ADGRAPH: A Graph-Based Approach to Ad and Tracker Blocking

Umar Iqbal*† Peter Snyder† Shitong Zhu‡ Benjamin Livshits†¶ Zhiyun Qian‡ Zubair Shafiq*

*University of Iowa †Brave Software ‡UC Riverside ¶Imperial College London

***Abstract*—User demand for blocking advertising and tracking online is large and growing. Existing tools, both deployed and described in research, have proven useful, but lack either the completeness or robustness needed for a general solution. Existing detection approaches generally focus on only one aspect of advertising or tracking (e.g. URL patterns, code structure), making existing approaches susceptible to evasion.**

In this work we present ADGRAPH, a novel graph-based machine learning approach for detecting advertising and tracking resources on the web. ADGRAPH differs from existing approaches by building a graph representation of the HTML structure, network requests, and JavaScript behavior of a webpage, and using this unique representation to train a classifier for identifying advertising and tracking resources. Because ADGRAPH considers many aspects of the context a network request takes place in, it is less susceptible to the single-factor evasion techniques that flummox existing approaches.

We evaluate ADGRAPH on the Alexa top-10K websites, and find that it is highly accurate, able to replicate the labels of human-generated filter lists with 95.33% accuracy, and can even identify many mistakes in filter lists. We implement ADGRAPH as a modification to Chromium. ADGRAPH adds only minor overhead to page loading and execution, and is actually faster than stock Chromium on 42% of websites and AdBlock Plus on 78% of websites. Overall, we conclude that ADGRAPH is both accurate enough and performant enough for online use, breaking comparable or fewer websites than popular filter list based approaches.

I. INTRODUCTION

The need for content blocking on the web is large and growing. Prior research has shown that blocking advertising and tracking resources improves performance [26], [43], [56], privacy [35], [42], [52], and security [44], [54], in addition to making the browsing experience more pleasant [23]. Browser vendors are increasingly integrating content blocking into their browsers [41], [57], [63], and user demand for content blocking is expected to grow in future [33], [34].

While existing content blocking tools are useful, they are vulnerable to practical, realistic countermeasures. Current techniques generally block unwanted content based on URL patterns (using manually-curated filter lists which contain rules that describe suspect URLs), or patterns in JavaScript behavior or code structure. Such approaches fail against adversaries who rotate domains quickly [39], proxy resources through trusted domains (e.g. the first party, CDNs) [20], or restructure or obfuscate JavaScript [51], among other common techniques.

As a result, researchers have proposed several alternative approaches to content blocking. While these approaches are interesting, they are either incomplete or susceptible to trivial circumvention from even mildly determined attackers. Existing proposals suggest filter lists, pre-defined heuristics, and machine learning (ML) approaches that leverage network or code analysis for identifying unwanted web content, but fail to consider enough context to avoid trivial evasions.

This work presents ADGRAPH, an accurate and performant graph-based ML approach for detecting and blocking unwanted (advertising and tracking) resources on the web. ADGRAPH makes blocking decisions using a novel graph representation of a webpage's past and present HTML structure, the behavior and interrelationships of all executed JavaScript code units, and the destination and cause of all network requests that have occurred up until the considered network request. This contextually-rich blocking approach allows ADGRAPH to both identify unwanted resources that existing approaches miss, and makes ADGRAPH more robust against simple evasions that flummox existing approaches.

ADGRAPH is designed for both online (i.e. in-browser, during page execution) and offline (i.e. for filter list construction) deployment. ADGRAPH is performant enough for online deployment; its performance is comparable to stock Chromium and better than Adblock Plus. ADGRAPH can also be used offline to create or augment filter lists used by extension-based content blocking approaches. This dual deployment strategy can benefit users of ADGRAPH directly as well as users of extension-based content blocking approaches.

This work makes the following contributions to the problem of identifying and blocking advertising and tracking resources on the web.

1) A **graph-based ML approach** to identify advertising and tracking resources in websites based on the HTML structure, JavaScript behavior, and network requests made during execution.
2) A **large scale evaluation** of ADGRAPH's ability to detect advertising and tracking resources on popular websites. We find that ADGRAPH is able to replicate the labels of human-generated filter lists with 95.33% accuracy. Further, ADGRAPH is able to outperform existing filter lists in many cases, by correctly distinguishing ad/tracker resources from benign resources in cases where existing filter lists err.
3) A **performant implementation** of ADGRAPH as a patch to Chromium.[1] Our approach modifies the Blink and V8 components in Chromium to instrument and attribute document behavior in a way that exceeds existing practical approaches, without significantly affecting browser performance. ADGRAPH loads pages faster than stock

[1]Since ADGRAPH is designed and implemented in Chromium, it can be readily deployed on other Chromium based browsers (e.g. Chrome, Brave).

DOI 10.1109/SP40000.2020.00005

Chromium on 42% of pages, and faster than AdBlock Plus on 78% of pages.

4) A **breakage analysis** of AdGraph's impact on popular websites. AdGraph has a noticeable negative affect on benign page functionality at rates similar to filter lists (affecting 15.0% versus 11.4% of websites respectively) and majorly affects page functionality less than filter lists (breaking 5.9% versus 6.4% websites, respectively).

The rest of this paper is structured as follows. Section II presents existing work on the problem of ad and tracker blocking, and discusses why existing approaches are insufficient as comprehensive blocking solutions. Section III describes the design and implementation of AdGraph. Section IV presents an evaluation of AdGraph's effectiveness as a content blocking solution, in terms of blocking accuracy, performance, and effect on existing websites. Section V describes AdGraph's limitations, how AdGraph can be further improved, and potential uses for AdGraph in offline scenarios. Section VI concludes the paper.

II. Background and Related Work

A. Problem Difficulty

Ad and tracker blocking is a well studied topic (e.g. [36], [37], [45], [46], [49], [58], [64], [65]). However, existing work is insufficient to form a comprehensive and robust blocking solution.

Many existing approaches (e.g. [37], [45]) are vulnerable to commonly deployed countermeasures, such as evading domain-based blocking through domain generation algorithms (DGA) [39], hosting tracking related code on the first-party domain [20], spreading tracking related behavior across multiple code units, and code obfuscation [51]. Much related work in the area is unable to reason about domains that host both "malicious" (ads and tracking) and "benign" (functional or user desireable) content, and end up over or under labeling resources.

Other existing work (e.g. [36], [49]) lacks realistic evaluations. Sometimes this takes the form of an ambiguous comparison to ground truth (making it challenging to ascertain the usefulness of the technique as a deployable solution). Other cases target advertising or tracking, but not both together. Still other cases target only a subset of advertising or tracking related resources (e.g. scripts or images), but fail to consider other ways advertising or tracking can be carried out (e.g. iframes and CSS styling rules).

Further existing work (e.g. [46], [64]) presents a strategy for blocking resources, but lacks an evaluation of how much benign (i.e. user desirable) functionality the approach would break. This leaves a proposal for preventing a subset of an application's code from executing, without an understanding of how it effects the functioning of the overall application (user-serving or otherwise). These approaches may fail to separate the wheat from the chaff; they may prevent advertising and tracking, but at the expense of breaking desirable functionality.

The rest of this section reviews existing work on blocking advertising and tracking content on the web. Emphasis is given both on the contributions of each work, and why each work is incomplete as a deployable, real-world blocking solution.

B. Existing Blocking Techniques

This subsection describes existing tracking and advertising blocking work, categorized by the types of evasions each approach is vulnerable to. Our goal is not to lessen the contributions of existing work (which are many and significant), but merely to highlight the kinds of practical and deployed evasions each is vulnerable to, to further motivate the need for a more comprehensive solution.

Note that many blocking approaches discussed here are vulnerable to multiple evasions. In these cases, we discuss only one category of evasion the work is vulnerable to. Table I summarily compares the strengths and weaknesses of existing approaches.

Domain Based Blocking. Many existing content blocking approaches attempt to prevent advertising and tracking by identifying suspect domains (eTLD+1), and blocking all requests to resources on such domains. These approaches are insufficient for several reasons. First, determined advertising and tracking services can use DGA to serve their content from quickly changing domains that are unpredictable to the client, but known to the adversary. Such evasions trivially circumvent approaches that depend primarily, or only, on domain blocking strategies [39]. Similarly, in many cases, domain-focused approaches are easily circumvented by proxying the malicious resource through the first-party domain [20]. A comprehensive blocking solution should be able to account for both of these evasion strategies.

AdBlock Plus [1], uBlock Origin [30], Ghostery [15], and Disconnect [8] are all popular and deployed solutions that depend solely or partially on the domain of the request, and are thus vulnerable to the above discussed approaches. These approaches use filter lists, which describe hosts, paths, or both of advertising and tracking resources.

