

WEBGRAPH: Capturing Advertising and Tracking Information Flows for Robust Blocking

Sandra Siby
EPFL

Umar Iqbal
The University of Iowa

Steven Englehardt

Zubair Shafiq
University of California, Davis

Carmela Troncoso
EPFL

Abstract

Millions of web users directly depend on ad and tracker blocking tools to protect their privacy. However, existing ad and tracker blockers fall short because of their reliance on trivially susceptible advertising and tracking content. In this paper, we first demonstrate that the state-of-the-art machine learning based ad and tracker blockers, such as ADGRAPH, are susceptible to adversarial evasions deployed in real-world. Second, we introduce WEBGRAPH, the first graph-based machine learning blocker that detects ads and trackers based on their *action* rather than their content. By building features around the actions that are fundamental to advertising and tracking – storing an identifier in the browser, or sharing an identifier with another tracker – WEBGRAPH performs nearly as well as prior approaches, but is significantly more robust to adversarial evasions. In particular, we show that WEBGRAPH achieves comparable accuracy to ADGRAPH, while significantly decreasing the success rate of an adversary from near-perfect under ADGRAPH to around 8% under WEBGRAPH. Finally, we show that WEBGRAPH remains robust to a more sophisticated adversary that uses evasion techniques beyond those currently deployed on the web.

1 Introduction

Users rely on privacy-enhancing blocking tools to protect themselves from online advertising and tracking. Many of these tools—including uBlock Origin [1], Ghostery [2], Firefox [3, 4], Edge [5], and Brave [6]—rely on manually curated filter lists [7, 8, 9] to block advertising and tracking. The research community is actively developing machine learning (ML) approaches to automate the detection of advertising and tracking and make filter lists more comprehensive. The first generation of ML-based blocking approaches analyze network requests [10, 11, 12] or JavaScript code [13, 14, 15] to learn distinctive behaviors of advertising and tracking. However, these ML-based blocking approaches are highly susceptible to adversarial evasion techniques that are already found in the wild,

including URL obfuscation [16] and code obfuscation [17]. To address this limitation, the next generation of ML-based blocking approaches leverage cross-layer graph information from multiple layers of the web stack [18, 19]. These approaches claim better robustness to evasion, as compared to single-layer approaches, due to their use of *structural* features of the graph (i.e., the hierarchy of resource inclusions) in addition to traditional *content* features (i.e., the resource’s network location or response content).

In this paper, we show that state-of-the-art ad and tracker detection approaches, such as ADGRAPH [18], are susceptible to adversarial evasion due to their disproportionate reliance on easy-to-manipulate content features. We show that a third-party adversary can achieve 8% evasion success by manipulating URLs of its resources. Worse yet, an adversary can achieve near-perfect evasion—as high as a 96% success rate—if they collude with the first party, e.g., by using the CNAME cloaking technique already deployed by some trackers [20, 21].

We introduce WEBGRAPH, the first ML-based ad and tracker blocking approach that *does not rely on content features*. WEBGRAPH improves the cross-layer graph representation by capturing a fundamental property of advertising and tracking services (ATS): the flow of information from one entity to the browser’s storage, the network, and to other entities loaded on a page. The intuition behind adding these features is to focus on the *actions* of the advertising and tracking services, rather than the *contents* of their resources. We posit that actions are harder to obfuscate. Advertising and tracking scripts need to generate and store identifiers for users, and those identifiers must be shared with any other entity with which they wish to share data (e.g., via cookie syncing [22]). Ultimately, if a script wishes to store an identifier in the browser, it will need to call a browser API, and as such, we monitor the flow of information to and from browser APIs. We build a graph representation of the page load by monitoring network requests, JavaScript execution, HTML element creations, and browser storage access. From this graph we extract *flow* features, which explicitly capture distinctive information flows in advertising and tracking. Our evaluation shows that WEBGRAPH’s graph

representation and flow features can entirely supplant content features, with comparable accuracy.

While high accuracy is necessary for deployment, it is not sufficient. We have repeatedly seen that advertisers and trackers will attempt to circumvent detection and evade blocking [16, 17, 20]. Therefore, in order for an advertising and tracking classifier to be useful in practice, it must be robust to adversarial manipulation. We show that WEBGRAPH represents a significant step forward in robustness to adversarial evasion when compared to previous approaches. In particular, we find that WEBGRAPH is robust to the types of URL, CNAME, and content manipulation evasion techniques that are in use on the web today. We also know that ad and tracking adversaries will attempt to deploy more sophisticated evasion techniques tailored to our classifier. To understand how robust WEBGRAPH would be in the face of these new evasion techniques, we propose a novel realistic graph manipulation evasion technique. We show that this attack achieves only limited evasion success against WEBGRAPH, while incurring a non-trivial usability loss in terms of mistakenly blocking its own advertising/tracking resources or other benign resources on the web page.

Overall, our findings suggest that the community should migrate away from unreliable content features for advertising and tracking blocking. We show that information flow features built upon the actions of advertisers and trackers provide a promising path forward.

In summary, our contributions are as follows:

- We show that existing ML-based ad and tracker detection approaches are susceptible to evasion due to their heavy reliance on content features. As a representative example, we show how an adversary can achieve near-perfect evasion of ADGRAPH using evasion techniques already in use on the web today.
- We introduce WEBGRAPH, the first ML-based ad and tracker blocking approach that does not rely on content features and captures fundamentally distinctive information flows in advertising and tracking.
- Our in-depth evaluation shows that WEBGRAPH achieves comparable accuracy to prior approaches and achieves significantly better robustness to adversarial manipulation of content features.
- We propose a novel graph manipulation evasion technique, and show that WEBGRAPH (and the information flow features it relies on) remain robust under this sophisticated attack.

Paper organization: The rest of this paper is organized as follows: Section 2 provides an overview of recent advance in ML-based ad and tracker blocking. Section 3 evaluates robustness of existing graph-based approaches, using ADGRAPH as a representative example. Section 4 describes the design and evaluation of WEBGRAPH. Section 5 further evaluates WEBGRAPH’s robustness to adversarial attacks. We discuss limitations of our work in Section 6 and conclude in Section 7.

2 Background & Related Work

Online behavioral advertising enables ad targeting based on users’ interests and behaviors. To target ads, online advertising relies on the intertwined tracking ecosystem that uses cookies for cross-site tracking. For instance, the real-time bidding (RTB) protocol that powers programmatic online advertising has built-in mechanisms for advertisers and trackers to share information [22, 23]. Thus, almost always, ads and trackers go together, often with intertwined execution flows and resource dependencies. Below, we revisit prior literature on ad and tracker blocking, and analyze its limitations.

Popular ad and tracker blocking tools such as Adblock Plus [24] rely on filter lists [7, 8]. These filter lists are manually curated based on user feedback. Prior work has shown that manually curated filter lists suffer from *scalability* and *robustness* issues. First, filter lists have trouble keeping up with the ever expanding advertising and tracking ecosystem. Filter lists have grown to include tens of thousands of rules that are often not updated in a timely fashion. For instance, prior work showed that filter lists may take as long as 3 months to add rules for newly discovered ads and trackers [25]. Once a filter rule is added to block an advertising and tracking service, it is rarely removed, even if it is no longer needed. In fact, prior work showed that almost 90% of the rules in filter lists are rarely or never used [16]. Second, filter lists are not robust to evasion attempts by advertisers and trackers. Filter lists are brittle in the face of domain rotation [26, 27] and manipulation of page structure [28, 29, 30]. For instance, prior work showed that filter lists are susceptible to evasion attacks such as randomization of URL path, hostname, or element attributes and IDs [31, 32].

Addressing scalability. To address the scalability issues that arise due to manual curation of filter lists, researchers have proposed to use machine learning (ML) for automated ad and tracker blocking. Prior ML-based approaches mainly detect ads and trackers at the network and JavaScript layers of the web stack. Specifically, these approaches detect ads and trackers by featurizing network requests [10, 11, 12] or JavaScript code [13, 14, 15].

Network layer approaches rely on content in URLs, HTTP headers, and request and response payloads (e.g., keywords, query strings, payload size) to extract features and train ML models to detect ads and trackers [10, 11]. While trying to mimic filter lists by detecting ad and tracker URLs, these approaches end up replicating some characteristics of filter lists and thus also naturally inherit their shortcomings. For example, presence of a certain keyword in the request URL could be a distinguishing feature. However, as discussed earlier, such keyword based features are brittle in the face of trivial evasions such as domain rotation [31, 32].

JavaScript layer approaches rely on static or dynamic analysis to extract features and train ML models to detect ads and trackers. Examples of features are n-grams of code state-

ments obtained via static analysis [15] or JavaScript API invocations captured via dynamic analysis [13]. These approaches are susceptible to JavaScript obfuscation [33, 34, 35]. These approaches are also susceptible to evasion such as script amalgamation or dispersion. They implicitly assume that tracking code is bundled in a single script or that tracking scripts only contain tracking code. However, in practice, tracking code could be distributed across several chunks and packaged with functional code [18].

Addressing robustness. While network and JavaScript layer approaches consider information at each layer in isolation, ads and trackers rely on all three layers (i.e. network, JavaScript, and HTML) of the web stack for their execution. Therefore, it is natural that focusing on only one layer lacks robustness against the aforementioned evasion attempts. To address this limitation, graph-based approaches aim to capture the interactions among and across network, JavaScript, and HTML layers of the web stack.

Graph-based approaches extract features from the cross-layer graph representation to train ML models to detect ads and trackers [18, 19]. These approaches leverage rich cross-layer context and thus claim to be robust to evasion attempts. ADGRAPH was the first graph-based approach to ad and tracker classification [18]. It extracts structural features from the graph such as node connectivity and ancestry information as well as content features such as URL length and presence/absence of certain keywords. Sjösten et al. [19] introduced PageGraph, which extends ADGRAPH’s graph representation by improving event attribution and capturing more behaviors. In addition to content and structural features, they also added perceptual features to train the classifier. Since *perceptual* features attempt to use the rendered resource content, they are also considered content features. Chen et al. [36] proposed an approach, using PageGraph, to detect trackers based on their execution signatures. In contrast to ML-based approaches, their signature-based approach would only be able to detect trackers that strictly match the signatures of tracking scripts, but miss trackers with even slight deviations in their behavior, such as changes in the execution order. Kargaran et al. [37] followed a different approach. Instead of building a graph representation per website, they combined graph representations across multiple websites to model relations between third parties on those sites. Just like ADGRAPH, they also extract structural and content features from the graph to train the classifier.

These graph-based systems use a combination of content and structural features for classification, which they claim increases the robustness to evasion attacks. While this combination should intuitively improve classifier robustness, we posit that it would be less robust than expected if the classifier relies heavily on content features. This is because content features pertain to a single node on the graph and are easy to manipulate for an adversary, e.g., using adversarial attacks on textual [38] and perceptual [39] content features, without causing undesired changes in other nodes. It is noteworthy that Zhu et

al. [38], also manipulate structural features, however their manipulations are only limited to graph size. Further, they do not evaluate the impact of their mutations on overall graph.

In the next section, we analyze the robustness of graph-based ad and tracker detection systems. We focus on ADGRAPH as it is representative of other graph-based systems that use similar structural and content features.

3 ADGRAPH Robustness

In this section, we analyze ADGRAPH’s robustness by evaluating its accuracy in the face of adversarial content manipulation.

ADGRAPH is a graph-based machine learning approach that detects ads and trackers based on their structural and content properties. ADGRAPH instruments the Chromium web browser to capture detailed execution of ads and trackers across the HTML, JavaScript, and the network layer, and models the interaction among these layers in the form of a graph. Using this graph, ADGRAPH extracts two categories of features: *content* (information related to individual nodes in the graph, such as URL length and presence of ad/tracking keywords in the URL) and *structure* (information about relationships between nodes, such as connectivity and ancestry information). It uses the extracted features to train a machine learning classifier to detect advertising and tracking resources. The full list of ADGRAPH features are described in Table 4.

