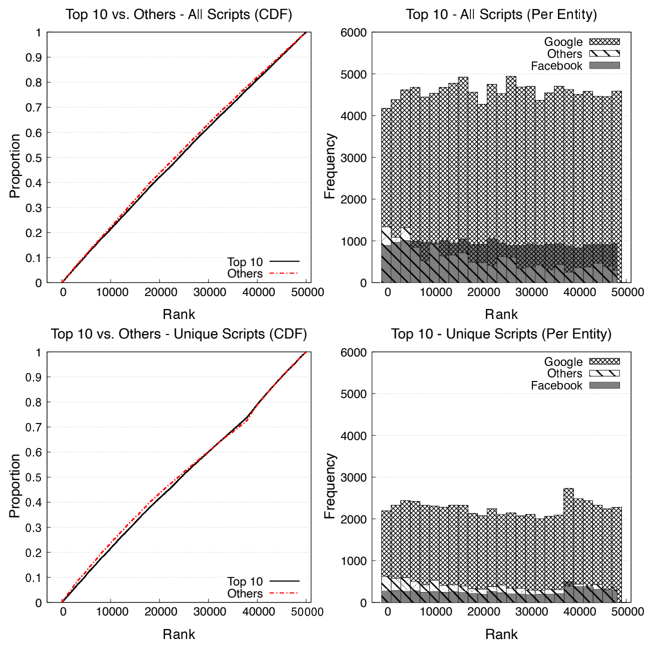


Figure 3: The top section includes all scripts and the bottom is plotted based on unique scripts. The left plots are comparison CDFs between observed (JavaScript) resource requests from the top 10 AT FQDNs in relation to other trackers over ranks. The right plots illustrate the dispersion of top 10 AT (FQDNs) across ranks when grouped by their entity.

benign JavaScript is shown in Table 2. The unique bytecodes will be used to train and test our models' performance (§5). To find the ratio of first- and third-party trackers, we compare the fully qualified domain names (FQDN) of the script URLs against the websites from where the requests were initiated. We note that 366,944 (99.5%) AT scripts are third-party resources whereas only 1,723 are first-party. This ratio is similar for unique scripts, with 129,819 (99.5%) being third-party as opposed to 690 from first-parties. If we use a more relaxed comparison using the apex domain (instead of FQDN), the third-party ratios of AT scripts are 359,083 (97.4%) and 126,288 (96.8%) for all and unique scripts respectively.

To better analyze the prevalence of tracking scripts, we group them based on their FQDNs. Table 3 shows the top 10 most frequent FQDNs that served AT JavaScript, which constitutes more than 50% of the trackers in our unique scripts data. We also note that more than half of the top 10 FQDNs belong to a single entity (i.e., Google). We then divide the top 10 tracking FQDNs into three separate entities: Google, Facebook, and others, and count their frequency throughout the data. Figure 3 shows the distribution of the FQDNs over different ranks. The first observation is that while the majority of trackers belong to Google, all three categories are dispersed uniformly across different ranks. Second, when we ignore the top 10 trackers we observe a similar distribution among ranks. In other words, our collected data is not biased towards a particular set of ranks (popular or unpopular) and is representative of AT-related JavaScript observed across different websites.

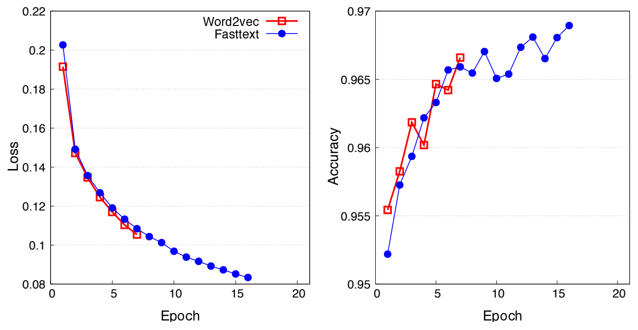


Figure 4: Loss and accuracy comparison of DPCNN training using different embeddings.

4.1 Model Parameters

DPCNN. In the original DPCNN paper [39], the authors used a 300-dim unsupervised embedding. The typical range for the dimension size is 50-300 [50]. However, because of the unique characteristics of our dataset which consists of very long sequences with a limited set of recurrent words, we opt for fewer dimensions as that is more efficient computationally. Therefore, to obtain the embedding, we train Word2vec with a vector size of 100 and a window size of 3. We do not set a vocabulary size and let the model choose the size based on the observed data. The default setting for Gensim's [54] implementation of Word2vec does not include words with a frequency less than 5 in the final vocabulary. We also use FastText with the same parameters to generate pre-trained unsupervised word embeddings to compare the performance of different embeddings. We use the vocabulary size, the vector size (100) and obtained embedding weights to create the (unsupervised) embedding layer. The embedding generated from Word2vec and FastText has 334 words.

In our experiments, we find that the maximum sequence of bytecode directly impacts the model's performance as the larger sequence can capture more global information. However, large sequences also require more GPU memory and negatively affect the computation time. With our experimental setup, we were able to fit bytecode with maximum sequence lengths of 280,000 words. We test different parameters by building on top of the parameters proposed in [55]. Initially, we considered a kernel size of 100 with a small pooling stride of 2 (which was used in the original model) and then gradually changed the parameters. We achieved the best results with a kernel size of 51 and a pooling stride of 15. We used the Adam optimizer [45] as the optimization algorithm. We experimented with different learning rates (e.g., 0.0005, 0.00075, 0.001, etc.) and as the learning rate increased, we observed more sensitivity and higher loss fluctuation. In general, lower learning rates may result in a more accurate model but can also cause slower convergence. Through extensive and systematic empirical analysis we found that the model performed well with a learning rate of 0.00075. Lastly, we set the dropout [69] to 0.2 to prevent overfitting.

FastText. We use the aforementioned parameters for Word2vec to obtain unsupervised word embedding. However, for the text classification task we set the word vector dimension to 100 and the learning rate to 1.0. We obtained the best results by setting the word n-gram to 10, the context window size to 10, and the training step to 150 epochs. Note that the word n-gram is set by passing the `wordNgram` argument, while the character n-gram (described in §3.4) is set by using the `maxn` and `minn` arguments in FastText's library. We use the default value for character n-gram parameters. Since the classes in our dataset are unbalanced, we used hierarchical softmax for the loss function [9]. We note that due to the large word n-gram and epochs, the training time is much higher compared to our Random Forest model, though less than DPCNN. We further explore prediction and training time comparison in §5 and §6.

Random Forest (RF). We used GridSearchCV [51] to find the best set of parameters for training a random forest classifier on our data. Our findings show that the best results are achieved using the Gini criterion, 200 estimators, and setting the number of features allowed for splitting the tree to the square root of all features.

