An Approach for Identifying JavaScript-loaded Advertisements through Static Program Analysis

Caitlin R. Orr Arun Chauhan Minaxi Gupta Christopher J. Frisz Christopher W. Dunn
School of Informatics and Computing, Indiana University
{crorr, achauhan, minaxi, cjfrisz, chrdunn}@cs.indiana.edu

ABSTRACT

Motivated by reasons related to privacy, obtrusiveness, and security, there is great interest in the prospect of blocking advertisements. Current approaches to this goal involve keeping sets of URL-based regular expressions, which are matched against every URL fetched on a web page. While generally effective, this approach is not scalable and requires constant manual maintenance of the filtering lists. To counter these shortcomings, we present a fundamentally different approach with which we demonstrate that static program analysis on JavaScript source code can be used to identify JavaScript that loads and displays ads. Our use of static analysis lets us flag and block ad-related scripts before runtime, offering security in addition to blocking ads. Preliminary results from a classifier trained on the features we develop achieve 98% accuracy in identifying ad-related scripts.

1. INTRODUCTION

Advertisements (or ads, as they are referred to in common parlance and throughout the rest of the paper) have become a mainstay on the Web. Nonetheless, many users perceive them obtrusive, and instances of ads containing malware are surprisingly common [5]. Further, ad companies often record users' browsing behavior in order to serve targeted ads, which leads to a slew of privacy concerns. While a simple solution to these issues would be to adopt the "Do Not Track" proposal put forward by the World Wide Web Consortium's (W3C), which allows a browser to signal an ad provider that the user wishes not to be tracked, its adoption is being actively debated among users' advocates and ad providers due to the $70-billion-a-year revenue that online ads bring to their providers [22]. In the meantime, users are resorting to ad blocking tools at their end, such as Adblock Plus [2] and Ghostery [12]. These tools identify ads by matching every requested URL on a web page against large lists of regular expression filters. While usually effective, this general approach has multiple shortcomings. First, as new cases of unblocked ads appear and are discovered, their corresponding regular expressions need to be added to the filtering lists. An inspection of the change logs of the most popular filtering list, EasyList, with over 12 million subscribers [1], indicates that between five and fifteen filters are added a few times a week [3]. Given that EasyList currently contains almost 18,000 regular expressions, this trend does not bode well for the number of filters that each fetched URL needs to be checked against, especially because obsolete filters are removed rather infrequently. Another important shortcoming is that filters are known to occasionally break website functionality, as was noted in the work by Leon et al. while evaluating the usability aspects of tools similar to Adblock Plus [17]. This might happen because of overly aggressive filtering, since the granularity of blocking is too coarse (URL). Finally, while Ghostery relies exclusively on URL-based regular expressions, Adblock Plus also makes use of HTML element-based regular expressions. The latter prevents the rendering of ad-related portions of web pages in cases where matches occur, but not the actual loading of ads, thereby failing to offer any security or save bandwidth.

A large fraction of ads on modern websites are loaded into the browser through JavaScript. In fact, almost all malware utilizes JavaScript [24]. Thus, the most effective way to ensure safety and keep JavaScript ads from loading is to disable JavaScript in the browser. However, encouraged by the standardization of JavaScript [11] and resulting from a proliferation of dynamic content on highly interactive web pages, websites are relying increasingly on JavaScript. Consequently, disabling JavaScript interferes with the basic features of the content on a large number of websites, which usually makes it an impractical option.

Fortunately, the widespread use of JavaScript in ads offers an opportunity to devise a solution for blocking them that centers around JavaScript itself. Specifically, in this paper we present an approach to identify ads loaded via JavaScript. Our approach uses features derived from static program analysis of JavaScript code belonging to ad-related versus other functionality of web pages. This static analysis-based approach lets us flag and block the offending scripts before they ever get a chance to run, offering security in addition to blocking. In contrast to past techniques developed in the context of privacy concerns that involved injecting code into a script before running it to track information flow [15], our static analysis approach can flag offending scripts before running them, eliminating substantial overheads. Our approach provides multiple advantages compared to past methods of detecting ads. First, relying

on quantitative measures, instead of manually-maintained lists, makes our approach adaptive, improving its robustness in the face of changing domain names or content. Second, rooted in machine learning, our approach is amenable to improvement over time without having to rely on an externally updated list.