Since ADGRAPH relies on content properties, in addition to structural properties, it is subject to same evasion attacks that succeed against the filter lists-based ad and tracker detection approaches [31, 32].

3.1 Threat Model & Attack

Our threat model assumes an adversarial third-party advertiser or tracker embedded on a site, who aims to change the classification of its resources from advertising and tracking services (ATS) to benign resources (Non-ATS) in order to evade detection by ad and tracker blocking tools.

We assume that the adversarial third party has limited cooperation with the first-party publisher. We do not assume full cooperation because the parties are mutually distrusting. The third-party adversary generally does not trust the first-party publisher to serve its advertising and tracking resources via a reverse proxy [40, 41]. Likewise, the first-party publisher does not trust the third-party adversary to host functional resources via the adversary-controlled CDN [42]. Given existing practices, we assume that the adversary can serve its advertising and tracking resources from a first-party subdomain but not arbitrarily within the first-party domain space. For example, the adversary can masquerade its resources through CNAME cloaking [43], which only requires a minor change in DNS records by the first party. Recent measurement studies have reported an increase in the prevalence of CNAME cloaking over the last few years. Dao et al. [20] showed that the usage

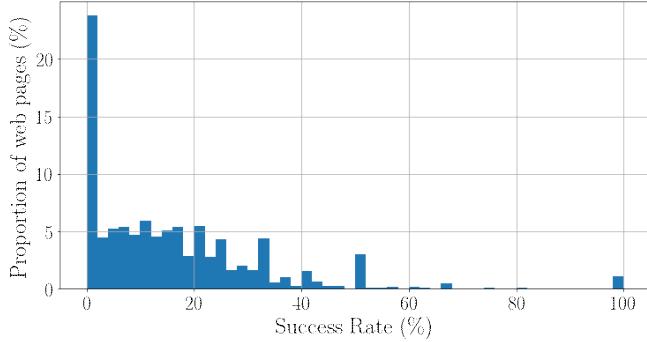


Figure 1: Classification switch success rate distribution by web page (over 10 folds) when the adversary does **not** collude with the first party. The average success rate per web page is $15.92 \pm 0.03 \%$.

of CNAME cloaking-based tracking has steadily increased between 2016 and 2020, with 1,762 of Alexa’s top-300K websites employing at least one CNAME-based tracker as of January 2020. Dimova et al. [21] also showed that the usage of CNAME cloaking has increased by 22% from 2018 to 2020, with 9.98% of Tranco’s top-10K websites now employing at least one CNAME-based tracker as of October 2020.

We assume that the adversary is able to manipulate their own URLs by altering the domain name or query string. Naturally, the adversary can only manipulate URLs that are under their control, and only attempts to manipulate the subset of its URLs that were initially correctly classified as ATS (ad and tracker URLs initially classified as Non-ATS already benefit the adversary). The adversary cannot manipulate the data used to train the classifier. Therefore, we only implement mutations during inference.

We implement two types of URL manipulations. For domain names, we allow the adversary to randomly change the URL’s domain, subdomain, or both. In practice, adversaries can rely on automated techniques to generate random domains and sub-domains. For example, they can use malware-inspired domain generation algorithms (DGA) techniques to generate a large number of domains [44, 45]. For query strings, we randomly change the number of parameters, the parameter names, the parameter values in the URL, or a combination of the three.

3.2 Results

Experimental setup. We extend OpenWPM [46] to automatically crawl websites with Firefox and build ADGRAPH’s representation. We crawl 10K sites sampled from the Alexa’s top-100K list, the top 1K sites and a random sample of 9K sites ranked between 1K-100K, and store their graph representations. Next, we implement a decision tree classifier that closely follows ADGRAPH’s design [18], and extract features from the graphs for training and testing. For ground truth, we use the same set of filter lists for data labeling that were used

by ADGRAPH [18]. A URL is labeled as ATS if it is present in one or more of the filter lists, and Non-ATS otherwise. We use 10-fold cross validation to obtain our results, where the folds are selected such that every fold uses a different set of web pages in the test set. Our classifier obtains comparable performance to the original results reported by [18]: 92.33% accuracy, 88.91% precision, and 92.14% recall. The minor differences are likely due to differences in crawled sites, updated filter lists, and a few subtle changes in our adaptation of ADGRAPH from online to offline. In ADGRAPH’s online implementation, features are extracted from each node in the graph as they are created. Our offline adaptation, instead, extracts features after page load completion. There are also some minor differences due to JavaScript attribution, caused by the differences in instrumentation between Chromium-based ADGRAPH and Firefox-based OpenWPM.¹

Adversarial success rate without collusion. In our first experiment, we assume that the adversary does not collude with the first party. The adversary can randomize their domain and subdomain, but cannot masquerade as the first party. Our content mutation procedure results in the mutation of $41.48 \pm 1.47 \%$ of all the test data URLs (averaged over 10 folds). The adversary’s success rate in evading the classifier is $8.72 \pm 0.42 \%$ (over 10 folds). While this may seem like a low percentage, we note that every successful mutation is a win for the adversary since it means that one more of their ads or trackers is now unblocked. Over all 10 folds, the adversary mutated 691,602 URLs, out of which 60,270 had their classifications switched.

We also observe that the evasion success rate varies across sites, as shown in Figure 1. For $\approx 1\%$ of the web pages in the test set (90 pages), the adversary achieves a perfect success rate, meaning that all third-party ads and trackers on the web page are now classified as benign content. It is noteworthy that 21.62% of the unblocked URLs belong to popular ad exchanges, which are responsible for further diffusion of user information due to the broadcast nature of real-time bidding (RTB) [47]. These unblocked ad exchanges can amplify the privacy harm because they often share information about page visits with multiple advertisers and trackers.

Adversarial success rate with collusion. In our second experiment, we assume that the adversary colludes with the first party. The adversary can perform domain mutation such that their URL is a subdomain of the first party. The adversary’s success rate increases to $96.62 \pm 0.37 \%$ (over 10 folds). This means that being able to use a first-party subdomain provides almost perfect evasion capabilities. Figure 2 shows the evasion success rate variation across sites. For $\approx 50\%$ of the web pages in the test set, the adversary achieves a perfect success rate. We also see a higher proportion (32.25%) of the unblocked

¹Due to these differences, our features are not exactly identical to the online implementation of ADGRAPH. For example, in ADGRAPH, a node can have a maximum of two parents, which need not be the case for our system. Therefore, we do not use ADGRAPH features specific to these two parents. The full feature list, showing these differences is provided in Appendix A.

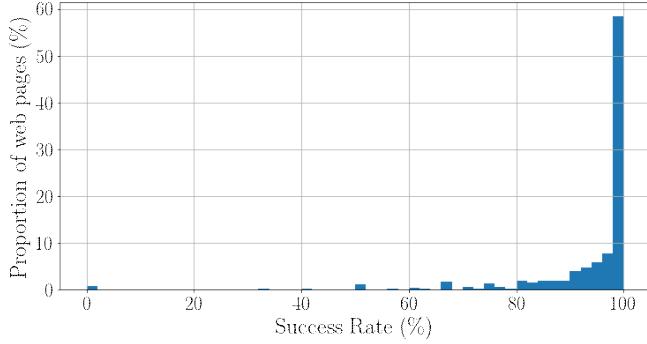


Figure 2: Classification switch success rate distribution by web page (over 10 folds) when adversary colludes with the first party. The average success rate per web page is $93.01 \pm 0.01\%$.

Feature	Category	Information Gain (%)
URL length	Content	14.87 ± 0.36
URL domain is a subdomain of the first party	Content	11.06 ± 1.24
URL is a third party	Content	10.67 ± 1.32
Degree of a node	Structure	7.56 ± 0.63
Number of edges divided by number of nodes	Structure	7.48 ± 0.41

Table 1: Top 5 most important features for ADGRAPH’s classification, their category, and information gain values (averaged over 10 folds).

URLs belonging to popular ad exchanges, as compared to the previous experiment.

To better understand why such URL manipulation is able to evade detection by ADGRAPH, we analyze feature importance using information gain (see Table 1). We see that content features are essential to the ADGRAPH classifier: not only are the top-3 most important features content features, their relative importance scores are also high compared to the other features. Two of the top-3 features depend on whether a URL is third-party, which explains why we obtain high success rates when the adversary has the capability to masquerade as the first party. These two features do not have an effect in the case where the adversary does not collude with the first party, since the adversary cannot change the fact that they are third party. However, the adversary’s manipulations still influence the third top feature, length of the URL. Hence, we observe lower but non-trivial success rates even without collusion.

These results show that graph-based classifiers such as ADGRAPH are vulnerable because of their over-reliance on content features. In the next section, we propose an approach to improve the robustness of graph-based ad and tracker blocking tools.

4 WEBGRAPH

Online advertising and tracking fundamentally relies on information sharing. Trackers need to share information with each other to improve their coverage of users’ browsing history [22, 46]. Trackers also need to share information with each

other as part of built-in dependencies in programmatic advertising protocols [48, 49]. We contend that leveraging such fundamental information sharing patterns can help build accurate and robust classifiers for ad and tracker blocking. We introduce WEBGRAPH, a classifier that explicitly captures these information sharing patterns as part of its cross-layer graph representation of the execution of a web page.

To illustrate the information sharing patterns that we want to capture in WEBGRAPH, let us revisit how information sharing between different origins is mediated by the browser. We deliberately use a loose definition of origin. An origin can be, depending on the specific use case, a site, a domain, or an entity, among others. At a high-level, the web browser isolates different origins, based on various policies, so that their data is not leaked to each other. Figure 3(a) illustrates how the browser limits information sharing between different origins: example.com, tracker1.com, and tracker2.com each have access to their isolated local storage (e.g., cookies, IndexedDB) that may be used to store user identifiers. The browser isolates information flows between the local storage and remote servers of different origins: tracker1.com and tracker2.com cannot generally access each others’ cookies.

Trackers typically circumvent these limitations in the browser in two main ways. First, Figure 3(b) illustrates how a tracker may share its identifier with another tracker through cookie syncing. This can be implemented in several ways. For example, let’s say example.com loads a JavaScript from Tracker 1 that first uses `document.cookie` to retrieve Tracker 1’s identifier cookie from its cookie storage and then initiates a GET request to Tracker 2. The script includes Tracker 1’s identifier cookie in the request URL as a query string parameter. Note that the request automatically includes Tracker 2’s identifier cookie in the Cookie header. Therefore, when Tracker 2’s remote server receives the request, it would be able to sync Tracker 1’s identifier with its own identifier. As another example, let’s say example.com first loads an invisible pixel from Tracker 1, which responds back with a 3XX redirect status code along with the URL in the Location header that points to Tracker 2 and includes Tracker 1’s identifier cookie. Upon receiving the response, the browser issues a GET request to Tracker 2 and includes Tracker 1’s identifier cookie in the request URL and Tracker 2’s identifier cookie in the Cookie header. Again, Tracker 2’s remote server is able to sync Tracker 1’s identifier with its own identifier.