5 EVALUATION

We perform two different sets of experiments to compare the performance of the models. In the first experiment, we compare the predictive power of models on the same set of test cases. Then we compare the generalizability of the models based on the filter lists released at a later time. For our performance comparison and manual analysis we use the following metrics:

(1) **False Positives:** JavaScript resources that are classified as AT by our model while they are labeled as benign by the filter lists.
(2) **True Positives:** Both our model and the filter lists classified these JavaScript resources as AT.
(3) **False Negatives:** JavaScript resources which are classified as benign by our model but included in the filter lists as AT.
(4) **True Negatives:** JavaScript resources classified as benign by our model while also not being included in the filter lists.

Given the above metrics, we explore the accuracy of classifying JavaScript resources (i.e., external scripts). In the first set of experiments, we used the collected data described in §3 to train and measure our models' performance. In models that use word embeddings, we use the embeddings generated by both Word2vec and FastText and compare the performance to understand which word embedding provides better results. Our goal is to compare the model's performance based on how the embedding is obtained and different model architectures for text classification.

After parsing the bytecode output, we split it into training (80%) and testing (20%) sets and *only use the training set* to train Word2vec and FastText and create the unsupervised embedding layer. For DPCNN, we replace the bytecodes with their corresponding indices in the embeddings. This data format along with the embedding will be used to train the DPCNN model. Bytecode sequences in our data have different lengths and the sequence inputs to DPCNN have to have the same length. Therefore, we limit the sequence length to the maximum length that we can fit in the GPU based on our experiment setup and pad sequences that are shorter with a padding index which has a vector of zeros in the embedding's look-up table. For FastText, we do not need to convert the bytecodes, we only make the compressed bytecode compatible with FastText's format and train the model. For RF, after replacing each bytecode with its corresponding vector, we calculate the mean of word vectors in each script and use it to train the classifier. Finally, we train and

Table 4: Model performance for script classification using different classifiers and embeddings. The FastText model refers to supervised text classification, while unsupervised learning was used to obtain the FastText embedding.

Model	Embedding	Accuracy(%)	Precision(%)	Recall(%)	F-1	Train Time (h)
DPCNN	Word2vec	96.65	96.07	93.66	94.85	140
	FastText	96.89	96.47	93.98	95.21	232
FastText	–	**97.08**	95.62	**95.49**	**95.56**	60
RF	Word2vec	96.48	98.61	90.58	94.42	**0.027**
	FastText	96.46	**98.64**	90.49	94.39	–

test the models and compare their performance on the same data. Because DPCNN and FastText require a significant amount of time to train, we only compared 5-fold cross-validation in the RF model and note that the results are similar and slightly better than the reported results in Table 4, which indicates that RF does not have a selection bias and generalizes well.

Figure 4 illustrates the loss and accuracy when using different embeddings in DPCNN; the kernel and stride were set to 51 and 15 respectively, and the max sequence length was set to 280,000. While Word2vec's embedding triggered the early stopping sooner, FastText's embedding resulted in lower loss and higher accuracy. The *upfront cost* for training is an important factor to consider, especially when adding a new set of bytecode to V8, or having additional samples that require re-training. Table 4 shows the performance of the classifiers as well as the trade-off between performance and the upfront cost. Training RF took less than 2 minutes and produced the lowest false positives compared to 60 hours training time for FastText which achieved the lowest false negatives. The results for DPCNN were in between these two models and the training time took >140 hours to complete. While no model achieved the best score in all metrics, FastText produced the best accuracy, recall and F-1 score compared to other classifiers.

Summary I: All three models performed comparably well in detecting AT scripts. Random Forest achieved the highest precision with the *lowest upfront cost*, while FastText had higher upfront cost, though lower than DPCNN, but achieved the highest overall performance.

5.1 Proactive Detection

We investigate the models' performance in detecting AT resources that are not yet present in the current version of filter lists, but are added in future versions. This experiment also examines how well the models are generalized in detecting new scripts. Filter lists such as EasyList and EasyPrivacy are frequently updated to include newly added rules for detecting unwanted resources or remove stale or incorrect rules that may cause website breakage. We downloaded updated versions of all filter lists 2 and 4 months after the initial download. We labeled the initial list, which was used for labeling the training and testing dataset, as list (A) and the subsequent lists as lists (B) and (C). We located the false positive cases of DPCNN and RF with Word2vec embedding (higher false positives) and FastText (See Figure 6), and tested them against the new filter lists. If they were detected as AT by the new lists, we downloaded their JavaScript file and compared the content (using `difflib` [4]) with the source code of the script we obtained during the data collection, to ensure that their inclusion in the updated filter lists was not due to modifications in the code. We also checked whether the scripts were detected as false due to being allowlisted in list (A), and whether the undetected URLs were identified as malicious by Virus Total [68].

Table 5: Proactive detection results by comparing false positives against new filter lists. Lists (B) and (C) were downloaded 2 and 4 months after the original filter list.

	List (B)				List (C)			
	Overall		Same Scripts		Overall		Same Scripts	
Model	Total	Malicious	Total	Malicious	Total	Malicious	Total	Malicious
DPCNN (w2v)	21	8	7	1	29	8	13	1
FastText	22	8	6	1	33	8	12	1
RF (w2v)	10	4	5	0	13	4	8	0

The results are reported in Table 5 and include both the total collection of scripts and the scripts where string matching reported at least a 90% similarity across script versions. The overall results also contain scripts where string matching was lower than 90% and the scripts where we received errors such as "404 Not Found" or "403 Forbidden". The malicious results are scripts that were detected as malicious at least by one source in Virus Total. None of the false positives were bypassed as a result of being allowlisted by list (A). We note that RF detected fewer scripts overall, which can indicate that the model is less generalized compared to FastText and DPCNN. Crucially, our findings show that these classifiers are able to proactively detect scripts that show AT behaviors. We also note that the number of true positives increased between two lists (2 months apart), indicating that the false positives reported in Table 4 are an overestimation.

We also performed a similar experiment for false negative samples to examine the proactive detection of allowlisted scripts. These are the samples that were labeled as AT by filter list (A) but detected as benign by our models. We checked these false negative samples against updated filter lists (B) and (C); if they were detected as benign by these lists we performed additional string matching and selected the ones with above 90% similarity. We found 28, 15, and 20 false negative samples for RF, FastText, and DPCNN respectively which were detected as true negatives when using the updated filter lists. These samples may have been added to the exception rules (or the initial rules were deleted) to potentially prevent usability issues. For instance, in the initial list, there was no specific rule for `palmettostatearmory.com`, whereas in the updated list (C) there was a rule that allows a website to send a request to `palmettostatearmory.com/static/`. We also found a similar case for `batteriesplus.com`. Evidently, both of these experiments show that the trained models are generalized well enough that our approach is able to go beyond simply detecting the current rules and proactively locate AT scripts that should be added or even removed (allowlisted) from the lists.