In this preliminary work, we make the following advances toward our goal. First, we develop a set of 20 features based on static JavaScript analysis that can effectively discriminate ad-related scripts from non ad-related scripts. Second, we develop a supervised machine learning-based classifier that uses these features to identify ad-related scripts. Testing our classifier in the wild shows that it can identify ad-related scripts with an accuracy of almost 98%. While much work remains before our approach can become a practical competitor to existing ad blocking tools, the preliminary results strongly indicate that the approach is promising.

2. OUR APPROACH

We use the word *script* in this paper to refer to a piece of JavaScript. A script always corresponds to the code from a URL that points to a JavaScript file on the server, or a piece of JavaScript code embedded within the HTML `script` tag. This is the granularity at which our analysis classifies ad-related JavaScript code. Later, we discuss the issue of refining the granularity in Section 5.

The basis of our approach is the detection of idiomatic consistencies, or features, that appear primarily in the source code for JavaScript ads. In order to identify these features, we first needed to be able to manually distinguish between ad-related scripts and other scripts on web pages. Toward that goal, we used Firebug, a popular web development plugin for Firefox [18], which shows the DOM of a web page in a separate window. Firebug allows the user to highlight portions of the web page when corresponding elements in the DOM are hovered over. Figure 1 shows a screenshot in which hovering the mouse over a DOM element highlights the corresponding portion in the rendered image. In this case the highlighted portion shows the ad in the top right part of the window. The DOM element that causes only the ad to be highlighted provides the subtree that contains all ad-related content, including any JavaScript.

We used this method to extract the DOM subtree of ads on 100 commonly visited web pages and identified all the scripts that were included in that subtree. These scripts were considered ad-related. However, this technique is not sufficient to identify all ad-related scripts. Often, the script that starts loading an ad is not located within the DOM subtree of the ad itself. Therefore, to isolate ad-related scripts in the rest of the page those scripts were dumped in well-formatted source form and inspected manually. We observed that most web pages had one or more external ads and most of these ads were at least partially loaded via JavaScript. We also made the observation that non ad-related scripts outnumbered ad-related ones in most cases.

We came up with ideas for features based on intuition and experience, and then manually studied both ad-related scripts and other scripts to determine whether these features were in fact notably distinguishing. Since JavaScript code is often cosmetically obfuscated, we instrumented Firefox's JavaScript engine, SpiderMonkey [14], to dump the Abstract Syntax Tree (AST) for each JavaScript file so we could examine the code's abstract representation. In order

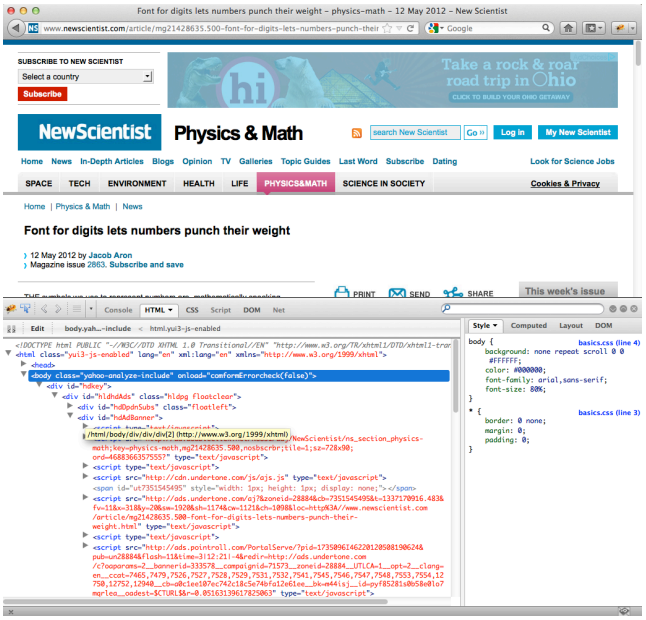

Figure 1: Screenshot of Firebug being used to identify the DOM-subtree corresponding to an ad.