Second, Figure 3(c) illustrates how a tracker may share its identifier with another tracker through various JavaScript APIs, in several ways. For example, let’s say example.com loads scripts from Tracker 1 and Tracker 2 which then share their identifiers by reading/writing to the global variables of the `window` object. The script from Tracker 1 may assign its identifier to a new global variable `foo` that is then read by the script from Tracker 2. Therefore, Tracker 1 and Tracker 2’s scripts would be able to sync identifiers with each other and also send them to their respective remote servers. As an

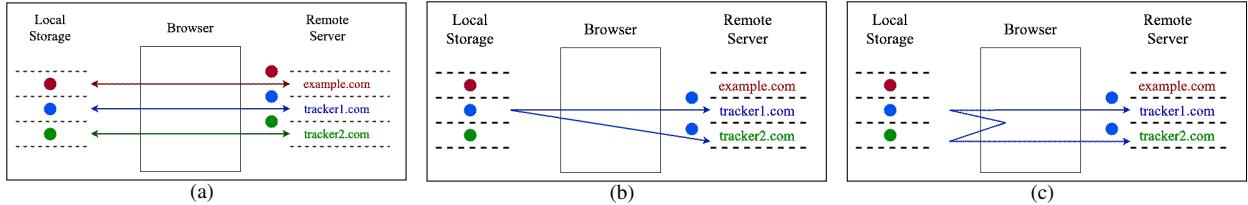


Figure 3: Origin isolation vs. sharing. Circles represent information about a user gathered by a particular domain (example.com, ●; tracker1.com, ●; and tracker2.com, ●). The box represents the browser which acts as channel between the local storage on the user’s device and the remote server of each domain. 3(a) Illustrates origin isolation in the browser: every domain can only access information in their own storage. 3(b) and 3(c) illustrate two information sharing patterns that trackers use to circumvent origin isolation: (b) cookie syncing, where users’ identifiers are sent to more than one domain; and (c) sharing identifiers using JavaScript APIs.

other example, let’s say example.com loads iframes from Tracker 1 and Tracker 2 which then share their identifiers using `postMessage`. While these iframes have different origins, Tracker 1’s iframe can use `window.parent` property to get a reference to the parent window and then use `window.frames` to get a reference to Tracker 2’s iframe. Tracker 1’s iframe can then use this reference to call `window.postMessage` and send its identifier to Tracker 2’s iframe, which can use `window.addEventListener` to receive the identifier. Tracker 2’s iframe can then send the shared identifier with its remote server to sync them.

Trackers use a wide variety of information sharing patterns, beyond the two aforementioned mechanisms. A sound and precise examination of all patterns warrants full-blown information flow tracking that adds significant implementation overheads and complexity [50, 51, 52]. As we discuss next, WEBGRAPH approximately² captures these information sharing patterns by including additional nodes and edges in its graph representation that correspond to elements and actions associated with these information sharing patterns. It then extracts new features on this enriched graph representation to train a classifier for detecting ads and trackers.

4.1 Design & Implementation

4.1.1 Graph Construction

WEBGRAPH captures the flow of information among and across the HTML, network, JavaScript, and storage layers of the web stack. At the HTML layer, WEBGRAPH captures creation and modification of all HTML elements, e.g., `iframe`, that are initiated with scripts. At the JavaScript layer, WEBGRAPH captures the scripts’ interaction with other layer, e.g., initiation of a network request. At the network layer, WEBGRAPH captures all outgoing network requests and their responses. At the storage layer, WEBGRAPH captures read/write in cookies and local storage through scripts and network re-

quests, and also the exchange of values between network requests.

OpenWPM Instrumentation. We extend OpenWPM [46] to capture the execution and interaction of HTML, network, JavaScript, and storage layers. To capture HTML elements creation and modifications, we instrument `createElement` method and register a `MutationObserver` interface. To capture network requests, we parse OpenWPM’s existing instrumentation, which uses a `webRequests` listener³, to capture all of the network requests, their responses, and redirects. To capture JavaScript interaction, we parse OpenWPM’s existing instrumentation, which relies on JavaScript’s stack trace to log JavaScript execution. To capture read/write to storage, we instrument `document.cookie` and `localStorage` methods and also intercept `cookie` read/write HTTP headers.

Graph Composition. Elements at each of the layers are represented with nodes and the interaction between these nodes is represented with edges. Specifically, each HTML element, network request, script, and stored value, is represented as a node. Edges to HTML nodes from script nodes represent the creation and modification of elements. Edges from HTML nodes to network nodes represent initiation of network requests to load content, such as scripts and images. Edges from script nodes to network nodes represent the initiation of `XMLHttpRequest` which will be parsed by the script. Edges between script and storage nodes and network and storage nodes, represent the read/write of values in the storage. Edges between network nodes either represent redirects or the presence of the same stored values.⁴

Graph Composition Example. To illustrate WEBGRAPH’s graph representation, let us consider the example web page given by Code 1. The web page embeds a script from Tracker 1 and an iframe from Tracker 2. The tracking iframe from Tracker 2 reads its tracking cookies and sends them to Tracker 3 via an XHR. Both trackers trigger requests to share

³<https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/API/webRequest>

⁴We match stored values with their encoded and hashed counterparts. Specifically, we look for presence of base64 encoded and MD5 and SHA-1 hashed values [23, 53].

²See Section 6 for a discussion of completeness of WEBGRAPH’s implementation.

```

1 <html>
2   <script src='tracker1.com/track.js'>
3     ...
4     var image = document.createElement('img');
5     image.src = 'tracker2.com/sync';
6     document.body.appendChild(image);
7     ...
8   </script>
9   ...
10  <iframe src='tracker2.com/track.html'>
11    <script>
12      ...
13      idCookie = document.cookie;
14      var newReq = new XMLHttpRequest();
15      newReq.open("GET", "tracker3.com?user_id=" +
16        " + idCookie);
17      ...
18    </script>
19  </iframe>
20 </html>

```

Code 1: An example web page sending requests to several trackers.

tracking identifiers. The HTTP requests and responses that result from loads in Code 1 are listed in Listing 1.

Tracker 1's script embeds an image element from Tracker 2, which causes the browser to send an HTTP request (Request 1 in Listing 1) that includes Tracker 2's cookie. Tracker 2 responds to this request with a redirect to Tracker 1 that embeds the user identifier Tracker 2 received via the initial request's Cookie header (i.e., user1). The browser makes a subsequent request (Request 2 in Listing 1) to Tracker 1. Tracker 1 responds with a tracking pixel image and a Set-Cookie header to set its own tracking cookie with the value userA. On the backend, Tracker 1 knows that userA is known as user1 by Tracker 2. Tracker 2's embedded iframe further shares its identifier cookie with Tracker 3. It does so by accessing its cookies locally via `document.cookie` and embedding them in an XHR to Tracker 3 (Request 3 in Listing 1).

Differences as compared to ADGRAPH. WEBGRAPH keeps ADGRAPH's HTML and JavaScript layers as they are, but extends the network layer and includes a new storage layer in the graph representation. WEBGRAPH also introduces information flow edges, which are absent in ADGRAPH, to entwine the extended network layer and the storage layer. The extension of network and the addition of storage layer allow WEBGRAPH to explicitly capture information sharing patterns used in advertising and tracking.

We illustrate the differences in Figure 4 which shows the graph representation of the web page in Code 1 and request and response sequences in Listing 1 for both ADGRAPH (Figure 4(a)) and WEBGRAPH (Figure 4(b)). ADGRAPH's representation of the example web page consists in two disjoint graphs which capture the individual actions of the two trackers: The first row of nodes (from 10 to 15) captures Tracker 2's tracking behavior: from the iframe loading to the initiation

```

-----
Request 1
URL: tracker2.com/sync
Cookie: user1
Response 1
Status: 302
Location: tracker1.com?tracker2_id=user1
-----
Request 2
URL: tracker1.com?tracker2_id=user1
Response 2
Status: 200
Set-Cookie: userA
Content: pixel.png
-----
Request 3
URL: tracker3.com?user_id=user1
Response 3
Status: 200

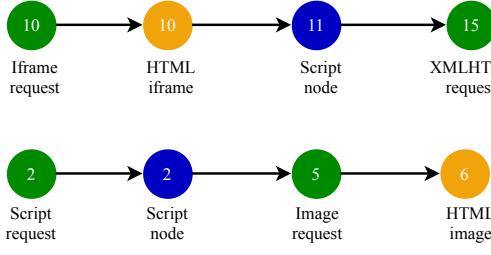
```

Listing 1: HTTP requests and responses initiated from Code 1.

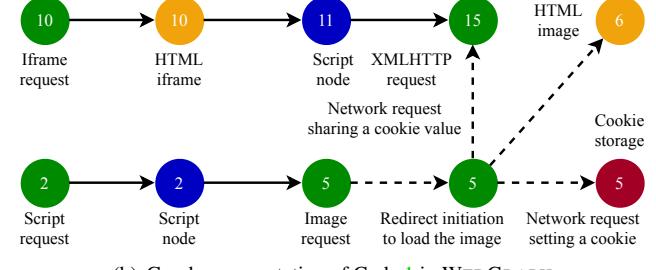
of an XHR request. The second row of nodes (from 2 to 6) captures Tracker 1's tracking behavior: from the script loading to the initiation of a network request for loading an image. In this figure, it becomes clear that ADGRAPH *does not* capture the information sharing pattern between the nodes, because of its inability to capture the redirect (Request 2) made by the image request (network node 5) and the cookie set (storage node 5; visible only in WEBGRAPH's graph) by the redirect request. WEBGRAPH, on the contrary, not only captures the flows appearing in ADGRAPH, but also captures the redirects (dotted edge between the two network nodes labeled 5) and cookies set by requests (the second network node 5 to storage node 5). This representation further enables WEBGRAPH to link requests that share common identifiers (node 5 to 15).

4.1.2 Features

We take the ADGRAPH feature set and augment them with three categories of features. These additional features come from WEBGRAPH's improved graph representation, i.e., extension of the network layer and a new storage layer. The features target storage, network, and information sharing behaviors that were absent in ADGRAPH. First, we extract features that measure the number of read/write cookie and localStorage accesses by a node. We obtain these features from the new storage layer. Second, we extract features that measure the number of requests and redirects to/from a node as well as the depth of a node in a redirect chain. These features come from our extension to the network layer. Third, we extract features that measure the number of different types of information sharing edges (e.g., nodes access the same storage node or share data of a storage node) to/from a node. We obtain these features using both the network and storage layers in WEBGRAPH's graph representation. We also extract some standard graph features (e.g., in-degree, out-degree, eccentricity) for the information



(a) Graph representation of Code 1 in ADGRAPH



(b) Graph representation of Code 1 in WEBGRAPH

Figure 4: Graph representation of Code 1 in ADGRAPH and WEBGRAPH. ● represents network nodes, ● represents script nodes, ○ represents HTML nodes, and ● represents storage nodes. Node numbers correspond to the lines in Code 1. In Figure 4(b), dotted (---) lines represent the additional edges that are captured by WEBGRAPH and missed by ADGRAPH.

sharing edges. We jointly refer to these three newly added categories of features as *flow* features. Table 4 in Appendix A lists the full set of features in WEBGRAPH, including these newly added flow features.

To illustrate the potential of these features in distinguishing ATS and Non-ATS resources, let us consider three flow features belonging to each of the categories described above: the number of storage elements set by a resource (Figure 5(a)), the number of requests that were redirected to a resource (Figure 6(b)), and the number of information sharing edge ancestors (Figure 6(c)). As explained in Section 4, ATS resources store user identifiers in storage elements and use redirects and sharing of identifiers in URLs to perform actions such as cookie syncing. Therefore, we expect ATS resources to set a larger number of storage elements, be at the receiving end of redirects, and be involved in a larger number of shared information edges than Non-ATS resources. We plot in Figure 5 the distributions of these features in our dataset. We see that, indeed, the distributions are different for benign and ATS resources, with ATS presenting higher values on average for the three features under study. The differences in distributions is especially apparent for Figure 6(c), which shows the number of shared information edge ancestors. In our dataset, we observe 589,218 cases of ATS receiving a cookie value in a request URL, as compared to 89,564 cases for non-ATS. This sharing is detected as an information sharing edge, which in turn leads to ATS having larger values in shared information edge properties than Non-ATS. In the case of redirects (Figure 6(b)), the probability that a Non-ATS resource has more than 7 redirects tends to 0, which is not the case with ATS resources. The number of ATS resources with more than 7 redirects is very small in our dataset ($\approx 0.04\%$). Yet, it is a top-20 feature in our classifier, as observing more than 7 redirects directly identifies the resource as ATS. Storage element setting (Figure 5(a)) shows a similar behavior, with ATS resources sometimes having more than 54 elements set, while Non-ATS resources never have so many.