Summary II: All three models were able to uncover new scripts that had not been detected by the existing filter lists, with FastText detecting the highest number. The number of previously undiscovered scripts increased over time which suggests that our findings represent a lower bound.

Manual Inspection. To further examine the false positive, we randomly selected 30 (out of 334) samples from RF's false positives

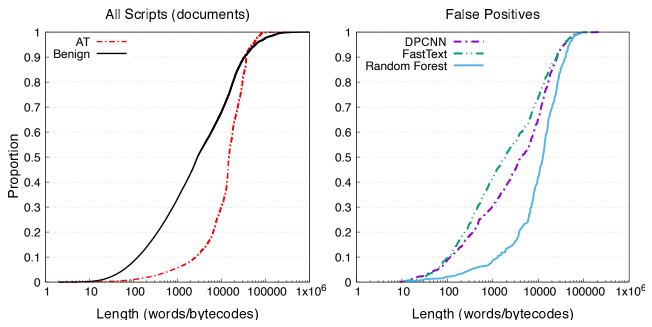


Figure 5: The left plot shows CDF of bytecode sequence length generated for AT and benign JavaScript code. The right plot provides a comparison between bytecode sequence lengths among the false positives.

(See Figure 6) and manually inspected their code to check whether an incorrectly detected script was actually a true positive by having code snippets related to tracking or advertisement behavior. We ensured that the selected samples are different from the samples reported in the proactive detection experiment. We used indicators such as having script comments (e.g., Tealium, Google Analytics) or JavaScript code with the purpose of collecting and sending tracking data (e.g., ad impressions). Lastly, we checked whether EasyList or EasyPrivacy had exception rules for these domains. Our manual analysis showed that out of 30 scripts, 23 (76%) scripts correctly detected by our model as tracking. Out of the 7 remaining samples, 5 did not have any discernible tracking code and 2 samples were being allowlisted in the filter lists. The manual inspection, as well as the proactive detection experiment, illustrate that our models can identify tracking scripts that are not present in the current filter lists.

We also examined the false positives from each model to investigate whether there were common characteristics among them. For instance, we examined whether there was a relationship between the length of the sequence and the misclassification of benign scripts by any of the models. To explore the sequence length's effect, we grouped each model's false positives based on the bytecode's sequence length and compared their distribution. The distribution for each model is shown in Figure 5. The left plot describes the (bytecode) length distribution of AT and benign scripts and the right plot shows the distribution only for the false positives. We note that RF's false positive cases, which had the highest precision, closely follow the length distribution of AT samples while FastText and DPCNN's false positives are closer to the benign class. We should note that when we only consider the shared cases (i.e., incorrectly classified by all models) the distribution is similar to RF's. Nevertheless, FastText and DPCNN misclassified more benign samples with shorter sequence lengths compared to RF. Therefore, in situations where the majority of samples have smaller bytecode sequences, the RF model may perform better in terms of precision. Even though Data URIs are challenging for request-based methods, our FastText model detected 3 Data URI cases that were incorrectly detected as benign by filter lists.

Detecting allowlisted scripts. Some AT scripts are added to filter lists' allowlists to prevent breakage and ensure that the website functions properly. While prior manual breakage analysis [37, 38, 60] can subjectively examine tools' performance from a user's perspective, it does not provide a systematic evaluation of a tool's ability in identifying allowlisted resources. As such, we created a new filter list by removing allowlisted rules. Next, we used both lists to identify allowlisted scripts. If a sample is detected as AT with the modified filter list (which does not have exception rules) but not the original list, we add it to the allowlisted samples. We checked all false positives detected by our models against the allowlisted samples. We call these false positive cases "allowlisted false positives", and report the results in Table 6. These are identified as false positives because they were explicitly allowed in the filter lists to bypass content blockers. We note that these scripts are a small portion of the 538 allowlisted samples. In other words, 538 cases were selectively bypassed by the filter lists for reasons such as breakage or causing usability issues and the models were able to correctly classify >90% of them. The most commonly occurring allowlisted false positive belonged to www.googletagmanager.com, which is also the most frequent FQDN in our data (Table 3).

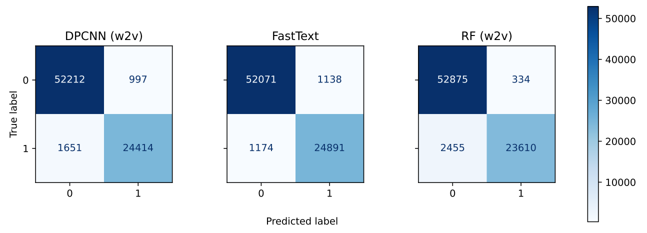


Figure 6: Confusion matrix comparison for the three models. For brevity, we only provide the confusion matrix for the models used in proactive detection. We use the "1" and "0" labels to represent AT and benign classes respectively.

Summary III: In addition to uncovering new AT scripts, all three models correctly identified more than 90% of the allowlisted scripts. The length of the sequence had a greater impact on DPCNN and FastText than RF, where false positives with shorter sequences were more prevalent.

5.2 Comparison to State-of-the-Art

In this section, we compare the classification accuracy of our approach to prior state-of-the-art machine learning-based techniques that specifically focused on identifying AT with high accuracy and varied in the type of features they used. Some of the techniques required new data collection. Therefore, did not consider methods that relied on (1) a special tool that was not publicly available, or required being built from scratch with significant effort (e.g., instrumenting a browser), or (2) a particular data collection method that was not available to us, such as collecting traffic from a large number of users [32]. Additionally, we only compared replicable methods where the features are *clearly* described in the paper.

Request-based. First, we look into prior work that used request-based features. We compare our bytecode classification model with BD+ [17] which is an offline method that applies machine learning to request-based features. We collected a new set of data for bytecode classification and BD+ by using OpenWPM [26] in parallel

Table 6: Allowlisted false positives.

Results \ Model	DPCNN (w2v)	DPCNN (ft)	FastText	RF (w2v)	RF (ft)
Allowlisted FP	48 (4.81%)	38 (4.24%)	47 (4.13%)	34 (10.17%)	33 (10.15%)
Total FP	997	895	1,138	334	325

with our instrumented Chromium and crawled 5K random websites ranked between 50K-60K (we choose a different range so as to obtain additional data than one used in previous experiments). We only visited the home page without interacting with the website similar to our large data collection. We call this dataset D1. In total, the shared data between both datasets has 42,927 samples with 29% of the samples belonging to AT scripts.