to process the ASTs we used our homegrown Ruby library, called RubyWrite [16]. RubyWrite is a domain-specific language (DSL) embedded in Ruby, used for analyzing, transforming, and unparsing ASTs. We used RubyWrite to analyze the scripts for multiple features, described below. We also used RubyWrite to implement an AST unparser, to get well-formatted scripts for manual inspection, which complemented the observations directly based on ASTs, as explained before.

While many of the features we tried were highly indicative of ad-related scripts from others, there were a few that turned out not to be. An example of a feature that was not discriminating was the size of the JavaScript file, since we found both ad-related and other scripts to cover a wide range of sizes. The features that discriminated ad-related and other scrips fall in the following seven categories (see Table 1). Note that the last feature only counts the number of features that got flagged for a particular JavaScript file, so it does not belong to any of the seven categories. The appendix lists the details of some of the features that we used.

2.1 Code Obfuscation

Conventional wisdom suggests that code obfuscation techniques would be used more widely by JavaScript programmers who wish to hide from a casual reader what a program does. Consequently, these techniques could be expected to be used more frequently in ad-related scripts than in other scripts. We started out by looking for simple code obfuscation mechanisms, such as the absence of whitespace and non-descriptive variable names. We additionally examined the call graphs of JavaScript files as well as individual functions in those files when they were present, with the hypothesis that added layers of function call indirection might also be a commonly used obfuscation technique. While we found ad-related scripts to definitely be using all of these code ob-

```
y.write("<script onreadystatechange='if (this.readyState==\"complete\") {"
      + "this.parentNode.removeChild(this);_domcontentready();}'"
      + "defer='defer' src='" + ca + "'><\/script>");
```

(a) HTML generation

```
Image.src = "http://ad.afy11.net/ad?mode=7" + "&publisher_dsp_id=2&external_user_id=" + uid;
```

(b) URL generation

Figure 2: Examples of constant string folding in ad-related scripts.

fuscation techniques, we found that JavaScript library writers and web developers were also making extensive use of these techniques. Often, all whitespace is removed and single character variable names are used in order to save on bandwidth, since many libraries are loaded a large number of times. Further, it is becoming increasingly common for JavaScript programs to be automatically generated by tools written in other programming languages, and depending on the code generator, the output JavaScript code may be highly unreadable, appearing to be deliberately obfuscated.

However, there is one type of obfuscation that ad-related scripts did reliably perform more often than other types of scripts: ad-related scripts often use a series of string concatenations to construct URLs, HTML, or JavaScript functions—the last often passed to `eval()`. Doing so not only makes it harder to trace where ads are coming from but also enables ad providers to dynamically choose ads for display. The latter is useful because, frequently, the ad to be displayed is chosen from large databases and a random number (as a simple example) is dynamically inserted into a URL to compose the ad URL. Feinstein and Peck [13] also found string manipulation techniques being used disproportionately in ad-related JavaScript code.

We use the AST produced by SpiderMonkey and apply a standard compiler optimization technique, *constant propagation and folding*, to concatenate string snippets. This feature is referred to as *folded strings* in Table 1, which summarizes the features used. Examples of two instances where constant folding can be applied are shown in Figure 2. In the first, string concatenation is used to generate HTML which registers an event and then executes some JavaScript code, which happens to be empty in this particular case. In the second, a URL is dynamically composed from two constant strings and an identifier.