While individual contributions of some of these flow features might be small, they provide a strong signal in distinguishing ATS when combined, as we show in the next section.

4.2 Evaluation

To evaluate WEBGRAPH, we use the same dataset of 10K web pages and method as in Section 3.2. To understand the marginal benefit of WEBGRAPH over ADGRAPH, we systematically compare the performance of different feature sets and graph representations. Table 2 summarizes the results.

Graph	Feature Set	Accuracy	Precision	Recall
ADGRAPH	Structural + Content	92.33 ± 0.50	88.91 ± 1.14	92.14 ± 0.65
	Structural	80.22 ± 0.81	71.85 ± 1.53	82.44 ± 1.26
WEBGRAPH	Structural+ Flow + Content	94.32 ± 0.27	92.24 ± 0.67	94.14 ± 0.30
	Structural + Flow	86.93 ± 0.64	80.57 ± 1.12	90.01 ± 0.50
	Structural	82.62 ± 0.47	75.67 ± 0.75	85.09 ± 1.41

Table 2: Evaluation of WEBGRAPH and ADGRAPH with different feature set variations.

We observe that ADGRAPH’s performance drops by at least 10% when content features are removed. Recall from Section 3.2 that if content features are present alongside structural features, ADGRAPH is particularly susceptible to evasion: trackers have an 8.72% evasion success rate on their own, and a 96.62% success rate if they collude with the first party. Thus, there is a trade-off in ADGRAPH between effectiveness (with content) and robustness (without content).

Second, Table 2 shows that WEBGRAPH’s performance is better than ADGRAPH due to its improved graph representation and new flow features. When using all feature sets, WEBGRAPH outperforms ADGRAPH by about 2-4%. If we remove content features for robustness, we observe a drop in accuracy limited to just 4-9% across all measures. We conclude that WEBGRAPH’s improved graph representation and new flow features can compensate for the loss of content features to a large extent.

Finally, Table 2 shows that WEBGRAPH’s improved graph representation by itself (i.e., even without the new flow features) contributes to about half of the improvement over ADGRAPH. WEBGRAPH with only structural features achieves 2-4% improvement across all measures as compared to ADGRAPH also with only structural features. We conclude that,

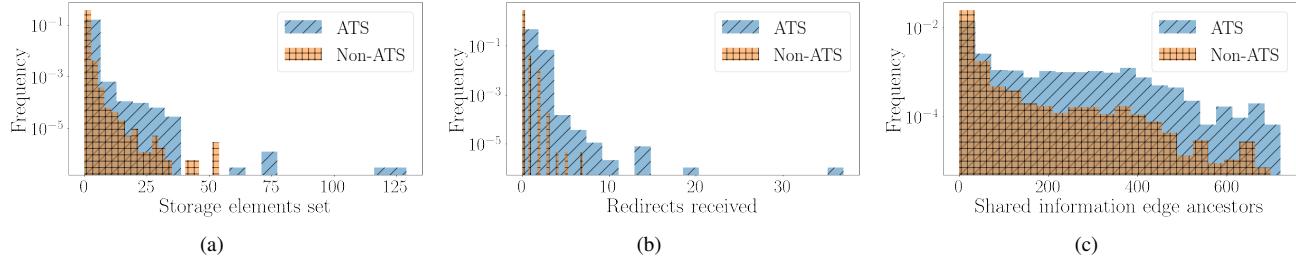


Figure 5: Histograms of three example flow features for ATS and Non-ATS resources (normalized, y-axis in log scale). (a) Number of storage elements set by a resource; (b) Number of network redirects received by a resource, and (c) Number of shared information ancestors of a resource. These features demonstrate different distributions for ATS and Non-ATS resources, and thus can help the classifier to distinguish between them.

while WEBGRAPH’s new flow features help improve its accuracy, the improved graph representation is an important contributor to performance.

Feature	Category	Information gain (%)
Shared information ancestors	Flow	6.48 \pm 0.69
Number of requests sent by node	Flow	5.9 \pm 0.69
Number of nodes in graph	Structure	5.46 \pm 0.35
Average degree connectivity of node	Structure	5.18 \pm 0.16
Number of edges in graph	Structure	4.19 \pm 0.34

Table 3: Top-5 most important features for WEBGRAPH’s classification, their category, and information gain (averaged over 10 folds).

To provide insights into the relative importance of flow and structural features, we list top five most important features in terms of information gain in Table 3. The two most important features are flow features. As discussed in Section 4.1.2, the top feature distribution (Figure 6(c)) is very different for ATS and non-ATS, so it’s not surprising that this feature contributes to the classification. Storage setting (Figure 5(a)) and received redirects (Figure 6(b)) contribute a smaller, but still useful, portion towards identification; they have information gains of 1.9% (± 0.37) and 2.5% (± 0.47) respectively (21st and 17th most important features). We also observe that structure features, enhanced by WEBGRAPH’s improved graph representation, also contribute towards the performance. We further analyze which features contribute most to each prediction of WEBGRAPH using treeinterpreter [54]. For $\approx 32\%$ of predicted ATS in the dataset, the flow features were the top contributors, indicating that they provide an important signal for the presence of trackers. In contrast, for $\approx 47\%$ of predicted Non-ATS in the dataset, structure features were the top contributors. These results confirm our earlier intuition that capturing information sharing behaviors that are unique to advertising and tracking carries significant predictive power.

4.3 Efficiency

We envision WEBGRAPH to be used for filter list curation and maintenance in an offline setting. WEBGRAPH relies on large

scale web crawls and notoriously expensive graph traversals for feature extraction. We now measure WEBGRAPH’s offline overhead to demonstrate its adequacy as a tool to periodically update filter lists.

Crawl time. Our implementation of WEBGRAPH has an upper bound of 60 seconds, enforced with a timeout, to crawl a website. In the average case, crawls take only ~ 26.46 seconds. Crawls can be parallelized over several instances to reduce the crawl time. For example, it took us around 10.5 hours to crawl 10K websites, parallelized over 7 instances. Without parallelization and if all websites would reach the timeout, the crawls would take ~ 166 hours.

Processing websites. On average, WEBGRAPH takes 0.72 seconds to build the graph, 15 seconds to extract features, and 0.25 seconds to train and test each website. For our crawl of 10K websites, it took us a total of ~ 44.36 hours to create their graphs and extract features on a single instance. This time can be significantly reduced using parallelization.

Update frequency. These estimates suggest that for 10K websites containing ~ 1.1 million requests WEBGRAPH will require, at most, ~ 166 (data crawling) and ~ 44.36 (data processing) hours with a single instance. However, when averaged over 7 instances, the computation time significantly reduces to only 16.83 hours (10.5 for crawling and 6.33 for processing). We anticipate the computation time for periodic updates to reduce significantly because many websites have low update frequency. Specifically, monitoring the update frequency of websites will allow us to only crawl when changes are expected in websites. In cases, where we determine that the website did not change since the last crawl, we will not recompute their classifications. With this performance, WEBGRAPH could be able to update filter lists on a daily basis, and certainly operate within the current the expiry period (mandated update frequency) of popular filter lists, e.g., 4 days for Easylist [7]. Frequent updates with WEBGRAPH can help remove outdated rules and as well as add new rules to block newly discovered ads and trackers.

5 WEBGRAPH Robustness

In this section, we evaluate WEBGRAPH’s robustness against content mutation attacks (described in Section 3) and structure mutation attacks.

5.1 Content mutation attacks

To evaluate WEBGRAPH against content mutations, we strengthen the threat model described in Section 3 to enable the adversary to also masquerade their resources as first party, i.e., through first-party subdomains. Overall, our attacks involve random mutations to domain names, subdomains, and the query string in URLs (Section 3.2).

By relying on content mutations, the adversary is able to switch 96.62% of their ATS resources to Non-ATS against ADGRAPH. Against WEBGRAPH, the adversary’s success rate plummets to just $8.34 \pm 0.66\%$ (over 10 folds). For example, `mylivesignature.com`, a tracking domain, was able to switch all of its 560 ATS resources to Non-ATS against ADGRAPH, but none against WEBGRAPH.

Note that, even though WEBGRAPH does not use content features, the evasion success rate against WEBGRAPH does not drop to zero. This is because some of the WEBGRAPH’s features implicitly rely on URL properties. For example, shared information edges, that consider sharing of cookie values via query strings in the URL, are affected by URLs manipulations.

5.2 Structure mutation attacks

Next, we evaluate WEBGRAPH’s robustness against structure mutations. We assume that the adversarial third-party has unrestricted black box access to the WEBGRAPH’s classifier, i.e., the adversary can make unlimited queries and observe WEBGRAPH’s classification output. This access enables the adversary to validate the effect of their structure mutations.

Attack details. We assume that the adversary can mutate the structure of a web page through resource addition, re-routing, and obfuscation. Moreover, we assume that the adversary also performs content mutations, to maximize its chance of success. Resource addition entails addition of new resources, such as images and scripts. Resource re-routing entails re-organization of existing redirect chains, i.e., dispersing a redirect chain in a sequence of XMLHttpRequest’s through one or multiple scripts. Resource obfuscation entails obfuscation of cookie or query string parameter values of existing resources, i.e., encoding or encrypting cookie or query string parameter values in a format that is not detected by WEBGRAPH’s implementation, before sharing them in network requests. To remain stealthy, we assume that the adversary does not delete functional content from the web page that could damage usability.

It is important to note that even simple mutations, such as adding a single element to the web page, can significantly

change graph properties and impact several features. For example, the addition of a child node causes a cascading effect. It increases the number of descendants of all the parent nodes in the branch, all the way up to the root node, and also impacts their centrality. Thus, the result of such simple mutations can become unpredictable and hard to control by the adversary: It can cause unintended classification changes for nodes under and outside the control of the adversary. Complex mutations, such as adding a combination of nodes at once, further complicate having control on the number of unintended classification changes. In our evaluation, we only consider atomic mutations, i.e., addition, re-routing, or obfuscation of individual resources.

Mutation algorithm. We capture the adversary’s unrestricted black box access to classifier by implementing a greedy random algorithm to find suitable mutations. This kind of algorithm is extensively used in the literature due to its simplicity and practicality [55, 56, 57]. The algorithm (formally described in Appendix D) iteratively mutates WEBGRAPH’s graph representation. At each step, it adds, re-routes, or obfuscates the resource that provides the best trade-off between *desired* (ATS to Non-ATS) and *undesired* (NON-ATS to ATS) classification switches. Resource addition is simulated by adding nodes to a randomly selected leaf nodes in the graph. Resource re-routing is simulated by adding each request, in a redirect chain, as an individual node to one or more randomly selected scripts. Resource obfuscation is simulated by replacing stored values in URLs with an encoding that is not detected by WEBGRAPH.

5.3 Empirical evaluation

Experimental Setup. To evaluate WEBGRAPH’s robustness, we must rebuild the graph and recompute the features after each mutation. To keep the evaluation time reasonable, we sample 100 web pages from our dataset, and we limit the graph growth to 20%. To ensure that this sample is representative of our dataset, we divide graphs into 5 bins according to their size and sample 20 web pages from each bin. We only consider web pages that have 250 or fewer nodes (i.e., 80% of the dataset; see Appendix B for the full distribution). We exclude large web pages to avoid exceptionally long evaluation times. For each web page, we designate the adversary as the third party with the highest number of resources classified as ATS. It is noteworthy that the adversary with the highest number of ATS resources has an opportunity to do maximum damage.

In this dataset, the median evaluation time per web page was 29.08 minutes, with 39% of the pages taking more than an hour to run. Even though this is a simulation, the computational cost is directly proportional to the operational cost for the adversary. The adversary must consume additional CPU cycles and memory and in the case of node addition, send additional network requests, thereby increasing the cost of their attack.