We performed a 5-fold cross-validation on the shared data and used the parameters proposed by [17] to train and test the models. We also made sure the data used for training and testing the models have the same ordering, to prevent any performance discrepancies. For BD+ we performed a k-Nearest Neighbors (kNN) with k = 5. For bytecode classification, we used a random forest (RF) model with FastText word vectors and in each iteration we trained the unsupervised embeddings (see §3.3) on the training samples prior to training, and then used it for vectorizing the training and testing samples and calculating the means. The results are reported in Table 7. We find that BD+ suffers from both high false positives and false negatives, which aligns with our intuition that classifiers that only rely on request-based features do not perform well as advertisers are incentivized to employ evasive techniques. BD+'s performance can also be attributed to its reliance on features that are not commonly found in JavaScript resources, such as whether the URL contains a semicolon or dimension information. Our bytecode classification achieved 27.62% higher precision and 11.22% higher accuracy than BD+. Having fewer false positives is particularly important in content blockers where the incorrect detection of scripts can cause breakage in websites.

JavaScript-based. We compare our method with the approach proposed by Kaizer and Gupta [41] that uses JavaScript-based features in addition to URL features, and WebGraph [60] that extends AdGraph [37]'s network layer by including storage information and data flow in the graph. [41] uses 18 features including 13 JavaScript Web API calls and 5 request-based features for classification. Since OpenWPM already collects the Web API calls from scripts, we used the same data collected for BD+ and selected the scripts that had at least one JavaScript API call, to ensure their execution trace was captured, and their matching requests and responses (which is used for URL features) where available. We found 20,309 samples that matched these conditions and overlapped with our bytecode dataset. We call this dataset D2. The AT samples constituted 46% of the data. The labeling approach described in [41] considers first-party external JavaScript requests as benign. However, using our ground truth filter lists (and focusing on apex domains) we found 681 first-party JavaScript samples from the original data, including 206 overlapping samples from D2 that belonged to the AT class. Therefore, we followed the same labeling approach explained in §3 for generating the ground truth. To provide a fair comparison, we perform a 5-fold cross-validation using a random forest classifier with 200 decision trees. In each fold, we trained an unsupervised

Table 7: 5-fold classification performance of bytecode classification compared to BD+ [17], Kaizer and Gupta [41], and WebGraph [60].

Data	Method	Precision(%)	Recall(%)	F-1	Accuracy(%)
D1	RF (FastText)	**98.61** ± 0.21	**81.41** ± 0.86	**89.18** ± 0.48	**94.32** ± 0.22
	BD+ [17]	70.99 ± 0.84	69.70 ± 1.79	70.33 ± 1.22	83.10 ± 0.59
D2	RF (FastText)	**98.34** ± 0.25	**88.06** ± 0.54	**92.91** ± 0.37	**93.75** ± 0.32
	Kaizer and Gupta [41]	87.17 ± 0.22	84.92 ± 0.69	86.03 ± 0.44	87.16 ± 0.37
D3	RF (FastText)	**98.96** ± 0.26	**74.82** ± 1.84	**85.20** ± 1.22	**92.62** ± 0.53
	WebGraph [60]	80.28 ± 1.59	68.44 ± 0.90	73.89 ± 1.19	86.25 ± 0.66

FastText embedding on the training set, used it to vectorize the training and testing samples, and then calculated the mean to obtain a vector representation for each script.

For WebGraph, we used the Docker image provided by the authors [15] and crawled the same 5K websites using the included OpenWPM alongside our instrumented Chromium browser. For extracting WebGraph's features, however, we were only able to process 1,500 websites due to the huge processing overhead (see §6). The pre-processing step was conducted using the default features provided in the WebGraph implementation, and the training and testing were done using the intersection of WebGraph's data and the collected bytecode. We call the shared dataset D3, which has 12,136 samples where 28% belong to the AT class. For both experiments, we used a random forest (RF) classifier. For bytecode classification, we used the FastText embedding to pre-process the data, and for WebGraph we used the parameters specified by the authors. We conducted a 5-fold cross-validation to compare the performance. Table 7 summarizes the cross-validation results where bytecode classification outperforms WebGraph across all measures, specifically in reporting fewer false positives with an 18% improvement in precision. Bytecode classification using D3 dataset had lower recall compared to D1 and D2 datasets, which can be attributed to the smaller sample size due to only processing 1,500 websites. We note that WebGraph, which outperforms AdGraph [37], reports a 4-9% improvement if content features are added. However, even then the precision is lower than what is achieved by our model while also decreasing WebGraph's robustness and making it more susceptible to evasion techniques.

Summary IV: Bytecode classification significantly outperforms existing methods, produces fewer false positives and false negatives while maintaining high accuracy in identifying AT scripts even in smaller datasets.

6 DISCUSSION

Here we further discuss pertinent aspects of our technique.

Upfront vs. downstream cost. To achieve the best results for the DPCNN model we utilized three discrete GPUs and required a week of training, whereas the FastText model only needed one CPU (with 16 cores) and less than three days for training. While the RF model that achieved the best precision only needed a few minutes to train, DPCNN and FastText generalize better (see §4). On the other hand, precision is an important metric due to the negative impact of false positives on the user experience. Overall, once the model is trained, pre-processing and classification time will remain the only (downstream) cost. Therefore, we view the training cost as an infrequent, one-time upfront cost that occurs when a new set of

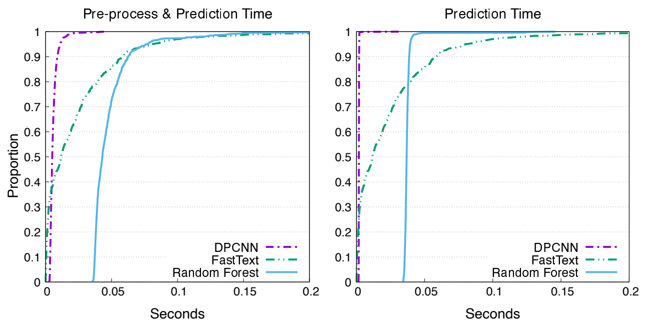


Figure 7: Comparison of prediction and pre-processing time per sample between different models. The left figure includes the pre-processing time in addition to the prediction time. Note that FastText does not have pre-processing overhead as the inputs are passed to the model directly.