Gugelmann et al. [45] developed a ML-based approach for augmenting filter lists, by using existing filter lists as ground truth, and training a classifier based on the HTTP and domain-request behavior of additional network requests. Bhagavatula et al. [37] developed a ML-based approach for generating future domain-and-path based filter lists, using the rules in existing filter lists as ground truth. These approaches may be useful in identifying additional suspect content, but are easily circumvented by an attacker willing to take any of the domain hiding or rotating measures discussed earlier.

Yu et al. [65] described a method for detecting tracking related domains by looking for third-parties that receive similar unique tokens across a significant number of first-parties. This approach hinges on an attacker using the same receiving domain over a large number of hosting domains. Apple's Safari browser includes a similar technique called Intelligent Tracking Protection [63], that identifies tracking related domains by looking for third-party contexts that access state without user interaction. Privacy Badger [25] also identifies tracking related domains by looking for third-party domains that track users

Approach	Ad/Tracker Blocking	Domain/URL Blocking	1st,3rd Party Blocking	DGA Susceptibility	Code Structure Susceptibility	Cross JS Collaboration Susceptibility	Breakage Analysis
Bau et al. [36]	Tracker	Domain	3rd party	Yes	-	-	No (-)
Yu et al. [65]	Tracker	Domain	3rd party	Yes	No	No	Yes (25%)
Wu et al. [64]	Tracker	Domain	3rd party	Yes	No	Yes	No (-)
Shuba et al. [58]	Ads	URL	1st,3rd party	Yes	No	No	No (-)
Kaizer and Gupta [49]	Tracker	Domain	3rd party	Yes	Yes	Yes	No (-)
Ikram et al. [46]	Tracker	URL	1st,3rd party	No	Yes	Yes	No (-)
Gugelmann et al. [45]	Ads,Tracker	Domain	3rd party	Yes	No	Yes	No (-)
Bhagavatula et al. [37]	Ads	URL	1st,3rd party	Yes	No	No	No (-)

TABLE I: Comparison of the related work, including the practical evasions and countermeasures each is vulnerable to. Ad/Tracker Blocking column represents blocking of ads, trackers, or both. Domain/URL Detection column represents blocking at domain or URL level. 1st,3rd Party Blocking column, represents blocking of third-party requests, first-party requests, or both. In DGA Susceptibility, Code Structure Susceptibility, and Cross JS Collaboration Susceptibility columns, Yes and No represent that the approach's susceptibility to specified countermeasure. The Breakage Analysis column represents whether the breakage analysis was performed by the approach and their results.

(e.g., by setting identifying cookies) on three or more sites. These techniques do not attempt to block advertising, and also require that the attacker use consistent domains. Bau et al. [36] proposed building a graph of resource-hosting domains and training a ML classifier based on commonalities of third-party hosted code, again relying on hosting domains being distinct, consistent, and long lasting.

JavaScript Code Unit Classification. Other blocking approaches attempt to identify undesirable code based on the structure or behavior of JavaScript code units. Such approaches take as input a single code unit (and sometimes the resulting behavior of that code unit), and train ML classifiers for identifying undesirable code.

Blocking approaches that rely solely on JavaScript behavior or structure are vulnerable to several easy to deploy countermeasures. Most trivially, these approaches do not consider the interaction between code units. An attacker can easily avoid detection by spreading the malicious behavior across multiple code units, having each code unit execute a small enough amount of suspicious behavior to avoid being classified as malicious, and then using a final code unit to combine the quasi-identifiers into a single exfiltrated value. Examples of such work includes the approaches given by Wu et al. [64] and Kaiser et al. [49], both of whom propose ML classifiers that take as input the DOM properties accessed by JavaScript (among other things) to determined whether a code unit is tracking related.

Other approaches attempt to identify tracking-related JavaScript based on the static features of the code, such as names of cookie values, or similar sub-sections in the code. Such approaches are vulnerable to many obfuscation techniques, including using JavaScript's dynamic nature to break identifying strings and labels up across a code base, using dynamic interpretation facilities in the language (e.g. `eval`, `new Function`) to confuse static detection, or simply using different parameters for popular JavaScript post-processing tools (e.g. JSMin [22], Browserify [5], Webpack [32], RequireJS [27]). Ikram et al. [46] proposed one such vulnerable technique, by training a ML classifier to identify static features in JavaScript code labeled by existing filters lists as being tracking related, and using the resulting model to predict whether future JavaScript code is malicious.

Evaluation Issues. Much related work lacks a comprehensive and realistic evaluation. Examples include ambiguous or unstated sources of ground truth comparison (e.g. [36]), unrealistic metrics for what constitutes tracking or non-tracking JavaScript code (e.g. [46] makes the odd assumption that JavaScript code that tracks mouse or keyboard behavior is automatically benign, despite the most popular tracking libraries including the ability to track such functionality [16]), or the decision to (implicitly or explicitly) whitelist all first-party resources (e.g. [36], [65], [64], [49], [45]).

More significantly, much related work proposes resource blocking strategies, but without an evaluation of how their blocking strategy would affect the usability of the web. To name some examples, [36], [64], [58], [49], [46], [45], and [37], all propose strategies for automatically blocking web resources in pages, without determining whether that blocking would harm or break the user-serving goals of websites ([65] is an laudable exception, presenting an indirect measure of site breakage by way of how often users disabled their tool when browsing). Work that presents how much *bad* website behavior an approach avoids, without also presenting how much *beneficial* behavior the approach breaks, is ignoring one half of the ledger, making it difficult to evaluate each work as a practical, deployable solution.

C. JavaScript Attribution

We next present existing work on a related problem of attributing DOM modifications to responsible JavaScript code units. JavaScript attribution is a necessary part of the broader problem of blocking ads and trackers, as its necessary to trace DOM modifications and network requests back to their originating JavaScript code units. Without attribution, it is difficult-to-impossible to understand which party (or element) is responsible for which undesired activity.

While there have been several efforts to build systems to attribute DOM modifications to JavaScript code units, both in peer-reviewed literature and in deployed software, all existing approaches suffer from completeness and correctness issues.

Below we present existing JavaScript attribution approaches and discuss why they are lacking.

JavaScript Stack Walking. The most common JavaScript attribution technique is to interpose on the prototype chain of the methods being observed, throw an exception, and walk the resulting stack object to determine what code unit called the modified (i.e. interposed on) method. This technique is used, for example, by Privacy Badger [25]. The technique has the benefit of not requiring any browser modifications, and of being able to run "online" (e.g. the attribution information is available during execution, allowing for runtime policy decisions).

Unfortunately, stack walking suffers from correctness and completeness issues. First, there are many cases where calling code can mask its identity from the stack, making attribution impossible. Examples include eval'ed code and functions the JavaScript runtime decides to inline for performance purposes. Malicious code can be structured to take advantage of these shortcomings to evade detection [40].

Second, stack walking requires that code be able to modify the prototype objects in the environment, which further requires that the attributing (stack walking) code run before any other code on the page. If untrusted code can gain references to unmodified data structures (e.g. those not interposed on by the attributing code), then the untrusted code can again avoid detection. Browsers do not currently provide any foolproof way of allowing trusted code to restrict untrusted code from accessing unmodified DOM structures. For example, untrusted code can gain access to unmodified DOM structures by injecting subdocuments and extracting references to from the subdocument, before the attributing code can run in the subdocument.

AdTracker. Recent versions of Chromium include a JavaScript attribution system called AdTracker [17], which attributes DOM modifications made in the Blink rendering system to JavaScript code execution in V8, the browser's JavaScript engine. AdTracker is used by Chromium to detect when third party code modifies the DOM in a way that violates Google's ad policy [57], such as when JavaScript code creates large overlay elements across the page. The code allows the browser to determine which code unit on the page is responsible for the violating changes, instead of holding the hosting page responsible.

AdTracker achieves correctness but lacks completeness. In other words, the cases where AdTracker can correctly do attribution are well defined, but there are certain scenarios where AdTracker is not able to maintain attribution. At a high level, AdTracker can do attribution in *macrotasks*, but not in *microtasks*. Macrotasks are a subset of cases where V8 is invoked by Blink or when one function invokes another within V8. Microtasks can be thought of as an inlining optimization used by V8 to save stack frames, and is used in cases like callback functions in native JavaScript APIs (e.g. callback functions to `Promises`). Effectively, AdTracker trades completeness for performance,[2] which means that a trivial code transformation can circumvent AdTracker.

JSGraph. JSGraph [53] is designed for offline JavaScript attribution. At a high level, JSGraph instruments locations where control is exchanged between Blink and V8, noting which script unit contains the function being called, and treating all subsequent JavaScript functionality as resulting from that script unit. At the next point of transfer from Blink to V8, a new script unit is identified, and following changes are attributed to the new script.