Success metrics. To measure adversary’s success, we define the following terms:

ATS_{Web} : Number of nodes classified as ATS.

ATS_{Adv} : Number of adversary nodes classified as ATS.

$Non-ATS_{Web}$: Number of nodes classified as Non-ATS.

$Non-ATS_{Adv}$: Number of adversary nodes classified as Non-ATS.

$desired$: Number of nodes switching from ATS_{Adv} to $Non-ATS_{Adv}$.

$undesired$: Number of nodes switching from $Non-ATS_{Web}$ to ATS_{Adv} .

$neutral$: Number of nodes switching from ATS to Non-ATS for non-adversary nodes.

Success rate: Desired changes from the adversary's point of view. It is calculated as $desired/ATS_{Adv}$.

Collateral damage: Undesired changes from the adversary's point of view. It is calculated as $undesired/(Non-ATS_{Adv} + Non-ATS_{Web})$.

Other changes: Non-consequential changes from the adversary's point of view. It is calculated as $neutral/ATS_{Web}$.

We illustrate the node switches, with the mutation algorithm, for an example graph in Appendix E.

5.3.1 Adversary's success

We assume that the adversary neither colludes with other third parties nor with the first party and can only perform mutations on the nodes and edges it controls. We conduct the attack on 100 web pages. We note that increasing the number of graph mutations increases the adversary's mean success rate from 38.6 ± 33.01 (median: 33.33) at 5% graph growth to 52.48 ± 33.4 (median: 50.00) at 20% graph growth. The classification switches lead to a decrease in the overall classification accuracy by 1.5%, recall by 8.85%, and precision by 2.29%.

However, the adversary's success comes at a cost of collateral damage. The average collateral damage rises from 2.17 ± 11.19 to 3.88 ± 13.55 (median: 0). In Figure 6, we illustrate the trade-off between success rate and collateral damage at 20% graph growth. The x-axis represents success rate, the y-axis represents collateral damage, and circles represent a trade-off between the two. The circles' color represents ATS_{Adv} or the number of classifications the adversary has to switch for the particular web page. The lighter the color, the more switches are required, i.e., the cost of success increases. For the web pages in this dataset, the adversary has, on average, $ATS_{Adv} = 5.98 \pm 5.39$ nodes classified as ATS. For certain pages, the ATS_{Adv} can be as high as 26.

Ideally, the adversary wants to be at the bottom right of the graph, where it achieves 100% success rate with zero collateral damage. The adversary is able to reach its ideal target on

only 13 web pages, which only required four switches. The adversary is able to achieve 50% or more success on 61 of the tested web pages. Together, they amount to 240 nodes switched, with 45 of these pages having non-zero collateral damage. On the other hand, we have 9 web pages that had a higher collateral damage than success rate: a net negative effect of the mutation. Out of these, 6 web pages had 0% success rate with non-zero collateral damage, and 3 web pages had a large collateral damage $> 75\%$ (with one web page hitting 83%).

Overall, our evaluation shows that even in the case of an unrealistic adversary that has the capabilities to manipulate structure features at will, and also the operational power to do so for a large number of iterations, there is no guarantee of perfect success.

Breakage. If undesired changes affect benign resources that are essential to the correct functioning of the web page, even a small collateral damage can break the page. This may have large impact on trackers. If users leave the broken web pages, the adversary cannot track them or show them ads.

We define website breakage as degradation in usability of the website. We say there is `major` breakage if the user is unable to complete the primary functionality of the web pages (e.g. login, search or page navigation). If the user is unable to complete a secondary functionality of the web pages (e.g. comment or review), we consider that there is `minor` breakage. Otherwise, we consider that the web page does not have any breakage.

We quantify breakage on all of the 21 web pages where the adversary experiences undesired classification switches.⁵ We open these web pages side by side on stock Firefox and a Firefox configured with an extension that blocks the URLs that switched classification, and we compare them side by side to identify any visual signs of breakage.

We ask two reviewers to perform the analysis. Our reviewers attain an agreement of 90.46% in their evaluation. They find that the undesired classification switches cause `major` breakage on 3 and `minor` breakage on 2 web pages. This breakage mostly happens when the first-party resources are switched from `Non-ATS` \rightarrow `ATS`.

Careless adversary. If the adversary is not concerned with changes to any non-adversarial nodes, their collateral damage decreases. The adversary still does not want their own content to be blocked, so it will optimize against their own nodes switching to `ATS`. This change in strategy updates the collateral damage calculation to: $undesired/Non-ATS_{Adv}$.

As per our modified definition, the web pages on which all of the adversary nodes are classified as `ATS`, there can be no collateral damage; we note 55 such web pages. For the remaining 45 web pages, where the adversary can experience collateral damage, the mean growth in success rate does not change much

⁵In total, the adversary experiences undesired classification switches on 45 web pages. However, 24 web pages no longer serve the switched `ATS` resources.

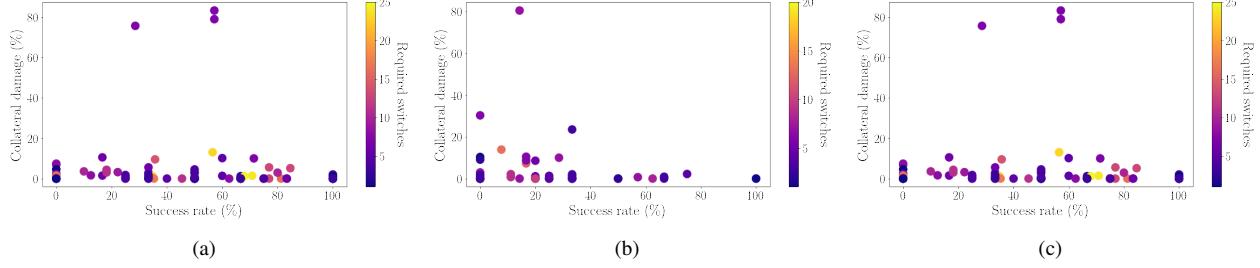


Figure 6: Adversary’s success rate vs. collateral damage for each web page in the test data at 20% graph growth. Figure 6(a) represents all mutations, 6(b) represents only structure mutations, and 6(c) represents only resource re-routing and obfuscation mutations. Colored circles represent the number of required switches.

from the previous scenario, but naturally the trade-off is better. Further, out of 45 web pages, only 8 web pages have collateral damage, as compared to 27 web pages that had collateral damage as per our original definition. Out of these 8 web pages, 4 had a higher collateral damage than success rate (net negative effect), and 6 web pages have a large collateral damage $> 20\%$ (with 2 web pages hitting 100%). Thus, even when an adversary is not concerned about collateral damage to other parties they are not significantly more successful in subverting WEBGRAPH.

Collusion with the first party. So far, we have assumed that the adversary is a single third party that does not collude with other third parties or the first party. If we assume the adversary colludes with both, the adversary can add child nodes to *any* node in the graph. This is a much stronger adversary than in Section 5.3.1, where in each iteration the adversary can only test a random subset of the options. Realistically, such a powerful collusion would be difficult to implement, as it would require coordination and cooperation among multiple parties to ensure that the mutation is feasible.

We repeat our experiment, but we now allow the adversary to consider all possible mutation options on any node, and pick the best one in each iteration. These experiments take longer to run (see Appendix C), so we only analyze 100 web pages whose graphs have at most 50 nodes. We see that collusion enables the adversary to have a slightly higher success rate (63 pages with success rate $> 50\%$ as compared to 60 for the non-colluding adversary) and lower collateral damage (9 pages with damage $> 0\%$ compared to 18 pages for a non-colluding adversary). These results are described in detail in Appendix F.

5.3.2 Impact of mutation choice

Next, we evaluate the adversary’s preference in selecting the most useful mutations. We notice that the adversary picks resource addition 81.70%, resource re-routing 17.26%, and resource obfuscation 0.04% of the time. Resource obfuscation is rarely chosen by the adversary because the graph already has content manipulations applied, and these manipulations have already severed many of the edges that would be severed

by resource obfuscation. To separate out the impact of different mutations, we conduct two additional experiments: (1) where the adversary can only perform resource addition, and (2) where the adversary can only perform resource re-routing and obfuscation.

We exclude 33 of the web pages for experiment 2 because these web pages do not have re-route-able or obfuscate-able resources. For the remaining 67 pages, we see that the re-routing/obfuscation mutations (Figure 6(c)) are more effective than addition mutations (Figure 6(b)). Re-routing/obfuscation not only yields higher success rates for the adversary, but also results in lower collateral damage. This is unsurprising because these mutations target information sharing patterns which are distinctive of trackers; changing these patterns removes an important signal for the classifier (see Table 3).

However, in practice, resource re-routing and obfuscation would entail high costs for the adversary since they involve the manipulation of identifier sharing patterns. Specifically, the adversary would have to coordinate with other parties on changes to these patterns, and redesign how they perform tracking in order to perform these mutations. The success of these mutations also depends on the degree to which flows are captured by the instrumentation used to create the graph. WEBGRAPH’s instrumentation approximates information flows and will not capture all attempts by an adversary to use re-routing and obfuscation. We argue that this is not a fundamental flaw in WEBGRAPH’s architecture but a limitation in our implementation that approximates information flow (Section 4). A fully fledged instrumentation would make these manipulations much more difficult to deploy. See Section 6 for an extended discussion. Resource addition has fewer costs for the adversary since it does not involve coordination with additional parties. This manipulation is not affected by the type of implementation because it is not related to the flow of identifiers.

5.3.3 Comparison with ADGRAPH.

We also evaluate whether WEBGRAPH, in addition to having superior classification performance, offers robustness benefits over ADGRAPH. For this comparison, we only use ADGRAPH’s structural features, as we already demonstrated that

content features are not robust. Because ADGRAPH does not have features based on flow information, we only perform resource addition. We find that the adversary has greater success against ADGRAPH than WEBGRAPH, but also suffers from more collateral damage (Figures 13 and 14 in Appendix F). This is because the structural effects of node additions are hard to control, as explained in Section 5.2. Since the former is beneficial to the adversary but the latter is not, it is not clear-cut as to whether one system provides more robustness than the other.

In summary, our results indicate that mutations to the structure of the graph are harder for an adversary to control than content mutations. It is not trivial for an adversary to produce the desired classification switches for their resources without producing any undesired changes. This makes WEBGRAPH, without content features, more robust to adversarial evasion attacks than prior approaches. Within the structural mutations, re-routing/obfuscating resources target information flow and are a more effective strategy than adding resources. At the same time, performing these mutations is not trivial for the adversary since they involve coordination with multiple parties.

6 Limitations

In this section, we discuss limitations of WEBGRAPH’s design, implementation, and evaluation.

Completeness. For efficiency reasons, WEBGRAPH focuses on a limited subset of the browser’s API surface, such as HTTP cookie headers, `document.cookie`, and `window.localStorage`. WEBGRAPH’s implementation is also geared towards capturing client-side information that is pertinent to stateful tracking. However, techniques used by ATS need not to be limited to these APIs or to stateful tracking. Some ATS have started to use stateless tracking techniques, such as browser fingerprinting, which use APIs that are not currently covered by our instrumentation [58, 59, 60]. To account for these techniques, WEBGRAPH’s instrumentation must be extended to include the corresponding APIs.

WEBGRAPH’s manually designed graph representation and feature set capture the most well-known information sharing patterns. The limits of these approach are shown in Section 5.3.2, where we show that an adversary capable of hiding or obfuscating traditional sharing flows has a better chance to bypass WEBGRAPH than doing structure modifications. This limitation is, however, linked to our implementation choices. To increase WEBGRAPH’s coverage of sharing behaviors, it suffices with increase the instrumentation to cover more information flows. Ideally, we would instrument full-blown information flow tracking. Such expansion would incur prohibitive runtime overheads (up to 100X-1000X [50]) and its complexity makes it hard to integrate in the browser [51, 52, 61, 62]. Nevertheless, the design of WEBGRAPH permits that the instrumentation to be upgraded gradually, as ATS evolve in response

to our evasion protection techniques, increasing the cost of evasion without fundamentally changing the detection approach.