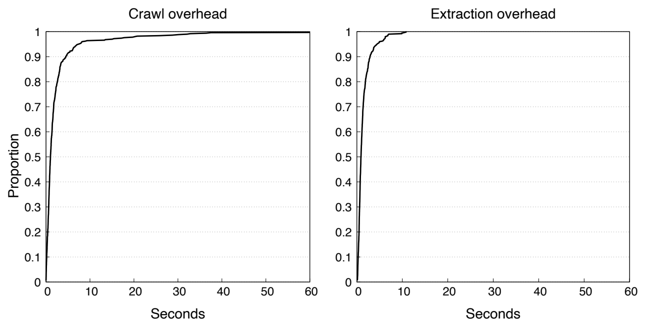


Figure 8: The added overhead for collecting bytecode from 500 websites using the instrumented Chromium as well as extracting bytecode.

bytecode is incorporated into V8, as opposed to the data collection and prediction performance.

We calculated the average pre-processing and prediction time for each sample based on 5 separate runs. The results are shown in Figure 7. While hardware requirement differences should be taken into account, we note that DPCNN and FastText are faster for a single sample classification compared to RF. However, performance can be vastly different and improved when batching is used. To measure the downstream cost of data collection, we selected 500 websites randomly from the top 10K and conducted a simultaneous crawl with and without printing the bytecode using our instrumented Chromium. We also set an upper bound of 120 seconds per website. We observed a mean and median of 2.58 and 1.01 seconds overhead respectively for saving the generated bytecode, and 1.4 and 0.91 seconds for bytecode extraction and processing. Figure 8 shows the CDF for the additional overhead incurred in crawling websites and storing bytecode, and subsequently extracting the bytecode. We compared our technique to a graph-based approach such as WebGraph [60], and the processing overhead using WebGraph was 345 seconds per website, which is significantly slower than our pipeline and consistent with the overhead reported by other studies [38]. While our method incurs a higher training cost, it outperforms prior approaches in terms of speed and accuracy when applied to large-scale experiments.

Robustness. Our approach is party-agnostic, does not rely on static features of the code or HTTP request information and is not vulnerable to code obfuscation techniques where previous approaches falter. Whether the loaded JavaScript resource is first or third-party is irrelevant to the model's ability for classifying the script's behavior. Using the party (i.e., origin) as a feature can introduce bias; for instance, while visiting `redbull.com`, our model identified a first-party script https://qm.redbull.com/analytics.js as tracking even though the URL did not match the filter lists' rules. Similarly, our model detected the https://www.footmercato.net/lib/prebid7.1.0.js script while visiting `footmercato.net`. Since the generated bytecode depends on how the website uses the script (execution), our method can differentiate between seemingly similar scripts (e.g., based on URL) originating from different websites. For example, the execution flow for a JavaScript library that contains both AT and benign functions will depend on which functions within the library are called at runtime, which results in having different bytecode streams for different websites even if they are using the same library. This variability in execution flows can be leveraged to distinguish allowlisted scripts from AT ones.

A model's robustness to evasion depends on its sensitivity to easily alterable features. In our approach, the input is a vector embedding learned from a bytecode sequence obtained from the script's execution. This is harder to manipulate since the embeddings capture the sequence's context. An adversary would need to create scripts that generate specific bytecode sequences where the average of its word vectors (for RF) would significantly change the model's prediction. This makes our approach more robust to adversarial evasion compared to models that rely on feature engineering.

Disclosure. We have disclosed the new resources identified during the manual inspection in §5 to EasyList's maintainers and are awaiting their response. Thus far, one of them has been accepted.

6.1 Limitations & Future Work

Data collection. We used a custom instrumented Chromium browser and OpenWPM to collect our data. However, some websites may use anti-crawling measures which can prevent automated browsers from accessing their content. This can limit the amount of data that can be collected, as well as potentially introduce bias in the collected data. Furthermore, the data received from the automated crawls of the homepage may differ from what real users receive through interactions with the websites which may affect the distribution of the collected data, particularly in websites that require user authentication or include dormant scripts [35] that do not execute during non-interactive page visits. However, prior research [77] also showed that compared to human browser users, crawlers experience more third-party activities. Moreover, interacting with the website and mimicking a real user's behavior can increase the execution coverage and mitigate the impact of dormant scripts. While such limitations are inevitable in large-scale studies, collecting data from a large number of websites can minimize the potential impact.

Evasion. While code obfuscation and URL randomization cannot evade our method, more challenging evasion techniques such as code bundling and inlining can impact dynamic analysis and circumvent filter-list-based techniques. Code bundling combines functional code with pieces of AT related code so that the website's functionality depends on allowing the external (bundled) script. Such resources often become allowlisted in filter lists (see §5). In code inlining, the AT code is directly wrapped inside the `<script>` tags instead of being loaded as an external script (i.e., `src` attribute of `<script>`) and, thus, cannot be blocked by filter lists even if they are detected by our model. Similar to prior work, we are still bound to analyzing a single bundled script as our goal is to identify externally loaded AT scripts and not to locate the parts of the script that exhibit tracking behavior. In other words, our method is an automated offline solution to detect external AT scripts and help craft rules for lists. If AT and benign code are bundled in one script, our method is not sufficient for locating parts of the script. To prevent function execution in inline or bundled scripts, a model has to classify *function bytecode* accurately, which have much smaller sequence lengths. This is very challenging, primarily due to the lack of high-quality ground truths especially when similar functions are prevalent among AT and benign scripts. Moreover, the models used in this work performed well on finding long-range associations in scripts' bytecode which do not have similar performance for functions' bytecode with much smaller sequence lengths. However, if a classifier accurately detects function bytecode sequences, an approach such as SugarCoat [64] can benefit from it for creating privacy-preserving function wrappers. We consider the exploration of language models suitable for function-level classification part of our future work.

Explainability. In recent years there has been an increasing interest in the ability to better understand (i.e., explain) the decisions reached by machine learning models. Certain models, such as decision trees, are easier to interpret as opposed to more complex models, especially those that rely on deep learning techniques (e.g., DNNs). Figure 9 (Appendix A.2) shows an example of the word importance obtained from the DPCNN model using Captum [3]. While such visualizations can provide some insights into these "blackbox" models, an interpretable system must also generate descriptions that are sufficiently simple and straightforward for a person to comprehend [30]. However, unlike other domains where explainability has garnered attention from researchers (e.g., specific regions in images, or specific words in natural language text), bytecode does not inherently convey sufficient meaning for a person so as to explain a model's decision making. Additionally, despite the clear explanation provided by RF's feature importance, the use of word vectors as features makes it difficult to interpret. Therefore, even though model interpretability methods exist, using bytecode as a feature poses challenges when it comes to providing human-understandable insight into the model's decision-making process.

non-JS requests. A limitation of our method is that we can only detect and craft rules for external JavaScript resources, as our primary goal is to improve the detection of AT scripts based on how the JavaScript interpreter observes them. For instance, we cannot detect tracking pixels that are statically included in HTML. Nevertheless, if a tracking HTTP request (e.g., pixel) is dynamically generated as a result of code execution, our approach can effectively detect the script from where the request originated.