JSGraph writes to a log file, which makes it potentially useful for certain types of offline forensic analysis, but not useful for online content blocking. More significantly, JSGraph suffers from correctness and completeness issues. First, like AdTracker, JSGraph does not provide attribution for functionality optimized into microtasks. Second, JSGraph's attribution provides incorrect results (e.g. unable to link eval'ed created script in a callback to its parent script) in the face of other V8 optimizations, such as deferred parsing, where V8 compiles different sections of a single script unit at different times. Third, JSGraph mixes all frames and subframes loaded in a page together, causing confusion as to which script is making which changes (the script unit identifier used by JSGraph is re-used between frames, so different scripts in different frames can have the same identifier in the same log file).

III. AdGraph Design

In this section we present the design and implementation of AdGraph, an in-browser ML-based approach to block ad and tracking related content on the web. We first describe a novel graph representation of the execution of a website that tracks changes in the HTML structure, behavior and interaction between JavaScript code, and network requests of the page over time. This graph representation allows for tracing the provenance of any DOM change to the responsible party (e.g. JavaScript code, the parser, a network request). Second, we discuss the Chromium instrumentation needed to construct our graph representation. Third, we describe the features AdGraph extracts from our graph representation to distinguish between ad/tracker and benign resources. Finally, we explain the supervised ML classifier and how AdGraph enforces its classification decisions at runtime. Figure 1 gives an architectural overview of AdGraph.

A. Graph Representation

Webpages are parsed and represented as DOM trees in modern browsers. The DOM tree captures relationships among HTML elements (e.g. parent-child, sibling-sibling). In AdGraph, we enrich this existing tree-representation with additional information about the execution and communication of the page, such as edges to capture JavaScript's interactions with HTML elements, or which code unit triggered a given network request. These edge additions transform the DOM

[2]These shortcomings are known to the Chromium developers, and are an intentional tradeoff to maximize performance.

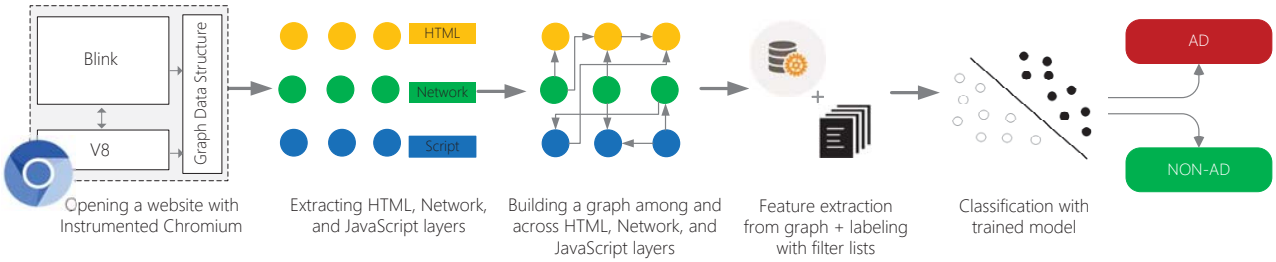


Fig. 1: AdGraph: Our proposed approach for ad and tracking blocking. We instrument Chromium to extract information from HTML structure, network, and JavaScript behavior of a webpage execution as a graph representation. We then extract distinguishing structural and content features from the graph and train a ML model to detect ads and trackers.

tree to a graph. AdGraph uses this graph representation to capture the execution of a webpage.

AdGraph's graph representation of page execution tracks changes in the website's HTML structure, network requests, and JavaScript behavior. The unique graph structure brings several benefits. First, because the graph contains information about the cause and content of every network request and DOM modification during the page's life cycle, the graph allows for tracing the provenance of any change or behavior back to either the responsible JavaScript code unit, or, in the case of initial HTML text, the browser's HTML parser. Second, the graph representation allows for extraction of context-rich features, which are used by AdGraph to identify advertising and tracking related network requests. For example, the graph allows for quick determinations of the source script sending an `AJAX` request, the position, depth, and location of an image request, and whether a subdocument was injected in a page from JavaScript code, among many others. The contextual information captured by these features in AdGraph far exceeds what is available to existing blocking tools, as discussed in Section II.

Next, we explain how AdGraph represents information during a page load as nodes and edges in a graph.

Nodes. AdGraph depicts all elements in a website as one of four types of node: *parser*, *HTML*, *network*, or *script*.

The parser node is a single, special case node that AdGraph uses to attribute document changes and network requests to the HTML parser, instead of script execution. Each graph contains exactly one parser node.

HTML nodes represent HTML elements in the page, and map directly onto the kinds of tags and markup that exist in websites. Examples of HTML nodes include image tags, anchor tags, and paragraph tags. HTML nodes are annotated to store information about the tag type and the tags HTML attributes (e.g. `src` for image tags, `class` and `id` for all tags, and `value` for input tags). HTML text nodes are represented as a special case HTML node, one without a tag type.

Network nodes represent remote resources, and are annotated with the type of resource being requested. Requests for sub-documents (i.e. `iframes`), images, `XMLHTTPRequest` fetches, and others are captured by network nodes.

Script nodes represent each compiled and executed body of JavaScript code in the document. In most cases, these can be thought of as a special type of HTML node, since most scripts in the page are tied to script tags (whether inline or remotely fetched). AdGraph represents script as its own node type though to also capture the other sources of script execution in a page (e.g. `javascript:` URIs).

Edges. AdGraph uses edges to represent the relationship between any two nodes in the graph. All edges in AdGraph are directed. Depending on the execution of pages, the graph may contain cycles. All edges in AdGraph are of one of three types, *structural*, *modification*, and *network*.

Structural edges describe the relationship between two HTML elements on a page (e.g. two HTML nodes). Mirroring the DOM API, edges are inserted to describe parent-child node relationships, and the order of sibling nodes.

Modification edges depict the creation, insertion, removal, deletion, and attribute modification of each HTML node. Each modification edge notes the type of event (e.g. node creation, node modification, etc) and any additional information about the event (e.g. the attributes that were modified, their new values, etc). Each modification edge leaves a script or parser node, and points to the HTML element being modified.

Network edges depict the browser making a request for a remote resource (captured in the graph as a network node). Network edges leave the script or HTML node responsible for the request being made, and point to the network node being requested. Network edges are annotated with the URL being requested.

Composition Examples. These four node types and three edge types together depict changes to DOM state in a website. For example, AdGraph represents an HTML tag `<img src="/example.png">` as an HTML node depicting the `img` tag, a network node depicting the image, and a network edge, leaving the former and pointing to the latter, annotated with the "`/example.png`" URL. As another example, a script modifying the value of a form element would be represented as a script node depicting the relevant JavaScript code, an HTML node describing the form element being modified, and a modification edge describing a modification

event, and the new value for the "value" attribute.

B. Graph Construction

AdGraph's graph representation of page execution requires low level modifications to the browser's fetching, parsing, and JavaScript layers. We implement AdGraph as a modification to the Chromium web browser.[3] The Chromium browser consists of many sub-projects, or modules. The Blink [6] module is responsible for performing network requests, parsing HTML, responding to most kinds of user events, and rendering pages. The V8 [7] module is responsible for parsing and executing JavaScript. Next, we provide a high level overview of the types and scope of our modifications in Chromium for constructing AdGraph's graph representation.

Blink Instrumentation. We instrument Blink to capture anytime a network request is about to be sent, anytime a new HTML node is being created, deleted or otherwise modified (and noting whether the change was due to the parser or JavaScript execution), and anytime control was about to be passed to V8. We further modify each page's execution environment to bind the graph representation of the page to each page's document object. This choice allows us to easily distinguish scripts executing in different frames/sub-documents, a problem that has frustrated prior work (see discussion of JSGraph in Section II-C). Finally, we add instrumentation to allow us to map between V8's identifers for script units, and the sources of script in the executing site (e.g. script tags, eval'ed scripts, script executed by extensions).[4]

V8 Instrumentation. We also modify V8 to add instrumentation points to allow us to track anytime a script is compiled, and anytime control changes between script units. We accomplish this by associating every function and global scope to the script they are compiled from. We then can note every time a new scope is entered, and attribute any document modifications or network requests to that script, until the scope is exited.

V8 contains several optimizations that make this general approach insufficient. First, V8 sometimes defers parsing of subsections of JavaScript code. A partial list of such cases includes eval'ed code, code compiled with the `Function` constructor, and anonymous functions provided as callbacks for some built in functions (e.g. `setTimeout`). To handle these cases, AdGraph not only maps functions to script units but also sub-scripts to scripts.

Second, V8 implements microtasks that make attribution difficult. Microtasks allow for some memory savings (much of the type information and vtable look-up overhead is skipped) and reduce some book-keeping overhead. Tracking attribution of DOM changes in microtasks is difficult because, at this level, V8 no longer tracks functions as C++ objects, but as compiled bytecode, requiring a different approach to determining which script unit "owned" any given execution. AdGraph solves this problem through additional instrumentation, and some runtime stack scanning, yielding completeness at the cost of a minor performance overhead.

JavaScript Attribution Example. AdGraph is able to attribute DOM modifications and network events to script units in cases where existing techniques fail. We give a representative example in code snippet 1.