Robustness analysis. Inspired by previous work on graph-based detection evasion [55, 56, 57], we use a greedy algorithm to attack WEBGRAPH. This algorithm only considers the best mutation in each iteration, and not the best overall mutation. Thus, it is not guaranteed to find the optimal mutation sequence that would lead to the best adversary performance. We note however that, as our experiments on small websites show, even exhaustive search does not lead to perfect success. We expect adversaries to try alternative algorithms to improve their success rates. However, any alternative that is close to exhaustive search will become prohibitively expensive for the adversary when the web page graph is large.

Another option for the adversary would be to perform more sophisticated graph mutations instead of the simple node additions that we perform. An adversary could tailor their mutations to the page’s graph structure by studying how their node changes affect the graph properties of the web page. However, this requires that the rest of the graph (i.e., the portions outside of the adversary’s control) remaining unchanged. Realistically, it would be difficult for an adversary to coordinate with other parties to generate these changes.

Finally, we note that the dynamism of modern websites [63] complicates the process for the adversary. Web pages change often, sometimes on every load. Even if the adversary manages to find an appropriate set of mutations, those mutations may be invalid the next time the page is reloaded.

7 Conclusion

In this paper, we showed that state-of-the-art ad and tracker blocking approaches are susceptible to evasion due to their reliance on easy-to-manipulate content features. We then showed that information sharing patterns in online advertising and tracking can instead be leveraged for robust blocking. Specifically, our proposed WEBGRAPH builds a cross-layer graph representation to capture such information flows and train a machine learning classifier for accurate and robust ad and tracker blocking. Our results showed that it is non-trivial to evade WEBGRAPH’s classifier without causing unavoidable collateral damage.

While it is not foolproof, we believe that WEBGRAPH raises the bar for advertisers and trackers attempting to evade detection. We foresee that advertising and tracking services would need to significantly re-architecture their information sharing patterns to achieve long-lasting evasion against WEBGRAPH. We note, however, that introducing new information flows may be quite complicated, as they may require collaboration among the first-party and numerous third-parties on a typical web page.

References

- [1] uBlock Origin. <https://github.com/gorhill/uBlock>.
- [2] Ghostery. <https://www.ghostery.com>.
- [3] MDN. Storage access policy: Block cookies from trackers. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Storage_acces s_policy.
- [4] MDN. Redirect tracking protection. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Redirect_Tracking_Protection.
- [5] Microsoft Edge Team. Introducing tracking prevention, now available in Microsoft Edge preview builds. <https://blogs.windows.com/msedgedev/2019/06/27/tracking-prevention-microsoft-edge-preview/>.
- [6] Brave. A Long List of Ways Brave Goes Beyond Other Browsers to Protect Your Privacy. <https://brave.com/privacy-features/>.
- [7] EasyList. <https://easylist.to/easylist/easylis t.txt>.
- [8] EasyPrivacy. <https://easylist.to/easylist/eas yprivacy.txt>.
- [9] Disconnect. <https://disconnect.me/>.
- [10] Sruti Bhagavatula, Christopher Dunn, Chris Kanich, Minaxi Gupta, and Brian Ziebart. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *ACM Workshop on Artificial Intelligence and Security*, 2014.
- [11] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. An Automated Approach for Complementing Ad Blockers' Blacklists. In *Privacy Enhancing Technologies Symposium (PETS)*, 2015.
- [12] Anastasia Shuba, Athina Markopoulou, and Zubair Shafiq. NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking. In *Privacy Enhancing Technologies Symposium (PETS)*, 2018.
- [13] Qianru Wu, Qixu Liu, Yuqing Zhang, Peng Liu, and Guanxing Wen. A Machine Learning Approach for Detecting Third-Party Trackers on the Web. In *ESORICS*, 2016.
- [14] Andrew J. Kaizer and Minaxi Gupta. Towards Automatic identification of JavaScript-oriented Machine-Based Tracking. In *IWSPA*, 2016.
- [15] Muhammad Ikram, Hassan Jameel Asghar, Mohamed Ali Kaafar, Anirban Mahanti, and Balachandar Krishnamurthy. Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning. In *Privacy Enhancing Technologies Symposium (PETS)*, 2017.
- [16] Peter Snyder, Antoine Vastel, and Benjamin Livshits. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. In *ACM SIGMETRICS*, 2020.
- [17] Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. Anything to hide? studying minified and obfuscated code in the web. In *The World Wide Web Conference*, pages 1735–1746, 2019.
- [18] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2020.
- [19] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. Filter List Generation for Underserved Regions. In *WWW*, 2020.
- [20] Ha Dao, Johan Mazel, and Kensuke Fukuda. Characterizing CNAME Cloaking-Based Tracking on the Web. In *Network Traffic Measurement and Analysis Conference*, 2020.
- [21] Yana Dimova, Gunes Acar, Lukasz Olejnik, Wouter Joosen, and Tom Van Goethem. The CNAME of the Game: Large-scale Analysis of DNS-based Tracking Evasion. *Proceedings on Privacy Enhancing Technologies*, 2021(1):109–126, 2021.
- [22] Panagiotis Papadopoulos, Nicolas Kourtellis, and Evangelos P. Markatos. Cookie synchronization: Everything you always wanted to know but were afraid to ask. In *Proceedings of the World Wide Web (WWW) Conference*, 2019.
- [23] Imane Fouad, Natalia Bielova, Arnaud Legout, and Natasa Sarafijanovic-Djukic. Missed by Filter Lists: Detecting Unknown Third-Party Trackers with Invisible Pixels. *Proceedings on Privacy Enhancing Technologies*, 2020:499–518, 04 2020.
- [24] Adblock Plus. <https://adblockplus.org/>.
- [25] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *IMC*, 2017.
- [26] Catalin Cimpanu. Ad Network Uses DGA Algorithm to Bypass Ad Blockers and Deploy In-Browser Miners.

<https://www.bleepingcomputer.com/news/security/ad-network-uses-dga-algorithm-to-bypass-ad-blockers-and-deploy-in-browser-miners/>, 2018.

[27] Zhang Zaifeng. Who is Stealing My Power III: An Adnetwork Company Case Study, 2018. <http://blog.netlab.360.com/who-is-stealing-my-power-iii-an-adnetwork-company-case-study-en/>.

[28] Andrew Bosworth. A New Way to Control the Ads You See on Facebook, and an Update on Ad Blocking. <https://newsroom.fb.com/news/2016/08/a-new-way-to-control-the-ads-you-see-on-facebook-and-an-update-on-ad-blocking/>, 2016.

[29] Garrett Sloane. Ad Blocker’s Successful Assault on Facebook Enters Its Second Month. <http://adage.com/article/digital/blockrace-adblock/311103/>, 2017.

[30] Ben Williams. Ping pong with Facebook. <https://adblockplus.org/blog/ping-pong-with-facebook>, 2018.

[31] Weihang Wang, Yunhui Zheng, Xinyu Xing, Yonghui Kwon, Xiangyu Zhang, and Patrick Eugster. WebRanz: Web Page Randomization For Better Advertisement Delivery and Web-Bot Prevention. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[32] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. Errors, misunderstandings, and attacks: Analyzing the crowdsourcing process of ad-blocking systems. In *Proceedings of the 2019 Internet Measurement Conference (IMC)*, 2019.

[33] Hung Dang, Yue Huang, and Eechien Chang. Evading Classifiers by Morphing in the Dark. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[34] Aurore Fass, Michael Backes, and Ben Stock. HideNoSeek: Camouflaging Malicious JavaScript in Benign ASTs. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[35] Niels Hansen, Lorenzo De Carli, and Drew Davidson. Assessing Adaptive Attacks Against Trained JavaScript Classifiers. In *16th EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2020.

[36] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. Detecting filter list evasion with event-loop-turn granularity javascript signatures. In *Proceedings of the IEEE Symposium on Security and Privacy (May 2021)*, 2021.

[37] Amir Hossein Kargaran, Mohammad Sadegh Akhondzadeh, Mohammad Reza Heidarpour, Mohammad Hossein Manshaei, Kave Salamatian, and Masoud Nejad Sattary. On detecting hidden third-party web trackers with a wide dependency chain graph: A representation learning approach. *arXiv preprint arXiv:2004.14826*, 2020.

[38] Shitong Zhu, Zhongjie Wang, Xun Chen, Shasha Li, Umar Iqbal, Zhiyun Qian, Kevin S Chan, Srikanth V Krishnamurthy, and Zubair Shafiq. A4: Evading learning-based adblockers. *arXiv preprint arXiv:2001.10999*, 2020.

[39] Florian Tramèr, Pascal Dupré, Gili Rusak, Giancarlo Pellegino, and Dan Boneh. Adversarial: Perceptual ad blocking meets adversarial machine learning. In *CCS*, 2019.

[40] Bypassing ad blockers for Google Analytics. <https://analytics-bypassing-adblockers.netlify.app/>.

[41] Bjørn Johansen. Tracking visitors with adblockers. <https://www.bjornjohansen.com/tracking-visitors-with-adblockers>.

[42] Jason Bloomberg. Ad Blocking Battle Drives Disruptive Innovation. <https://www.forbes.com/sites/jasonbloomberg/2017/02/18/ad-blocking-battle-drives-disruptive-innovation>, 2017.

[43] Romain Cointepas. CNAME Cloaking, the dangerous disguise of third-party trackers. <https://medium.com/nextdns/cname-cloaking-the-dangerous-disguise-of-third-party-trackers-195205dc522a>, 2010.

[44] Catalin Cimpanu. Ad Network Uses DGA Algorithm to Bypass Ad Blockers and Deploy In-Browser Miners. <https://www.bleepingcomputer.com/news/security/ad-network-uses-dga-algorithm-to-bypass-ad-blockers-and-deploy-in-browser-miners/>, 2018.

[45] Daniel Plohmann, Khaled Yakdan, Michael Klatt, Johannes Bader, and Elmar Gerhards-Padilla. A Comprehensive Measurement Study of Domain Generating Malware. In *Proceedings of the 25th USENIX Security Symposium*, 2016.

[46] Steven Englehardt and Arvind Narayanan. Online tracking: A 1-million-site measurement and analysis. In *Proceedings of ACM CCS 2016*, 2016.

[47] Muhammad Ahmad Bashir and Christo Wilson. Diffusion of user tracking data in the online advertising ecosystem. *Proceedings on Privacy Enhancing Technologies*, 2018(4):85–103, 2018.

[48] Lukasz Olejnik, Minh-Dung Tran, and Claude Castelluccia. Selling off privacy at auction. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[49] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. The web never forgets: Persistent tracking mechanisms in the wild. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[50] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. Jsflow: Tracking information flow in javascript and its apis. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, pages 1663–1671, 2014.

[51] Andrey Chudnov and David A Naumann. Inlined information flow monitoring for javascript. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 629–643, 2015.

[52] Quan Chen and Alexandros Kapravelos. Mystique: Uncovering information leakage from browser extensions. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1687–1700, 2018.

[53] Steven Englehardt, Jeffrey Han, and Arvind Narayanan. I never signed up for this! privacy implications of email tracking. *Proceedings on Privacy Enhancing Technologies*, 2018(1):109–126, 2018.

[54] treeinterpreter. <https://pypi.org/project/treeinterpreter/>.

[55] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. Adversarial attacks on neural networks for graph data. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2847–2856, 2018.