New browsers & retraining. Since JavaScript bytecodes are not standardized and each browser has its own implementation, the trained classifier that works on one browser (e.g., Chromium) may not be able to effectively detect AT scripts when presented with the bytecode sequence captured by a different browser. However, this does not affect our overall objective since our goal is to augment the filter list creation process, and these lists are typically browser-agnostic. Nevertheless, exploring the effectiveness of bytecode classification across browsers is part of our future research direction. Additionally, as browsers evolve and new features are being added, the generated bytecode in the JavaScript engine can also change. This means that while the bytecode classification approach as a technique is applicable, different Chromium versions with a new set of bytecode words may require additional data collection and re-training of the models, which increases the upfront costs.

Ground truth. Lastly, we relied on popular filter lists to label our data. Even though EasyList and EasyPrivacy are widely used and frequently updated, there can be biases and inconsistencies in the lists (e.g., human error) that are also learned by the models. Nonetheless, using filter lists as the ground truth is common practice in prior studies [22, 37, 38]. Due to the scale, examining and manually labeling each script is not feasible. One solution would be to decrease the number of samples and create a smaller dataset through manual analysis. However, such an approach can be prone to more biases as the small sample size cannot cover the vast majority of websites and requires prior knowledge of the tracking practices used by the scripts. Summarily, as further proven by our manual analysis, our approach is highly accurate and our models are also generalizable and able to proactively uncover new scripts.

7 CONCLUSION

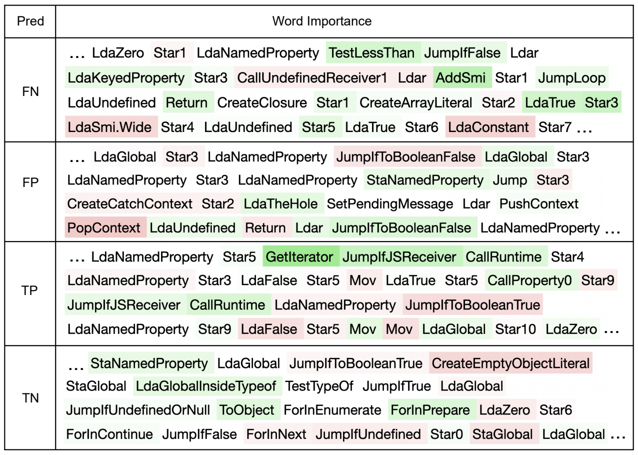
Content blockers rely on filter lists which are maintained by a community of users, and can provide many benefits to users such as reducing distractions and protecting their privacy and security. Maintaining the filter lists, however, requires a significant amount of manual effort for reviewing and updating the rules frequently, particularly due to the arms race between advertisers and adblockers. In this paper, we presented a novel and automated pipeline for accurately identifying unwanted scripts based on bytecode words obtained from the JavaScript engine. Our proposed technique is resilient against code obfuscation, captures code dependencies between inline and external scripts and can scale due to imposing minimal overhead. We show that in addition to detecting current scripts, bytecode classification uncovers currently-undetected AT scripts and, more importantly, can accurately classify allowlisted resources. While bytecode classification only relies on bytecode sequences as features, it outperforms the existing state-of-the-art detection systems that use request information, JavaScript features or a combination of both. To augment the filter list generation process and help the community of curators, we will release our system as an automated tool that can identify AT scripts in websites as well as determine which resources may need to be allowlisted.

ACKNOWLEDGEMENTS

We thank the reviewers for their helpful feedback. This project was supported by the National Science Foundation (CNS-2211574, CNS-2143363). The views in this paper are only those of the authors and may not reflect those of the US Government or the NSF.