This code uses `eval` to parse and execute a string as JavaScript code. The resulting code uses a `Promise` in a `setTimeout` callback. This `Promise` callback is optimized in V8 as a microtask, which evades the attribution techniques used in current work (e.g. PrivacyBadger / stack walking, AdTracker, JSGraph, discussed in Section II-C). Existing tools would not be able to recognize that this code unit was responsible for the image fetched in the `Promise` callback.

AdGraph, though, is able to correctly attribute the image request to this code unit. Figure 2 shows how this execution pattern would be stored in AdGraph. Specifically, the edge between nodes 2 and 4 records the attribution of the `eval` call to the responsible JavaScript code unit, and the edge between nodes 7 and 9 in record that the image request is a result of code executed in the microtask. Existing approaches would either miss the edge between 2 and 4, or 7 or 9.

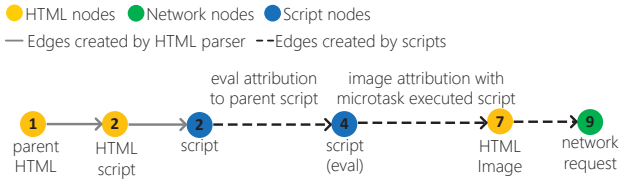


Fig. 2: AdGraph's representation of example code snippet 1. Node numbers correspond to line numbers in code snippet 1. This example highlights connections and attributions not possible in existing techniques.

```
1  <html>
2     <script>
3          ...
4          eval("setTimeout(function xyz() {
5          const p = Promise.resolve('A');
6          p.then(function abc(_) {
7            var img = document.createElement('img');
8            img.setAttribute('id','ad_image');
9            img.src = 'adnetwork.com/ad.png';
10         }) }, 5) ");
11         ...
12    </script>
13 </html>
```

Code 1: A microtask in an eval created script loading an ad.

C. Feature Extraction

Next, we present the features that AdGraph extracts from the graph to distinguish ads and trackers from functional resources. These features are designed based on our domain knowledge and expert intuition. Specifically, we manually

[3]The source code of our Chromium implementation is available at: https://uiowa-irl.github.io/AdGraph/.

[4]The architectural independence between the V8 and Blink projects made this an unexpectedly difficult problem to solve, with many unanticipated corner cases that were not discovered until we subjected AdGraph to extensive automatic and manual testing.

analyze a large number of websites and try to design features that would distinguish ad/tracking related resources from functional (or benign) resources.

The extracted features broadly fall into two categories: "structural" (features that consider the relationship between nodes and edges in the graph) and "content" (features that depend on the values and attributes of nodes in isolation from their connections). In total we extract 64 structural and content features. Table II gives a summary and representative examples of features from each category. Below we provide a high-level description of structural and content features. More detailed analysis of features and their robustness is presented in Section IV-D.

Structural Features
Graph size (# of nodes, # of edges, and nodes/edge ratio)
Degree (in, out, in+out, and average degree connectivity)
Number of siblings (node and parents)
Modifications by scripts (node and parents)
Parent's attributes
Parent degree (in, out, in+out, and average degree connectivity)
Sibling's attributes
Ascendant's attributes
Descendant of a script
Ascendant's script properties
Parent is an eval script
Content Features
Request type (e.g. `iframe`, `image`)
Ad keywords in request (e.g. banner, sponsor)
Ad or screen dimensions in URL
Valid query string parameters
Length of URL
Domain party
Sub-domain check
Base domain in query string
Semi-colon in query string

TABLE II: Summarized feature set used by ADGRAPH.

Structural Features. Structural features target the relationship between elements in a page (e.g. the relationship between a network request and the responsible script unit, or a HTML nodes' parents, siblings and cousin HTML nodes). Examples of structural features include whether a node's *parents* have ad-related values for the `class` attribute, the tag names of the node's *siblings*, or how deeply nested in the document's structure a given node is.

Structural features also consider the interaction between JavaScript code, and the resource being requested. These features rely on ADGRAPH's instrumentation of Blink and V8. Examples of JavaScript features include whether the node initiating a network request was inserted by JavaScript code, the number of scripts that have "touched" the node issuing the request, and, in the case of requests that are not directly related to HTML elements (e.g. AJAX), whether the JavaScript code initiating the request was inlined in the document or fetched from a third-party.

Content Features. Content features relate to values attached to individual nodes in the graph (and not the connections *between* nodes in the graph). The most significant value considered is the URL of the resource being requested. These content features are similar to what most existing content blocking tools use. ADGRAPH's specific set of features though is unique. Examples of ADGRAPH's content features include whether the origin of the resource being requested is first-or-third party, the number of path segments in the URL being requested, and whether the URL contains any ad-related keywords.

D. Classification

ADGRAPH uses random forest [38], a well-known ensemble supervised ML classification algorithm. Random forest combines decisions from multiple decision trees, each constructed using a different bootstrap sample of the data, by choosing the mode of the predicted class distribution. Each node for a decision tree is split using the best among the subset of features selected at random. This feature selection mechanism provides robustness against over-fitting issues. We configure random forest as an ensemble of 100 decision trees with each decision tree trained using $int(\log M + 1)$ features, where M is the total number of features.

ADGRAPH's random forest model classifies network requests based on the provenance (creation and modification history) of a node and the context around it. These classification decisions are made before network request are sent, so that ADGRAPH can prevent network communication with ad and tracking related parties. A single node may initiate many network requests (either due to it being a script node, or being modified by script to reference multiple resources). As a result, any node may be responsible for an arbitrary number of network requests. ADGRAPH classifies three categories of network requests:

1) Requests initiated by the webpage's HTML (e.g. the image referenced by an `<img>` tag's `src` attribute).
2) Requests initiated by a node's attribute change (e.g. a new background image being downloaded due to a new CSS style rule applying because of a mouse hover).
3) Requests initiated directly by JavaScript code (e.g. `AJAX` requests, image objects not inserted into the DOM).

IV. ADGRAPH EVALUATION

In this section we evaluate the accuracy, usability, and performance of ADGRAPH when applied to live, real-world, popular websites.

A. Accuracy

We first evaluate how accurately ADGRAPH is able to distinguish advertising and tracking content from benign web resources.

Ground Truth. To evaluate ADGRAPH's accuracy, we first need to gather a ground truth to label a large number of ad/tracking related network requests. We generate a trusted set of ground truth labels by combining popular crowdsourced filter lists that target advertising and/or tracking, and applying them to popular websites. Table III lists the 8 popular filter lists

we combine to form our ground truth. These lists collectively contain more than a hundred thousand crowdsourced rules for determining whether a URL serves advertising and/or tracking content.

List	# Rules	Citation
EasyList	72,660	[9]
EasyPrivacy	15,507	[10]
Anti-Adblock Killer	1,964	[2]
Warning Removal List	378	[31]
Blockzilla	1,155	[3]
Fanboy Annoyances List	38,675	[12]
Peter Lowe's List	2,962	[24]
Squid Blacklist	4,485	[29]

TABLE III: Crowd sourced filter lists used as ground truth for identifying ad and tracking resources. Rule counts are as of Nov. 12, 2018.

Advertising resources include audio-visual promotional content on a website. Tracking resources collect unique identifiers (e.g., cookies) and sensitive information (e.g., browsing history) about users. In practice, there is no clear division between ad and tracking resources. Many resources on the web not only serve advertising images and videos but also track the users who view it. It is also noteworthy that EasyList (to block ads) and EasyPrivacy (to block trackers) have a significant overlap. Because of this overlap, we do not attempt to distinguish between advertising and tracking resources.

Note that while these crowdsourced filter lists suffer from well-known shortcomings [62], we treat them as "trusted" for three reasons. First, they are reasonably accurate for top-ranked websites even though they suffer on low-ranked websites [42], [54]. Second, a more accurate alternative, building a web-scale, manually generated, expert set of labels would require labor and resources far beyond what is feasible for a research project. Third, we use several filter lists together to maximize their coverage and reduce false negatives.

We visit the homepages of the Alexa top-10K websites with our instrumented Chromium browser. We expect that the top-10K websites is a diverse and large enough set to contain most common browsing behaviors. We limit our sample of websites to the 10K most popular sites to avoid biasing our sample; previous work has found that popular filter lists work reasonably well for popular sites [42], [54]. Applying crowdsourced filter lists to unpopular sites (sites that, almost by definition, the curators of filter lists are less likely to visit) risks skewing our data set to include a large number of false negatives (i.e. advertising and tracking resources that filter list authors have not encountered).