[56] Shifu Hou, Yujie Fan, Yiming Zhang, Yanfang Ye, Jingwei Lei, Wenqiang Wan, Jiabin Wang, Qi Xiong, and Fudong Shao. *acyber*: Enhancing robustness of android malware detection system against adversarial attacks on heterogeneous graph based model. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, pages 609–618, 2019.

[57] Xiaoyun Wang, Joe Eaton, Cho-Jui Hsieh, and Felix Wu. Attack graph convolutional networks by adding fake nodes. *arXiv preprint arXiv:1810.10751*, 2018.

[58] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. Fingerprinting the Fingerprinters:Learning to Detect Browser Fingerprinting Behaviors. In *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2021.

[59] Anupam Das, Gunes Acar, Nikita Borisov, and Amogh Pradeep. The Web’s Sixth Sense: A study of scripts accessing smartphone sensors. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1515–1532, 2018.

[60] Vasilios Mavroudis, Shuang Hao, Yanick Fratantonio, Federico Maggi, Christopher Kruegel, and Giovanni Vigna. On the privacy and security of the ultrasound ecosystem. *Proceedings on Privacy Enhancing Technologies*, 2017(2):95–112, 2017.

[61] Ben Stock, Sebastian Lekies, Tobias Mueller, Patrick Spiegel, and Martin Johns. Precise Client-side Protection against DOM-based Cross-Site Scripting. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 655–670, San Diego, CA, 2014.

[62] Sebastian Lekies, Ben Stock, and Martin Johns. 25 million flows later: Large-scale detection of DOM-based XSS. In *Proceedings of the 2013 ACM SIGSAC conference on Computer and Communications Security*, pages 1193–1204, 2013.

[63] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, 2011.

A Comparison between ADGRAPH and WEBGRAPH features

Table 4 compares and contrasts the features used in WEBGRAPH and ADGRAPH. WEBGRAPH does not use content features. Graph size, Degree and Centrality features come under both structure and flow categories, since they include graph properties that are based on both normal (structure feature) and shared information edges (flow feature). WEBGRAPH uses both types of edges, whereas ADGRAPH uses only normal edges. Some structural features used in ADGRAPH are not used in WEBGRAPH due to WEBGRAPH being adapted for offline use, whereas the features are useful in an online context.

B Distribution of graph sizes

Figure 7 shows the distribution of number of nodes in the graph representations of the web pages in our dataset. Since 80% of web pages have 250 nodes or fewer, we sample from this subset in our structural mutation experiments in Section 5.

Table 4: WEBGRAPH features comparison with ADGRAPH. ● indicates that a feature is present. WEBGRAPH calculates Graph size, Degree and Centrality features using both normal and shared information edges. The former comes under structural features while the latter comes under flow features.

Feature	Type	WEBGRAPH	ADGRAPH
Request type (e.g. iframe, image)	Content		●
Ad keywords in request (e.g. banner, sponsor)	Content		●
Ad or screen dimensions in URL	Content		●
Valid query string parameters	Content		●
Length of URL	Content		●
Domain party	Content		●
Sub-domain check	Content		●
Base domain in query string	Content		●
Semi-colon in query string	Content		●
Graph size (# of nodes, # of edges, and nodes/edge ratio)	Structure	●	●
Degree (in, out, in+out, and average degree connectivity)	Structure	●	●
Centrality (closeness centrality, eccentricity)	Structure	●	
Number of siblings (node and parents)	Structure		●
Modifications by scripts (node and parents)	Structure		●
Parent's attributes	Structure		●
Parent degree (in, out, in+out, and average degree connectivity)	Structure		●
Sibling's attributes	Structure		●
Ascendant's attributes	Structure	●	●
Descendant of a script	Structure	●	●
Ascendant's script properties	Structure	●	●
Parent is an eval script	Structure	●	●
Local storage access (# of sets, # of gets)	Flow (storage)	●	
Cookie access (# of sets, # of gets)	Flow (storage)	●	
Requests (sent, received)	Flow (network)	●	
Redirects (sent, received, depth in chain)	Flow (network)	●	
Common access to the same storage node	Flow (shared information)	●	
Sharing of a storage node's value in a URL	Flow (shared information)	●	
Graph size (# of nodes, # of edges, and nodes/edge ratio)	Flow (shared information)	●	
Degree (in, out, in+out, and average degree connectivity)	Flow (shared information)	●	
Centrality (closeness centrality, eccentricity)	Flow (shared information)	●	

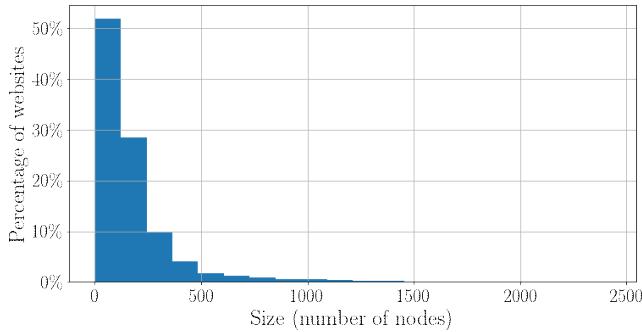


Figure 7: Distribution of number of nodes in the graph representations of the web pages in the dataset. 80% of the web pages have 250 nodes or fewer.

C Experimental run times

Figures 8 and 9 describe the run times for the experiment described in Section 5.3.1 (adversary without collusion). Figure 8 shows the impact of graph size on each iteration of the experiment. As expected, smaller graphs have lower run times since features have to be calculated over a smaller number of nodes. Note that graph size is not the only contributing factor to run times. Other factors such as the complexity of the structure and flow behaviors would also contribute towards time spent in each iteration, which is why we observe variations in iteration time among graphs of the same size. We see that the mean time per iteration can be as high as ≈ 1200 seconds (median is ≈ 68 seconds). Figure 9 shows the total experiment time over all iterations for a graph. Since we increase the sizes of graphs by 20% of their original size, bigger graphs will have a

larger number of iterations. In our dataset, the maximum time taken for an experiment is 46654.19 seconds, the minimum is 15.67 seconds, and the median is 1745.11 seconds. 39% of the graphs in our dataset have a run time of more than an hour.

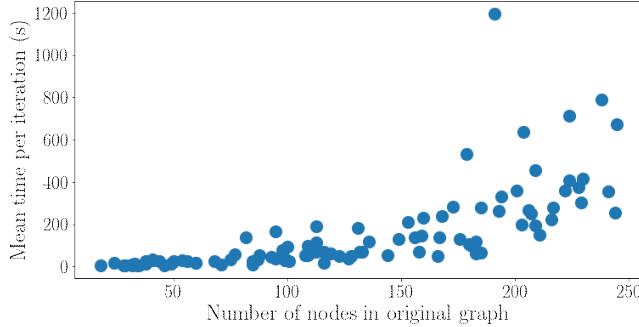


Figure 8: Mean time per iteration vs graph size for the experiment without collusion. Standard deviation over all the iterations for each graph was less than 2%. Larger graph sizes take longer time for each iteration.

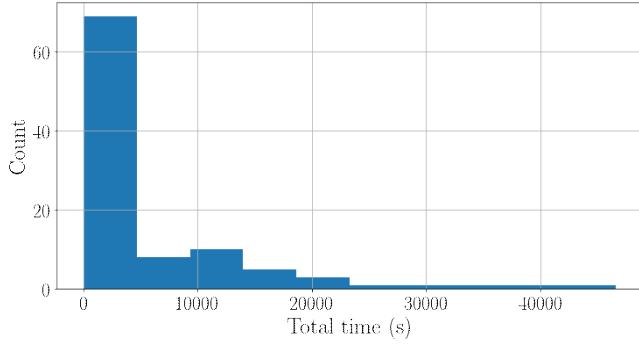


Figure 9: Distribution of total run time for the experiments in Section 5.3.1. 39% of the graphs in our dataset have a run time of more than an hour.

Figure 10 shows the total experiment time over all iterations for the experiment described in Section 5.3.1 (collusion with first party). The median time is 265.03 seconds, with the maximum time going up to 992.67 seconds, despite the maximum graph size being only 50 nodes. In comparison, for the adversary without collusion, for graph sizes up to 50 nodes, the median is 21.46 seconds and the maximum is 221.51 seconds. Since the adversary considers all nodes in the graph as potential parents, each iteration takes a longer amount of time.

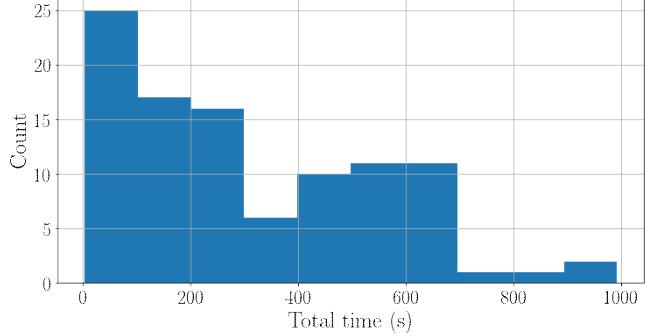


Figure 10: Distribution of total run time for the experiments in Section 5.3.1. The experiment time can be as high as 992.67 seconds for a graph size ≤ 50 nodes.

D Graph Mutation algorithm

In each iteration, the algorithm mutates WEBGRAPH’s graph representation and probes the model for classification decisions. The algorithm takes the following inputs: a graph representation of a web page, G_0 , consisting of all the nodes in the graph; a set of nodes and edges T of size l_T , representing the resources loaded by the adversary, hereafter referred to as the adversary resources; a trained classifier M that identifies ATS in WEBGRAPH; and a maximum number of iterations that the algorithm can run, max_iter .

The algorithm processes the input as follows: It first uses the classifier M to obtain classifications of all nodes in the original graph G_0 (lines 1–4 in Algorithm D). Second, it iterates over the steps from lines 9–20 max_iter times. In each iteration, every adversary node tries resource addition, resource re-routing, and obfuscation, and produces a new mutated graph, G_i (line 11). Third, it extracts features from the mutated graph G_i and uses them to classify all the nodes in this graph (lines 11–12). Fourth, it compares the predictions in the original and mutated graphs to obtain the number of desired and undesired switches (line 13). We assume an adversarial goal for which *desired* switches are all those in which an adversary node is switched from ATS to Non-ATS, whereas *undesired* switches are all those where any Non-ATS node is switched to ATS node. We call the total number of adversarial ATS nodes whose prediction the adversary wishes to change to Non-ATS the number of *required* switches. The switching of nodes not under the adversary’s control from ATS to Non-ATS do not affect the adversary. These switches are, therefore, neither desired nor undesired. Finally, the adversary chooses the mutation that provides the best result, i.e., the one with the best trade-off between desired and undesired switches (lines 14–15). The adversary updates its T based on the chosen mutation (line 18). To keep memory and run time manageable, at the end of every iteration the algorithm randomly samples l_T adversarial nodes and edges from T (line 19) to be considered in the next iteration.

Algorithm 1 Greedy random graph mutation. G_0 is a web page representation, T is the set of l_T nodes and edges controlled by the adversary, M is a trained model, and max_iter is the maximum number of operations.