REFERENCES

[1] 2023. . https://github.com/tofukko/filter/issues/24
[2] 2023. *Adblock Plus.* https://adblockplus.org/
[3] 2023. *Captum.ai.* https://captum.ai
[4] 2023. *difflib — Helpers for computing deltas.* https://docs.python.org/3/library/difflib.html
[5] 2023. *EasyList.* https://easylist.to/easylist/easylist.txt
[6] 2023. *EasyPrivacy.* https://easylist.to/easylist/easyprivacy.txt
[7] 2023. *Fanboy's Enhanced Tracking List.* https://www.fanboy.co.nz/enhancedstats.txt
[8] 2023. *FastText.* https://fasttext.cc/
[9] 2023. *FastText.* https://fasttext.cc/docs/en/faqs.html
[10] 2023. *hCaptcha.* https://www.hcaptcha.com/
[11] 2023. *How to write filters.* https://help.adblockplus.org/hc/en-us/articles/360062733293
[12] 2023. *jsdom.* https://github.com/jsdom/jsdom
[13] 2023. *Lazy parsing.* https://v8.dev/blog/preparser
[14] 2023. *Puppeteer.* https://github.com/puppeteer/puppeteer.git
[15] 2023. *WebGraph.* https://github.com/spring-epfl/WebGraph
[16] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. 2019. Errors, Misunderstandings, and Attacks: Analyzing the Crowdsourcing Process of Ad-Blocking Systems. In *Proceedings of the Internet Measurement Conference '19.*
[17] Sruti Bhagavatula, Christopher Dunn, Chris Kanich, Minaxi Gupta, and Brian Ziebart. 2014. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop (AISec '14).*
[18] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.
[19] Brave. 2023. *Ad Block engine in Rust.* https://github.com/brave/adblock-rust
[20] Brave. 2023. *Brave Browser.* https://brave.com
[21] Quan Chen, Panagiotis Ilia, Michalis Polychronakis, and Alexandros Kapravelos. 2021. Cookie Swap Party: Abusing First-Party Cookies for Web Tracking. In *Proceedings of The Web Conference (WWW).*
[22] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. 2021. Detecting Filter List Evasion with Event-Loop-Turn Granularity JavaScript Signatures. In *2021 IEEE Symposium on Security and Privacy (SP).* 1715–1729.
[23] Savino Dambra, Iskander Sanchez-Rola, Leyla Bilge, and Davide Balzarotti. 2022. When Sally Met Trackers: Web Tracking From the Users' Perspective. In *31st USENIX Security Symposium (USENIX Security 22).*
[24] Ha Dao, Johan Mazel, and Kensuke Fukuda. 2021. CNAME Cloaking-Based Tracking on the Web: Characterization, Detection, and Protection. *IEEE Transactions on Network and Service Management* 18, 3 (2021), 3873–3888.
[25] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. 2021. The cname of the game: Large-scale analysis of dns-based tracking evasion. *arXiv preprint arXiv:2102.09301* (2021).
[26] Steven Englehardt and Arvind Narayanan. 2016. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security.* 1388–1401.
[27] Aurore Fass, Michael Backes, and Ben Stock. 2019. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security.* 1899–1913.
[28] Imane Fouad, Cristiana Santos, Arnaud Legout, and Nataliia Bielova. 2022. My Cookie is a phoenix: detection, measurement, and lawfulness of cookie respawning with browser fingerprinting. In *PETS 2022-22nd Privacy Enhancing Technologies Symposium.*
[29] Mohammad Ghasemisharif, Peter Snyder, Andrius Aucinas, and Benjamin Livshits. 2019. SpeedReader: Reader Mode Made Fast and Private. In *The World Wide Web Conference (WWW '19).*
[30] Leilani H Gilpin, David Bau, Ben Z Yuan, Ayesha Bajwa, Michael Specter, and Lalana Kagal. 2018. Explaining explanations: An overview of interpretability of machine learning. In *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA).* IEEE, 80–89.
[31] gorhill. 2023. *uBlock Origin.* https://github.com/gorhill/uBlock
[32] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. 2015. An Automated Approach for Complementing Ad Blockers' Blacklists. *Proceedings on Privacy Enhancing Technologies* 2015 (02 2015).
[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition.* 770–778.
[34] Muhammad Ikram, Hassan Jameel Asghar, Mohamed Ali Kaafar, Balachander Krishnamurthy, and Anirban Mahanti. 2016. Towards seamless tracking-free web: Improved detection of trackers via one-class learning. *arXiv preprint arXiv:1603.06289* (2016).
[35] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2021. Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors. In *2021 IEEE Symposium on Security and Privacy (SP).* IEEE, 1143–1161.
[36] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. 2017. The ad wars: retrospective measurement and analysis of anti-adblock filter lists. In *Proceedings of the 2017 Internet Measurement Conference.* 171–183.
[37] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2020. Adgraph: A graph-based approach to ad and tracker blocking. In *2020 IEEE Symposium on Security and Privacy (SP).* IEEE, 763–776.
[38] Umar Iqbal, Charlie Wolfe, Charles Nguyen, Steven Englehardt, and Zubair Shafiq. 2022. Khaleesi: Breaker of Advertising and Tracking Request Chains. In *31st USENIX Security Symposium (USENIX Security 22).*
[39] Rie Johnson and Tong Zhang. 2017. Deep Pyramid Convolutional Neural Networks for Text Categorization. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).*
[40] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2017. Bag of Tricks for Efficient Text Classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2.*
[41] Andrew J. Kaizer and Minaxi Gupta. 2016. Towards Automatic Identification of JavaScript-Oriented Machine-Based Tracking. In *Proceedings of the 2016 ACM on International Workshop on Security And Privacy Analytics (IWSPA '16).*
[42] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. 2020. Carnus: Exploring the Privacy Threats of Browser Extension Fingerprinting.. In *In Proceedings of the 27th Network and Distributed System Security Symposium.*
[43] Soroush Karami, Faezeh Kalantari, Mehrnoosh Zaeifi, Xavier J. Maso, Erik Trickel, Panagiotis Ilia, Yan Shoshitaishvili, Adam Doupé, and Jason Polakis. 2022. Unleash the Simulacrum: Shifting Browser Realities for Robust Extension-Fingerprinting Prevention. In *31st USENIX Security Symposium.*
[44] Tom Kenter, Alexey Borisov, and Maarten De Rijke. 2016. Siamese cbow: Optimizing word embeddings for sentence representations. *arXiv preprint arXiv:1606.04640* (2016).
[45] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[46] Scott M Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.). Curran Associates, Inc., 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf
[47] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems.* 3111–3119.
[48] Nick Nikiforakis, Alexandros Kapravelos, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. 2013. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy.* IEEE, 541–555.
[49] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos Markatos. 2019. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *The World Wide Web Conference.* 1432–1442.
[50] Kevin Patel and Pushpak Bhattacharyya. 2017. Towards Lower Bounds on Number of Dimensions for Word Embeddings. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers).*
[51] 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. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
[52] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2018. Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156* (2018).
[53] Jeremy Poulain. 2023. *nlplay.* https://github.com/jeremypoulain/nlplay
[54] Radim Rehurek and Petr Sojka. 2011. Gensim–python framework for vector space modelling. *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic* 3, 2 (2011).
[55] Muhammad Fakhrur Rozi, Sangwook Kim, and Seiichi Ozawa. 2020. Deep Neural Networks for Malicious JavaScript Detection Using Bytecode Sequences. In *2020 International Joint Conference on Neural Networks (IJCNN).* IEEE, 1–8.
[56] Iskander Sanchez-Rola and Igor Santos. 2018. Knockin' on Trackers' Door: Large-Scale Automatic Analysis of Web Tracking. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc (Eds.). Springer International Publishing, Cham, 281–302.
[57] Yijun Shao, Stephanie Taylor, Nell Marshall, Craig Morioka, and Qing Zeng-Treitler. [n. d.]. Clinical Text Classification with Word Embedding Features vs. Bag-of-Words Features. In *2018 IEEE International Conference on Big Data.*
[58] Anastasia Shuba and Athina Markopoulou. 2020. Nomoats: Towards automatic detection of mobile tracking. *Proceedings on Privacy Enhancing Technologies* (2020).
[59] Anastasia Shuba, Athina Markopoulou, and Zubair Shafiq. 2018. NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, Vol. 2018.
[60] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. 2022. WebGraph: Capturing Advertising and Tracking Information Flows for Robust Blocking. In *31st USENIX Security Symposium (USENIX Security 22).*