We apply filter lists to websites in the following manner. We visit the homepage of each site with our instrumented version of Chromium and wait for each page to finish loading (or 120 seconds, whichever occurs first). Next we record every URL of every resource fetched when loading and rendering each page. We then label each fetched resource URL as AD and NON-AD, based on the whether they are identified as ad or tracking related by any of a set of filter lists. Our final labeled dataset consists of 540,341 URLs, fetched from 8,998 successfully crawled domains.[5]

Results. We use the random forest model to classify each fetched URL. We then compare each predicted label with the label derived from our ground truth data set, the set of filter lists described above. We then evaluate how accurately our model can reproduce the filter list labels through a stratified 10-fold cross validation, and report the average accuracy. ADGRAPH classifies AD and NON-AD with a high degree of accuracy, achieving 95.33% accuracy, with 89.1% precision, and 86.6% recall.

As Table IV shows, ADGRAPH classifies web resources with a high degree of accuracy. We note that ADGRAPH is more accurate in classifying visual resources such as images (98.95% accuracy) and CSS (96.32% accuracy) than invisible resources like JavaScript (90.52% accuracy) and AJAX requests (93.55% accuracy). This suggests an interesting possibility, that ADGRAPH's labels are correct, and filter lists miss-classify invisible resources due to their reliance on human crowdsourced feedback. We investigate this possibility, and more broadly the causes of disagreements between ADGRAPH and filter lists in the next subsection.

B. Disagreements Between ADGRAPH *and Filter Lists*

We now manually analyze the cases where ADGRAPH disagrees with filter lists to determine which labeling is incorrect, ADGRAPH's or filter lists'. Overall, we find that ADGRAPH is able to identify many advertising and tracking resources missed by filter lists. We also find that ADGRAPH correctly identifies many resources as benign which filter lists incorrectly block. These findings imply that ADGRAPH's actual accuracy is higher than 95.33%.

Methodology. To understand why ADGRAPH disagrees with existing filter lists, we perform a manual analysis of a sample of network requests where ADGRAPH identifies a resource as ad/tracking related but filter lists identify as benign (i.e. false positives) and where filter lists identify a resource as ad/tracking related but ADGRAPH identifies as benign (i.e. false negatives). We select these "false positives" and "false negatives" from the most frequent advertising and tracking related resource types: JavaScript code units and images. We manually analyze all of the 282 distinct images and a random sample of 100 script URLs that ADGRAPH classifies as AD but filter lists label as NON-AD and a random sample 300 images and 100 script URLs that ADGRAPH classifies as NON-AD but filter lists label as AD. The goal of our manual analysis is to assign each JavaScript unit or image to one of the following labels:

1) **True Positive**: ADGRAPH's classification is correct and the filter lists are incorrect; the resource is related to advertising or tracking.

[5]The success rate of about 90% in our crawl is in line with those of previous studies [42], [54].

Resource	# Resources	Blocked by Filter Lists	Blocked by AdGraph	Precision	Recall	FPR	FNR	Accuracy
Image	201,785	11,584	10,228	93.09%	88.29%	0.39%	11.71%	98.95%
Script	167,533	67,959	60,030	88.32%	88.33%	7.97%	11.67%	90.52%
CSS	124,207	9,255	5,834	83.61%	63.03%	0.99%	36.97%	96.32%
AJAX	24,365	8,305	7,442	91.31%	89.60%	4.40%	10.40%	93.55%
iFrame	20,091	7,745	7,244	92.31%	93.53%	4.88%	6.47%	94.50%
Video	2,360	23	14	93.33%	60.86%	0.04%	39.14%	99.57%
Total	540,341	104,871	90,792	89.1%	86.6%	2.56%	13.4%	95.33%

TABLE IV: Number of resources, broken out by type, encountered during our crawl, and incidence of ad and tracking content, as determined by popular filter lists and AdGraph.

2) **False Positive**: The label by filter lists is correct and AdGraph's classification is incorrect; the resource is not related to advertising or tracking.
3) **True Negative**: AdGraph's classification is correct and the filter lists are incorrect; the resource is not related to advertising or tracking.
4) **False Negative**: The label by filter lists is correct and AdGraph's classification is incorrect; the resource is related to advertising or tracking.
5) **Mixed**: The resource is dual purpose (i.e. both ad/tracker and benign). This label is only used for script resources.
6) **Undecidable**: It was not possible to determine whether the resource is an ad/tracker.

We decide whether an image was advertising or tracking related through the following three steps. First, we label all tracking pixels (1×1 sized images used to initiate a cookie or similar state-laden communication) as "true positive" if AdGraph classified it as AD and "false negative" if AdGraph classified it as NON-AD. Second, we consider the content of each image and look for text indicating advertising, such as the word "sponsored", prices, or mentions of marketers. If the image has such text, we consider the image as an advertisement and label it "true positive" if AdGraph classified it as AD and "false negative" if AdGraph classified it as NON-AD. If the case is ambiguous, such as an image of a product that could either be advertising or a third-party discussion of the product, we use the "undecidable" label. Third, we label all remaining cases as "false positive" if AdGraph classified them as AD and "true negative" if AdGraph classified them as NON-AD.

Deciding the labels for the sampled script resources is more challenging. Determining the purpose of a JavaScript file requires inspecting and understanding large amounts of code, most of which has no documentation, and which is in many cases minified or obfuscated. We label a script as "true positive" (advertising or tracking related) if most of the script performs any of the following functionality: cookie transmission, passive device fingerprinting, communication with known ad or tracking services, sending beacons, or modifying DOM elements whose attributes are highly indicative of an ad (e.g. creating an image carousel with the `id` "ad-carousel"); and AdGraph classified it as AD and "false negative" if AdGraph classified it as NON-AD. If the script primarily includes functionality distinct from the above (e.g. form validation, non-ad-related DOM modification, first-party AJAX server communication), we label it as "false positive" if AdGraph classified it as AD and "true negative" if AdGraph classified it as NON-AD. If the script contains significant amounts of both categories of functionality, we label the script as "mixed". In cases where the functionality is not discernable, we use the "undecidable" label.

False Positive Analysis. Table V presents the results of our disagreement analysis for false positives. In cases where AdGraph identifies a resource as suspect, and filter lists label it as benign, AdGraph's determination is correct 11.0%–33.0% of the time for JavaScript and 46.8% of the time for images.

AdGraph is often able to detect advertising and tracking resources that are missed by filter lists. For example, AdGraph blocks a 1x1 pixel on cbs.com that includes a tracking identifier in its query string. In another example, AdGraph blocks a script (js1) on nikkan-gendai.com that performs browser fingerprinting. Filter lists likely missed these resources because they are often slow to catch up when websites introduce changes [47].

There are however several false positives that are actual mistakes by AdGraph. For example, AdGraph blocks a third-party dual purpose script (avcplayer.js), a video player library that also serves ads, on inquirer.net. Interestingly, AdGraph detects many such dual-purposed scripts that are beyond the ability of binary-label filter lists.

These results demonstrate that AdGraph is able to identify many edge case resources (e.g. mixed-use) that can be used to refine future versions of AdGraph. As discussed in Section V-B, AdGraph can be extended to handle such mistakes by implementing more fine-grained blocking.

	Image		Script	
	#	%	#	%
True Positive	132	46.8%	11	11.0%
False Positive	129	45.7%	63	63.0%
Mixed	0	0%	22	22.0%
Undecidable	21	7.4%	4	4.0%

TABLE V: Results of manual analysis of a sample of cases where AdGraph classifies a resource as AD and filter lists label it as NON-AD.

False Negative Analysis. Table VI presents the results of our disagreement analysis for false negatives. In cases where

ADGRAPH identifies a resource as benign, and filter lists label it as suspect, ADGRAPH's determination is correct 22%–32% of the time for JavaScript and 27.7% of the time for images.

Again, ADGRAPH is often able to identify benign content that is incorrectly over-blocked by filter lists. For example, ADGRAPH does not block histats.com when visited as a first-party in our crawl, but this domain is blanketly blocked by the Blockzilla filter list even when visited as a first-party. In another example, ADGRAPH does not block a social media icon facebook-gray.svg (served on postimees.ee as a first-party resource) and a privacy-preserving analytics script piwik.js (served on futbol24.com as a first-party resource). It can be argued that many of these resources are neither ads nor pose a tracking threat [11], [50]. Filter lists over-block in such cases because of the inclusion of overly broad rules (e.g. blocking entire domains, or any URL containing a given string).

There are however several false negatives that are actual mistakes by ADGRAPH. For example, ADGRAPH misses fingerprint2.min.js served by a CDN cloudflare.com on index.hr. ADGRAPH likely made this mistake because a popular third-party CDN, which is typically used to serve functional content, is used to serve a fingerprinting script. As discussed in Section V-B, ADGRAPH can be extended to handle such mistakes by extracting new features from JavaScript APIs.

	Image		Script	
	#	%	#	%
True Negative	83	27.7%	22	22%
False Negative	180	60.0%	55	55%
Mixed	0	0%	10	10%
Undecidable	37	12.3%	13	13%

TABLE VI: Results of manual analysis of a sample of cases where ADGRAPH classifies a resource as NON-AD and filter lists label it as AD.