Input: $G_0, T, C, M, \text{max_iter}$

```

1: for  $v \in G_0$  do
2:    $x_{G_0} \leftarrow \text{ExtractFeatures}(v) \forall v \in G_0$ 
3:    $y_{G_0} \leftarrow \text{Classify}(M, x) \forall x \text{ in } x_{G_0}$ 
4: end for
5:  $G \leftarrow G_0$ 
6:  $i \leftarrow 0$ 
7:  $\text{graph-info} = []$ 
8: while  $i < \text{max\_iter}$  do
9:   for  $t \in T$  do
10:     $G_t \leftarrow \text{MutateGraph}(G, t)$ 
11:     $x_t \leftarrow \text{ExtractFeatures}(v) \forall v \in G_t$ 
12:     $y_t \leftarrow \text{Classify}(M, x) \forall x \text{ in } x_t$ 
13:     $d, u \leftarrow \text{GetDesiredAndUndesired}(y_t, y_{G_0})$ 
14:     $\Delta_t = d - u$ 
15:     $\text{graph-info}[t] \leftarrow (\Delta_t, t, G_t)$ 
16:   end for
17:    $G \leftarrow G_t$  in  $\text{graph-info}[t]$  with largest  $\Delta_t$ 
18:    $T \leftarrow \text{UpdateAdv}(T, t \in \text{graph-info}[t])$ 
19:    $T \leftarrow \text{sample}(T, l_T)$ 
20:    $i \leftarrow i + 1$ 
21: end while

```

Mutation example. An example iteration of the greedy random algorithm, using resource addition as the mutation, is shown in Figure 11. The algorithm proposes two mutations (right) to the initial graph (left), and chooses the mutation that provides the best trade-off between the desired and undesired classification switches. Figure 11 also illustrates that even a simple node addition may lead to unintended changes in classification decisions: the adversary may change the classification decisions for some nodes from ATS to Non-ATS, but as a side effect nodes previously classified as Non-ATS are now classified as ATS.

E Mutations on a single web page.

To illustrate how mutations result in classification switches, we take as an example a web page in which the third party with the highest number of ATS resources is `assets.wogaa.sg`, which has 12 nodes in the graph. Figure 12 shows the breakdown of classification switches as the adversary mutates the graph using the greedy mutation algorithm. The ATS_{Adv} or the number of classifications the adversary wants to switch is 5 (pink line —). From the adversary's point of view, adversarial nodes switching from ATS \rightarrow Non-ATS are desired (blue line $\text{—}\bullet\text{—}$), whereas adversarial nodes switching from Non-ATS \rightarrow ATS are undesired (orange line $\text{—}\blacktriangle\text{—}$). We consider Non-ATS \rightarrow ATS changes on non-adversarial nodes to be undesired because they

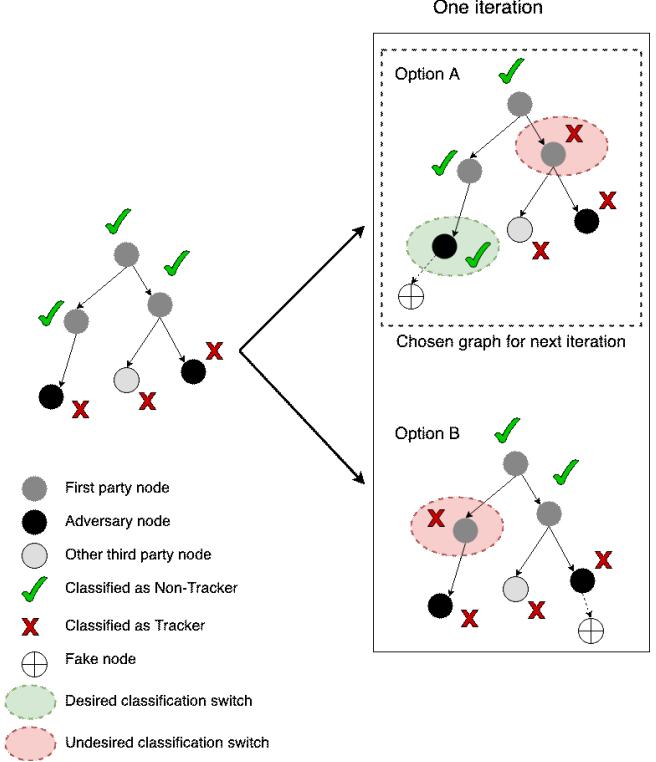


Figure 11: One iteration of the greedy random mutation algorithm. In this iteration the algorithm selected two adversarial nodes to add a child (Option A: bottom left node, top; Option B: bottom right node, top). In option A, the change leads to one desired (green circle) and one undesired (red circle) modification. In option B, the change only causes one undesired modification. The algorithm would pick the graph with best trade-off between desired and undesired switches for the next iteration: i.e., Option A.

may have unintended impact on the web page (red line $\text{—}\blacktriangle\text{—}$ and brown line $\text{—}\blacksquare\text{—}$). For instance, a first party Non-ATS \rightarrow ATS switch may break the web page. We note that, if the adversary's goal is to just create a denial of service and force the user to disable ad and tracker blocking, the adversary might be unconcerned about breakage. In our experiments, switches that do not affect the adversary, such as ATS \rightarrow Non-ATS for non-adversary nodes, are neither considered desired nor undesired (purple line $\text{—}\bullet\text{—}$ and green line $\text{—}\bullet\text{—}$).

There are two points worth highlighting from Figure 12: (1) Even if an adversary achieves the maximum number of desired switches, the mutations may produce undesirable changes, to both the adversary's nodes and others. For instance, at 20% of growth, 3 of the adversary's ATS nodes are classified as Non-ATS, but also 7 Non-ATS nodes (3 adversary and 4 non-adversary) switch to the undesired ATS classification. (2) The evolution of desired and undesired switches is not monotonic, i.e. the classification of nodes may change in both directions

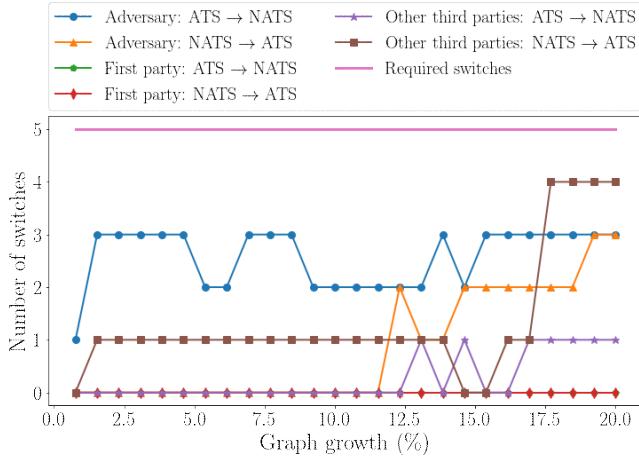


Figure 12: Example breakdown of classification switches for the adversary’s and other nodes on the graph. NATS is shorthand for Non-ATS. $ATS_{Adv} = 5$ (pink line), $Non-ATS_{Adv} = 7$, $ATS_{Web} = 62$, $ATS_{Web} = 13$ (not shown in plot). At 20% growth, the adversary achieves 3 desired switches, 7 undesired switches and 1 neutral switch. This leads to a success rate of 60%, a collateral damage of 10.14% and other changes of 7.7%.

as the adversary mutates the graph, resulting in increasing or decreasing counts. This finding reinforces our argument that it can be cumbersome for an adversary to create targeted structural mutations without any unintended consequences. Not only it is hard to predict how mutations will affect adversary’s own desired classification, but also how those mutations may result in undesirable changes to others.

F WEBGRAPH robustness experiments

We show the success rate vs. collateral damage plots for the experiments described in Sections 5.3.1 and 5.3.3.

Figures 13 and 14 show the results for an adversary that performs only resource addition against ADGRAPH (with only structural features) and WEBGRAPH respectively. ADGRAPH shows a higher number of successes for the adversary (44 pages with success rate $> 50\%$ as compared to 30 for WEBGRAPH). At the same time, ADGRAPH also shows a higher amount of collateral damage (which is not beneficial for the adversary) – 66 pages with non-zero collateral damage, as compared to 47 for WEBGRAPH. Hence, there is no clear-cut winner between the two classifiers in terms of robustness. However, we do see that ADGRAPH has lower successes and higher collateral damage than WEBGRAPH against the powerful adversary that can do all mutations as shown in Figure 6(a) (note that this adversary cannot be used against ADGRAPH since ADGRAPH does not use information flow edges), since this adversary targets the effective, but costly, information sharing patterns.

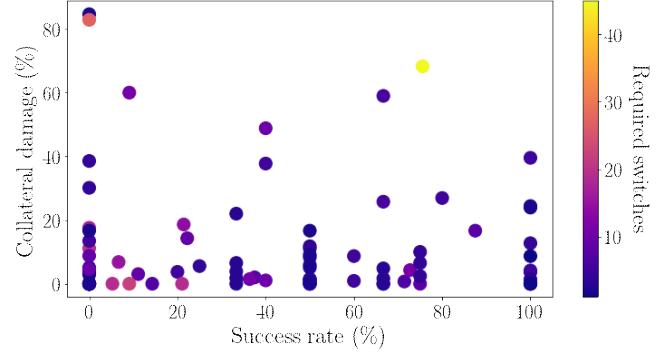


Figure 13: Adversary’s success rate vs. collateral damage for each web page in the test data at 20% graph growth, for resource addition against WEBGRAPH. Color denotes the number of required switches.

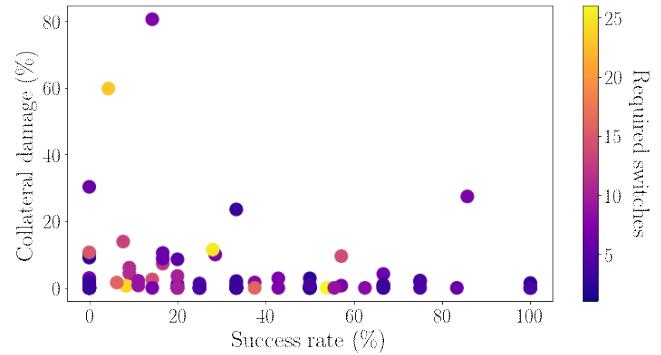


Figure 14: Adversary’s success rate vs. collateral damage for each web page in the test data at up to 20% graph growth, for resource addition against WEBGRAPH. Color denotes the number of required switches.

Figures 15 and 16 show the results for an adversary that colludes against an adversary with no collusion (Section 5.3.1). A colluding adversary shows a higher number of successes (63 pages with success rate $> 50\%$ as compared to 60 for the non-colluding adversary), and a lower collateral damage (9 pages with damage $> 0\%$ compared to 18 pages for a non-colluding adversary).

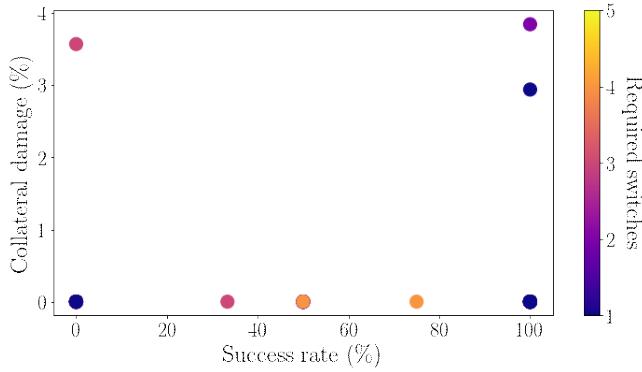


Figure 15: Adversary’s success rate vs. collateral damage for each web page in the test data at 20% graph growth, for a colluding adversary. Color denotes the number of required switches.

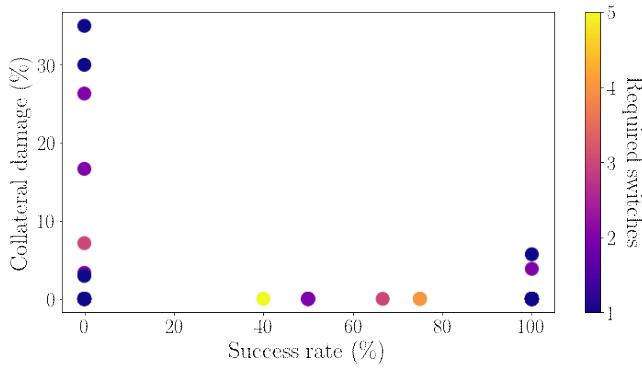


Figure 16: Adversary’s success rate vs. collateral damage for each web page in the test data at up to 20% graph growth, for a non-colluding adversary. Color denotes the number of required switches.