Pred	Word Importance
FN	… LdaZero Star1 LdaNamedProperty TestLessThan JumpIfFalse Ldar LdaKeyedProperty Star3 CallUndefinedReceiver1 Ldar AddSmi Star1 JumpLoop LdaUndefined Return CreateClosure Star1 CreateArrayLiteral Star2 LdaTrue Star3 LdaSmi.Wide Star4 LdaUndefined Star5 LdaTrue Star6 LdaConstant Star7 …
FP	… LdaGlobal Star3 LdaNamedProperty JumpIfToBooleanFalse LdaGlobal Star3 LdaNamedProperty Star3 LdaNamedProperty StaNamedProperty Jump Star3 CreateCatchContext Star2 LdaTheHole SetPendingMessage Ldar PushContext PopContext LdaUndefined Return Ldar JumpIfToBooleanFalse LdaNamedProperty …
TP	… LdaNamedProperty Star5 GetIterator JumpIfJSReceiver CallRuntime Star4 LdaNamedProperty Star3 LdaFalse Star5 Mov LdaTrue Star5 CallProperty0 Star9 JumpIfJSReceiver CallRuntime LdaNamedProperty JumpIfToBooleanTrue LdaNamedProperty Star9 LdaFalse Star5 Mov Mov LdaGlobal Star10 LdaZero …
TN	… StaNamedProperty LdaGlobal JumpIfToBooleanTrue CreateEmptyObjectLiteral StaGlobal LdaGlobalInsideTypeof TestTypeOf JumpIfTrue LdaGlobal JumpIfUndefinedOrNull ToObject ForInEnumerate ForInPrepare LdaZero Star6 ForInContinue JumpIfFalse ForInNext JumpIfUndefined Star0 StaGlobal LdaGlobal …

Figure 9: Word importance examples obtained from the DPCNN model using Captum [3]. The colors indicate that the bytecode in that position had a positive (green) or negative (red) attribution to the predicted label.

[61] Suphannee Sivakorn, Angelos D Keromytis, and Jason Polakis. 2016. That’s the way the Cookie crumbles: Evaluating HTTPS enforcing mechanisms. In *Proceedings of the 2016 ACM on Workshop on Privacy in the Electronic Society.*

[62] Suphannee Sivakorn, Iasonas Polakis, and Angelos D Keromytis. 2016. The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In *2016 IEEE Symposium on Security and Privacy (SP).* IEEE, 724–742.

[63] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. 2019. Anything to hide? studying minified and obfuscated code in the web. In *The world wide web conference.* 1735–1746.

[64] Michael Smith, Pete Snyder, Benjamin Livshits, and Deian Stefan. [n. d.]. SugarCoat: Programmatically Generating Privacy-Preserving, Web-Compatible Resource Replacements for Content Blocking *(CCS ’21).*

[65] Peter Snyder, Antoine Vastel, and Ben Livshits. 2020. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking *(SIGMETRICS ’20).*

[66] Konstantinos Solomos, Panagiotis Ilia, Soroush Karami, Nick Nikiforakis, and Jason Polakis. 2022. The Dangers of Human Touch: Fingerprinting Browser Extensions through User Actions. In *31st USENIX Security Symposium.*

[67] Konstantinos Solomos, John Kristoff, Chris Kanich, and Jason Polakis. 2021. Tales of favicons and caches: Persistent tracking in modern browsers. In *Network and Distributed System Security Symposium (NDSS’21).*

[68] Gaurav Sood. 2021. *virustotal: R Client for the virustotal API.*

[69] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15, 56 (2014), 1929–1958.

[70] Jingxue Sun, Zhiqiu Huang, Ting Yang, Wengjie Wang, and Yuqing Zhang. 2021. A system for detecting third-party tracking through the combination of dynamic analysis and static analysis. In *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS).*

[71] O. Tange. 2011. GNU Parallel - The Command-Line Power Tool. *;login: The USENIX Magazine* 36, 1 (Feb 2011), 42–47. https://doi.org/10.5281/zenodo.16303

[72] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghwi Kwon, Xiangyu Zhang, and Patrick Eugster. 2016. WebRanz: Web Page Randomization for Better Advertisement Delivery and Web-Bot Prevention. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE).*

[73] Qianru Wu, Qixu Liu, Yuqing Zhang, Peng Liu, and Guanxing Wen. 2016. A Machine Learning Approach for Detecting Third-Party Trackers on the Web. In *ESORICS.*

[74] Zhiju Yang, Weiping Pei, Monchu Chen, and Chuan Yue. 2022. WTAGRAPH: Web Tracking and Advertising Detection using Graph Neural Networks. In *IEEE Symposium on Security and Privacy.*

[75] Zhiju Yang and Chuan Yue. 2020. A comparative measurement study of web tracking on mobile and desktop environments. *Proceedings on Privacy Enhancing Technologies* 2020, 2 (2020).

[76] Zhonghao Yu, Sam Macbeth, Konark Modi, and Josep M. Pujol. 2016. Tracking the Trackers. In *Proceedings of the 25th International Conference on World Wide Web (WWW ’16).*

[77] David Zeber, Sarah Bird, Camila Oliveira, Walter Rudametkin, Ilana Segall, Fredrik Wollsén, and Martin Lopatka. 2020. The representativeness of automated web crawls as a surrogate for human browsing. In *Proceedings of The Web Conference.*

A APPENDIX

This section provides additional information that is pertinent to our analysis and methodology.

A.1 EasyList Syntax

The rules in popular filter lists such as EasyList and EasyPrivacy follow a specific syntax that instructs the blockers when to block or allow a network request that matches the rules. While Adblock Plus provides detailed documentation on writing filters [11], we only address the allowlisted rules here as it is relevant to our work.

Listing 1 provides an example for two rules. The first line begins with a `||`, and specifies that this rule should be applied to any URL that matches `googletagmanager.com` pattern beginning from the domain name (e.g., supports both `http` and `https`). The caret symbol (`^`) acts a delimiter that separates the (domain) pattern and instructs the blocker to block the request regardless of what comes after the pattern, such as port, path or query. On the other hand, line 2, which begins with “@@”, indicates that this rule should be treated as an exception rule. The two pipelines serve the same purpose as before, while `$script` specifies that the rule only applies to script requests. Therefore, this is an exception rule for the rule presented in line 1. Lastly, `domain=okwave.jp` determines that this rule only applies to requests originating from `okwave.jp`. A rule can be applied to multiple originating domains, but they should be separated by a `|` symbol (e.g., `okwave.jp|example.com`). Understanding the structure of the exception rule helped us measuring the performance of our model in detecting allowlisted scripts (§5).

A.2 Explainability

There have been many studies on explaining how a machine learning model arrives at a particular output or decision, which has resulted in the creation of tools such as Captum [3] and SHAP [46] to help explain the output of a model. Figure 9 shows parts of four bytecode samples (taken from long sequences) next to their predictions by the DPCNN model. The word importance is obtained using Captum. Green and red colors illustrate the positive or negative attribution of the bytecode in that particular position to the predicted class. The intensity of the colors indicate the strength of the feature attributions. Such visualizations are helpful in understanding how features have participated in the outcome. However, interpretation is only useful (to humans) if the features are also understandable. For example, humans understand natural language; therefore, in a sentiment-classification task, highlighting words such as “bad” or “good” conveys that the model assigns weights to words similar to how humans use them. On the other hand, since bytecode is not understandable to humans, such visualizations do not convey a meaningful explanation, especially in cases where sequences are extremely long.