C. Site Breakage

Content blocking tools carry the risk of breaking benign site functionality. Content blockers prevent resources that the website expects to be in place from being retrieved, which can have the carry over effect of harming desireable site functionality, especially when tools mistakenly block benign resources [19]. Thus assessing the usefulness of a content blocking approach must also include an evaluation of how many sites are "broken" by the intervention.

Next we evaluate how often, and to what degree, ADGRAPH breaks benign (i.e. user desired) website functionality. We do so by having two human reviewers visit a sample of popular websites using ADGRAPH, and having them independently record their assessment of whether the site worked correctly. We find that ADGRAPH only affects benign functionality on a small number of sites, and at a rate equal to or less than popular filter lists.

Methodology. We estimate how many sites ADGRAPH breaks by having two evaluators use ADGRAPH on a sample of popular websites and independently record their determination of how ADGRAPH impacts the site's functionality. Because of the time consuming nature of the task, we select a smaller sample of sites for this breakage evaluation than we use for the accuracy evaluation.

Our evaluators use ADGRAPH on two sets of websites: first the Alexa top-10 websites, and second on a random sample of 100 websites from the Alexa top-1K list, resulting in a total of 110 sites for breakage evaluation.

Automatic site breakage assessment is challenging due to the complexity of modern web applications [55], [65]. Unfortunately, manual inspection for site breakage assessment is not only time-consuming but also likely to lose completeness as the functionalities of a website are often triggered by certain events that may be hard to manually cover exhaustively. As a tradeoff, we adopt the approach from [59], which is a manual analysis but focuses on *the user's perspective*. In other words, we intentionally ignore the breakages that only affect the website owner as they do not have any impact on user experience.

For each website, our evaluators independently perform the following steps.

1) Open the website with stock Chromium, as a control, and perform as many actions as possible within two minutes. We instruct our evaluators to exercise the kinds of behaviors that would be common on each site. For example, in a news site this might be browsing through an article; on a e-commerce site this might include searching for a product and proceeding to checkout etc.
2) Open the website with ADGRAPH, repeat the actions performed above, and assign a breakage level of
   (a) **no breakage** if there is no perceptible difference between ADGRAPH and stock Chromium;
   (b) **minor breakage** if the browsing experience is altered, but objective of the visit can still be completed; or
   (c) **major breakage** if objective of the visit cannot be completed.
3) Open the website with Adblock Plus[6], repeat the actions, and assign a breakage level as above.

To account for the subjective nature of this analysis, we have each evaluator visit the same sites, at similar times, and determine their "breakage" scores independently. Our evaluators give the same score 87.7% of the time, supporting the significance of their analysis.

Tool	No breakage		Major		Minor		Crash	
	#	%	#	%	#	%	#	%
ADGRAPH	93.5	85.0%	6.5	5.9%	7.5	6.8%	2.5	2.3%
Filter lists	97.5	88.6%	7	6.4%	4	3.6%	1.5	1.4%

TABLE VII: Breakdown of breakage analysis results (# columns are the average of two independent scores.)

Results. Table VII reports the site breakage assessments as the average of two reviewers. The evaluation shows that

[6]Adblock Plus is configured with the same 8 filter lists that are used to train ADGRAPH.

AdGraph and filter lists are comparable in terms of site breakage. AdGraph and filter lists do not cause any breakage on 85.0% and 88.6% of the sites, respectively. The major breakage rate (5.9%) is also on par with the filter lists (6.4%). We also note that AdGraph's breakage is much lower than other commonly used privacy oriented browsers (e.g. 16.3% for Tor Browser [59]).

D. Feature Analysis

Next, we discuss the intuitions behind some of the features used in AdGraph, and evaluate their ability to distinguish ad/tracking content from benign content. We describe some of the features that are most useful (in terms of information gain [48]) in AdGraph's predictions.

Structural Features. Two of the structural features that provided the highest information gain are a node's *average degree connectivity* and its *parents' attributes*.

We expect AD nodes to have lower *average degree connectivity*, since the interaction of these nodes is confined to only ad/tracking content, and thus appear in less connected cliques. Conversely, we expect that NON-AD nodes appear alongside, and interact with, functional content more, and thus have higher *average degree connectivity*. Our results in Figure 3(a) support this intuition. AD nodes do indeed have have lower *average degree connectivity* than NON-AD nodes.

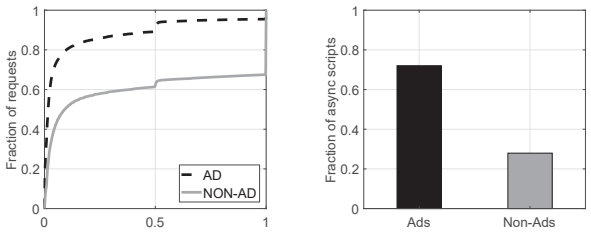
We also expect the parents of AD nodes to have different attributes than NON-AD nodes. This intuition came from the expectation that AD nodes are more likely to follow common practices and standards, such as those proposed by the Interactive Advertising Bureau (IAB) [21]. For example, IAB's LEAN standard [18] requires ad related scripts to load asynchronously (indicated by the presence of the `async` attribute on a script node). We capture this intuition in a feature by considering the attributes of each network requests' parent nodes (in our graph representation, the parent of a network request might be the script element that initiates the network request). Our results in Figure 3(b) support this intuition. The parents of AD nodes with `script` tag name were 3 times more likely to have the `async` attribute than NON-AD nodes.

We note that some structural features are more robust to obfuscation than others. For example, to flummox the classifier, it would be more challenging for an adversary to manipulate a node's average degree connectivity (which depends on all of the node's neighbors) than it would be to manipulate the attributes of a parent node.

Content Features. Two of the content features that provided the highest information gain are a node's *domain party* and its *URL length*.

We expect AD nodes to be more likely to come from third-party domains than NON-AD nodes. We capture this intuition in a boolean feature, recording whether the domain of a network request differs from domain of the first party document. Figure 4(a) shows this intuition to be correct. More than 90% of the ads came from third-party domains.

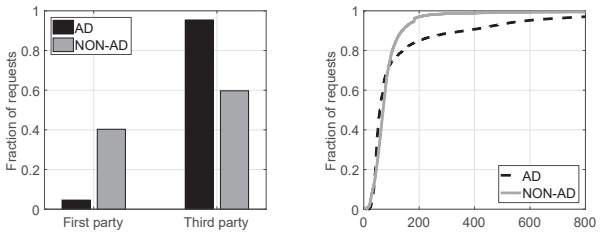
We also expect AD nodes to include a large number of query parameters in their URLs. We capture this intuition by using a request's *URL length* as a numeric feature. Figure 4(b) shows this intuition to be correct. AD node URLs were on average longer than NON-AD node URLs.



(a) average degree connectivity (b) async script

Fig. 3: Conditional distributions for structural features.

We again note that some content features are more robust to obfuscation than others. For example, to flummox the classifier, it would be more challenging for an adversary to switch ads/trackers from third-party to first-party that it would be to manipulate the length of a URL.



(a) domain party (b) length of URL

Fig. 4: Conditional distributions for content features.

Ablation Analysis. Next, we separately evaluate structural and content features in terms of their contribution to AdGraph's accuracy. To this end, we train additional classifiers separately, one using only structural features, the other using only content features. While structural features and content features have comparable accuracy they provide complementary information, which when used together improve AdGraph's accuracy. For example, excluding structural features results in a decrease of 6.6% in precision, 8.7% in recall, and 2.7% in accuracy.

We also expect structural features to be more robust than content features. Structural features consider neighboring graph structure of a node while content features only consider a node in isolation. To manipulate the structural features, an adversary would need to change the target node, its neighbors, and subsequently their neighbors. Manipulating content features would only require changing the target node.

Thus, we conclude that the graph-based representation of AdGraph, as captured by structural features, contributes to its accuracy and robustness.

E. Tradeoffs in Browser Instrumentation

Recall from Section III-B that ADGRAPH modifies Chromium to attribute DOM modifications to JavaScript code units. This is different from most existing content blocking tools that operate at the extension layer. ADGRAPH's browser instrumentation is a trade off; it gains attribution accuracy at the cost of ease of distribution. This raises the question of whether ADGRAPH can instead be implemented as a browser extension on any web browser.

We investigate this question by implementing ADGRAPH as a browser extension, using the best possible attribution option available at the extension layer (JavaScript stack walking, discussed in Section II-C). We test the accuracy of the best possible extension implementation of ADGRAPH by re-crawling the Alexa-10k with a modified version of ADGRAPH, using the same methodology described in Section IV-A. This modified version of ADGRAPH uses JavaScript stack walking to attribute DOM modifications to script units, instead of the Blink and V8 modifications. We then train and test ML classifier on the graphs constructed using JavaScript stack walking.

We compare the accuracy of this best-possible-extension implementation to our in-browser implementation of ADGRAPH. We find that implementing ADGRAPH as a browser extension significantly reduces classification accuracy. Implementing ADGRAPH as a browser extension degrades precision by 1.5%, recall by 16%, and accuracy by 2.3%. Thus, the mistakes JavaScript stack walking makes in attribution lead to more errors in classification. We conclude that costs of implementing ADGRAPH's as a set of browser modifications (i.e. difficulty in distribution) is more than offset by the benefits (i.e. increased classification accuracy), and that ADGRAPH is best implemented as Blink and V8 modifications.

F. Performance

We evaluate ADGRAPH's performance as compared to stock Chromium and Adblock Plus. ADGRAPH performs faster in most cases than the most popular blocking tool, Adblock Plus, and in many cases results in faster performance than stock Chromium. This is the result of both careful engineering in ADGRAPH's implementation, and ADGRAPH's instrumentation overhead (often) being more than offset by the network and rendering savings gained by having to fetch and render less page content (i.e. the content blocked by ADGRAPH).

To measure whether ADGRAPH is a practical blocking solution, we compare the performance of ADGRAPH, stock Chromium, and Chromium with Adblock Plus installed (using Adblock Plus's default configuration) on the Alexa 1K. Our simulated network uses a 10 Mbps downlink with a latency of 100ms. We visit the landing page of each website 10 times and record the average page load time (measured as the difference between the DOM's `navigationStart` and `loadEventEnd` events). Figure 5 presents ADGRAPH's page load time compared to stock Chromium, and Chromium with Adblock Plus.[6]

Resource Type	ADGRAPH faster	Chromium faster
Image	24.59%	14.92%
Script	20.82%	17.96%
CSS	6.47%	0.79%
AJAX	48.03%	36.14%
iFrame	37.66%	30.47%
Video	7.14%	6.20%

TABLE VIII: Comparison of average percentage of resources ADGRAPH blocks on sites where ADGRAPH outperforms Chromium, and vise versa. For all resource types, ADGRAPH performs faster when more resources are blocked.

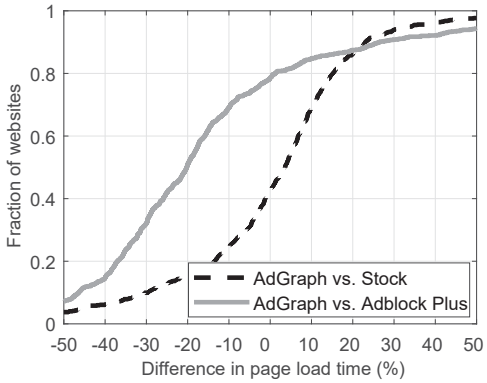


Fig. 5: Overhead ratio in terms of page load time.

ADGRAPH performs faster than Chromium on 42% of websites. ADGRAPH is often faster than stock Chromium because it needs to fetch and render fewer resources than stock Chromium (i.e. the network requests blocked by ADGRAPH). Table VIII shows that ADGRAPH outperforms Chromium on sites where it blocks more ad/tracking content, as compared to sites where it blocks less. Put differently, the more content ADGRAPH blocks, the more it is able to make up for the instrumentation and classification overhead with network and rendering savings.

ADGRAPH performs faster than Adblock Plus on 78% of websites. ADGRAPH is faster than Adblock Plus for two reasons. First, Adblock Plus implements element hiding rules (i.e. rules describing elements that are still fetched, but hidden when rendering), which carries with it an enforcement and display-reflow overhead ADGRAPH does not share. Second, ADGRAPH's blocking logic is implemented in-browser which leads to performance improvement over Adblock Plus's implementation at the extension layer.

Overall, we conclude that ADGRAPH is performant enough to be a practical online content blocking solution. Future implementation refinements, and the exploration of cheaper features, could further improve ADGRAPH's performance.

V. DISCUSSIONS

A. Offline Application of ADGRAPH

ADGRAPH is designed and implemented to be used as an online, in-browser blocking tool. This is different than most blocking tools, which operate as extensions on mainstream browsers (e.g. Chrome, Firefox). Since ADGRAPH

requires browser instrumentation, it cannot be directly used by extension-based blockers that rely on offline manually curated filter lists. ADGRAPH can benefit existing blocking tools through the creation and maintenance of filter lists in several ways.

First, the accuracy of filter lists suffers because of they are manual generated and rely on informal crowdsourced feedback. As discussed in Section IV-B, filter list maintainers can analyze disagreements between ADGRAPH and filter lists to identify and fix potential inaccuracies in filter lists

Second, ADGRAPH can support the generation of filter lists targeting under-served languages or region on the web. Filter lists are inherently skewed towards popular websites and languages because of their larger and more active blocking user base [42], [54]. Filter list maintainers receive much less feedback to fix inaccuracies on less popular websites. This makes the creation and maintenance of filter lists for underserved regions (geographically and linguistically) difficult, since these sites have less visitors. Language/region specific filter lists are updated much less frequently than general (and mostly English targeting) filter lists like EasyList. Many languages and regions (most notably Africa) do not have dedicated filter lists at all. ADGRAPH can assist in automatically generating filter lists for smaller or underserved regions.

Third, the manual nature of filter list maintenance has lead to increasing number of outdated and stale rules. Filter list rules can quickly get outdated because most websites frequently update and are highly dynamic. Prior research found that filter lists can take months to update in response to such changes [47]. Even when filter lists are updated, new rules are typically added (rather than editing old rules) which leads to accumulation of stale rules over time. Prior research reported that only 200 rules account for 90% blocking activity for EasyList [62]. In other words, the number of rare-to-never used rules in EasyList is increasing over time which has performance implications. ADGRAPH can by used by filter list maintainers to periodically audit filter lists for identifying outdated and stale rules.

B. ADGRAPH *Limitations And Future Improvements*

Ground Truth. ADGRAPH relies on filter lists as ground truth to train a ML classifier for detecting ads/trackers. As we showed in Sections IV-B, filter lists suffer from inaccuracies due to both false negatives and false positives. ADGRAPH can address these inaccuracies in ground truth by gathering valuable user feedback when it is deployed at scale. ADGRAPH can retrain its ML classifier periodically on improved ground truth as user feedback is received.

Features. The features used by ADGRAPH are manually designed, based on our domain knowledge and expert intuition, with the goal of achieving decent accuracy. Note that the feature set is by no means "complete" and there is room for additional feature engineering to further improve accuracy. New features can be systematically discovered by incorporating user feedback, which may reveal new characteristics of ads/trackers over time that are not currently covered by ADGRAPH. New features may require addition of new instrumentation points such as JavaScript APIs or new feature modalities altogether, such as image based perceptual information [28], [60], [61].

Classification Granularity. ADGRAPH is currently designed to make binary decisions to either block or allow network requests. However, as discussed in Section IV-B, ADGRAPH is also able to detect cases when a single JavaScript is used for both ad/tracking and functional content. The cases where JavaScript code serves dual-purpose are challenging because blocking the request may break page functionality, while allowing the request will allow ads/trackers on the page. ADGRAPH's context rich classification approach can be adapted to more than two labels for handling such dual-purpose scripts. Specifically, ADGRAPH can be trained at a more granular level to distinguish between ads/trackers, functional, and dual-purpose resources. ADGRAPH can respond to such dual-purpose resources with different remediations than outright allowing/blocking, such as giving those scripts a reduce set of DOM capabilities (e.g. reading/writing cookies [14], [63], access to certain APIs [4], [13]), or blocking network requests issued from such scripts.

VI. CONCLUSION

In this paper we proposed ADGRAPH, a graph-based ML approach to ad and tracker blocking. We designed ADGRAPH to leverage fine-grained interactions between network requests, DOM elements, and JavaScript code execution to construct a graph representation that is used to trace relationships between ads/trackers and the rest of the page content. To implement ADGRAPH, we instrumented Chromium's rendering engine (Blink) and JavaScript execution engine (V8) to efficiently gather complete HTML, HTTP, and JavaScript information during page load. We leveraged this rich context by extracting distinguishing features to train a ML classifier for in-browser ad and tracker blocking at runtime.

We showed that ADGRAPH not only blocks ads/trackers with 95.33% accuracy but uncovers many ad/tracker and functional resources that are missed and over-blocked by filter lists, respectively. We also showed that ADGRAPH's breakage is on par with filter lists. In addition to high accuracy and comparable breakage, we showed that ADGRAPH loads pages much faster as compared to existing content blocking tools.

We designed ADGRAPH to be used both online (for in-browser blocking) and offline (filter list curation). Since the vast majority of extension-based blocking tools currently rely on manually curated filer lists, ADGRAPH's offline use case will aid filter list monitoring and maintenance. Overall, we believe that ADGRAPH significantly advances the state-of-the-art in ad and tracker blocking.

ACKNOWLEDGMENT

This work is supported in part by the National Science Foundation under grant numbers 1715152, 1719147, and 1815131.

REFERENCES

[1] Adblock Plus. https://adblockplus.org/.

[2] Anti-Adblock Killer. https://github.com/reek/anti-adblock-killer.

[3] Blockzilla. https://zpacman.github.io/Blockzilla/.

[4] Brave Browser Fingerprinting Protection Mode. https://github.com/brave/browser-laptop/wiki/Fingerprinting-Protection-Mode.

[5] Browserify. http://browserify.org/.

[6] Chromium Blink Rendering Engine (Renderer). https://cs.chromium.org/chromium/src/third_party/blink/renderer/.

[7] Chromium V8 JavaScript Engine. https://v8.dev/.

[8] Disconnect . https://disconnect.me/.

[9] EasyList. https://easylist.to/.

[10] EasyPrivacy. https://easylist.to/easylist/easylist.txt.

[11] EFF's Open Letter to Facebook. https://www.eff.org/files/filenode/social_networks/openlettertofacebook.pdf.

[12] Fanboy Annoyances List. https://www.fanboy.co.nz/.

[13] Fingerprinting Defenses in The Tor Browser. https://www.torproject.org/projects/torbrowser/design/#fingerprinting-defenses.

[14] Firefox Storage Access Policy. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Privacy/Storage_access_policy.

[15] Ghostery. https://www.ghostery.com/.

[16] Google Analytics. https://developers.google.com/analytics/devguides/collection/analyticsjs/events.

[17] Google Chrome AdTracker. https://cs.chromium.org/chromium/src/third_party/blink/renderer/core/frame/ad_tracker.h?rcl=fabe78ea42052335674f6cc9c809dd610a8eea29&l=32.

[18] IAB Standard Ad Unit Portfolio. https://www.iab.com/wp-content/uploads/2017/08/IABNewAdPortfolio_FINAL_2017.pdf.

[19] Incorrectly Removed Content by Filer Lists. https://forums.lanik.us/viewforum.php?f=64.

[20] Instart Logic AppShield Ad Integrity. https://www.instartlogic.com/products/advertising-marketing-recovery.

[21] Interactive Advertising Bureau. http://www.iab.com/.

[22] JSMin. http://www.crockford.com/javascript/jsmin.html.

[23] PageFair, 2017 Global Adblock Report. https://pagefair.com/downloads/2017/01/PageFair-2017-Adblock-Report.pdf.

[24] Peter Lowe's list. http://pgl.yoyo.org/adservers/.

[25] Privacy Badger. https://www.eff.org/privacybadger.

[26] Putting Mobile Ad Blockers to the Test. https://www.nytimes.com/2015/10/01/technology/personaltech/ad-blockers-mobile-iphone-browsers.html.

[27] RequireJS. https://requirejs.org/.

[28] Sentinel - The artificial intelligence ad detector. https://adblock.ai/.

[29] Squid blacklist. http://www.squidblacklist.org/.

[30] UBlock Origin. https://github.com/gorhill/uBlock/.

[31] Warning removal list. https://easylist-downloads.adblockplus.org/antiadblockfilters.txt.

[32] Webpack. https://webpack.js.org/.

[33] PageFair 2015 Adblock Report. https://pagefair.com/blog/2015/ad-blocking-report/, 2015.

[34] PageFair 2016 Mobile Adblocking Report. https://pagefair.com/blog/2016/mobile-adblocking-report/, 2016.

[35] ANTHES, G. Data Brokers Are Watching You. *Communications of the ACM* (2015).

[36] BAU, J., MAYER, J., PASKOV, H., AND MITCHEL, J. C. A Promising Direction for Web Tracking Countermeasures. In *W2SP* (2013).

[37] BHAGAVATULA, S., DUNN, C., KANICH, C., GUPTA, M., AND ZIEBART, B. Leveraging Machine Learning to Improve Unwanted Resource Filtering. In *ACM Workshop on Artificial Intelligence and Security* (2014).

[38] BREIMAN, L. Random Forests. In *Machine learning* (2001).

[39] CIMPANU, C. 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.

[40] CURTSINGER, C., LIVSHITS, B., ZORN, B., AND SEIFERT, C. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection. In *USENIX Security Symposium* (2011).

[41] DOLANJSKI, P. Mozilla Firefox The Path to Enhanced Tracking Protection. https://blog.mozilla.org/futurereleases/2018/10/23/the-path-to-enhanced-tracking-protection.

[42] ENGLEHARDT, S., AND NARAYANAN, A. Online Tracking: A 1-million-site Measurement and Analysis. In *ACM Conference on Computer and Communications Security (CCS)* (2016).

[43] GARIMELLA, K., KOSTAKIS, O., AND MATHIOUDAKIS, M. Ad-blocking: A Study on Performance, Privacy and Counter-measures. In *WebSci* (2017).

[44] GERVAIS, A., FILIOS, A., LENDERS, V., AND CAPKUN, S. Quantifying Web Adblocker Privacy. In *ESORICS* (2017).

[45] GUGELMANN, D., HAPPE, M., AGER, B., AND LENDERS, V. An Automated Approach for Complementing Ad Blockers' Blacklists. In *Privacy Enhancing Technologies Symposium (PETS)* (2015).

[46] IKRAM, M., ASGHAR, H. J., KAAFAR, M. A., MAHANTI, A., AND KRISHNAMURTHY, B. Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning . In *Privacy Enhancing Technologies Symposium (PETS)* (2017).

[47] IQBAL, U., SHAFIQ, Z., AND QIAN, Z. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *IMC* (2017).

[48] JOHN ROSS QUINLAN. Induction of decision trees.

[49] KAIZER, A. J., AND GUPTA, M. Towards Automatic identification of JavaScript-oriented Machine-Based Tracking. In *IWSPA* (2016).

[50] KONTAXIS, G., POLYCHRONAKIS, M., KEROMYTIS, A. D., AND MARKATOS, E. P. Privacy-Preserving Social Plugins. In *Usenix Security Symposium* (2012).

[51] LE, H., FALLACE, F., AND BARLET-ROS, P. Towards accurate detection of obfuscated web tracking. In *IEEE International Workshop on Measurement and Networking (M&N)* (2017).

[52] LERNER, A., SIMPSON, A. K., KOHNO, T., AND ROESNER, F. Internet Jones and the Raiders of the Lost Trackers: An Archaeological Study of Web Tracking from 1996 to 2016. In *USENIX Security Symposium* (2016).

[53] LI, B., VADREVU, P., LEE, K. H., AND PERDISCI, R. JSgraph: Enabling Reconstruction of Web Attacks via Efficient Tracking of Live In-Browser JavaScript Executions. In *25th Annual Network and Distributed System Security Symposium* (2018).

[54] MERZDOVNIK, G., HUBER, M., BUHOV, D., NIKIFORAKIS, N., NEUNER, S., SCHMIEDECKER, M., AND WEIPPL, E. Block Me If You Can: A Large-Scale Study of Tracker-Blocking Tools. In *IEEE European Symposium on Security and Privacy* (2017).

[55] NICK NIKIFORAKIS AND WOUTER JOOSEN AND BENJAMIN LIVSHITS. PriVaricator: Deceiving Fingerprinters with Little White Lies. In *WWW* (2015).

[56] PUJOL, E., HOHLFELD, O., AND FELDMANN, A. Annoyed Users: Ads and Ad-Block Usage in the Wild. In *ACM Internet Measurement Conference (IMC)* (2015).

[57] RAMASWAMY, S. Building a better web for everyone. https://www.blog.google/topics/journalism-news/building-better-web-everyone/, 2017.

[58] SHUBA, A., MARKOPOULOU, A., AND SHAFIQ, Z. NoMoAds: Effective and Efficient Cross-App Mobile Ad-Blocking. In *Privacy Enhancing Technologies Symposium (PETS)* (2018).

[59] SNYDER, P., TAYLOR, C., AND KANICH, C. Most websites don't need to vibrate: A cost-benefit approach to improving browser security. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (2017), ACM, pp. 179–194.

[60] STOREY, G., REISMAN, D., MAYER, J., AND NARAYANAN, A. The Future of Ad Blocking: An Analytical Framework and New Techniques. In *arXiv:1705.08568* (2017).

[61] TRAMER, F., DUPRE, P., RUSAK, G., PELLEGRINO, G., AND BONEH, D. Ad-versarial: Defeating Perceptual Ad-Blocking. In *arXiv:1811.03194* (2018).

[62] VASTEL, A., SNYDER, P., AND LIVSHITS, B. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced AdBlocking. In *arXiv:1810.09160* (2018).

[63] WILANDER, J. Apple Safari Intelligent Tracking Prevention. https://webkit.org/blog/8311/intelligent-tracking-prevention-2-0/, 2018.

[64] WU, Q., LIU, Q., ZHANG, Y., LIU, P., AND WEN, G. A Machine Learning Approach for Detecting Third-Party Trackers on the Web. In *ESORICS* (2016).

[65] YU, Z., MACBETH, S., MODI, K., AND PUJOL, J. M. Tracking the Trackers. In *World Wide Web (WWW) Conference* (2016).