2.2 Dynamic Code and URL Generation

Performing constant folding also allowed us to more easily identify other features relating to strings, such as the presence of JavaScript code, HTML, and URLs in strings. The presence of these indicates that ad-related scripts are dynamically generating JavaScript code, HTML, and URLs. Each case has obvious uses: a URL string may be passed to, for example, the JavaScript `get()` function, which executes an HTTP GET request, an HTML string might be given as an argument to any of several DOM modifying functions, and a script string may be passed to `eval()`. These are simple ways to extend a script, inject more code into the DOM, or even obfuscate a script's purpose by adding layers of indirection to its execution.

Of course, it is not always possible to perform constant folding on these strings. In the most common case, ad-related scripts will generate other scripts, HTML, or URLs conditionally. For example, a portion of a URL may be generated so that it contains the user's screen resolution, as a way of transmitting user information. In the case of HTML generation, string content ranged from snippets to entire HTML documents. In fact, in many cases, we found the strings resulting from this type of dynamic code generation were abnormally long, which we were also able to use as an indicator. We found that a threshold of 200 characters acted as a good separator between the strings likely to contain script or HTML information and other strings. The four features resulting from this category are numbered 2-5 in Table 1.

2.3 Code Structure

Work by Curtsinger et al. [9] and Rieck et al. [21] investigated JavaScript code structure in the context of malicious scripts. Here, *code structure* refers to specific programming constructs or certain syntactic idioms, such as, the ternary "`?  :`" operator for a conditional expression. Inspired by these works, we examined the structure of ad-related and other scripts to find distinctive program constructs and syntactic idioms. For example, `try-catch` constructs are more popular in ad-related scripts, perhaps to ensure that ad-related code does not break the functionality of a website. Similarly, bitwise XOR operation and the use of shift operator was more common in ad-related scripts, perhaps as a means to obfuscate. On the other hand, arrays were more common in non ad-related scripts. So was the use of `throw`, which was often used to ensure the correct operation of a website. While these program constructs were helpful distinguishers, there were several we initially thought might be helpful that were not. An example of a program construct that did not help distinguish ad-related scripts from other scripts was the comma operator, which was popular in both types of scripts.

Table 1 lists the two features corresponding to program constructs used to distinguish ad-related and other scripts. The list of program constructs appears in the appendix.

We also found many distinguishing syntactic idioms in both ad-related and other scripts. For example, a `key:value` pair where `key` was a string was commonly found in non ad-related scripts, often in the context of supporting pull down menus. Similarly, `return [expression,...];` was more common in ad-related scripts. What this does is evaluate each listed expression, but only return the value of the last one. This idea is more intuitively expressed as an imperative list, rather than embedding the expressions inside the return statement. Ultimately, we decided upon 14 syntactic idioms in ad-related scripts and 19 in scripts that were not ad-related. These features are numbered 8-9 in Table 1. The appendix details the syntactic idioms we used.

Category	#	*Feature*	*Description*
Code obfuscation	1	folded strings	counts string concatenations
Dynamic code and URL generation	2	JavaScript in strings	counts the presence of JavaScript in strings
	3	HTML in strings	counts the presence of HTML in strings
	4	URLs in strings	counts the presence of URLs in strings
	5	long strings	counts the number of strings longer than 200 characters
Code structure	6	program constructs in ad scripts	counts the presence of three program constructs
	7	program constructs in other scripts	counts the presence of two different program constructs
	8	syntactic idioms in ad scripts	counts the presence of 14 syntactic idioms
	9	syntactic idioms in other scripts	counts the presence of 19 different syntactic idioms
Function call distribution	10	instances of `eval()`	counts instances of `eval()`
	11	function calls in ad scripts	counts the occurrence of 19 built-in JavaScript functions
	12	function calls in other scripts	counts the occurrence of 14 different built-in functions
Event handling	13	event handlers in ad scripts	counts the occurrence of 8 JavaScript event handlers
	14	event handlers in other scripts	counts the occurrence of 19 different event handlers
Script origin	15	script is loaded from third party	flags a script as third party or not
Presence of keywords	16	keywords in URLs	counts the corresponding regular expression matches
	17	keywords in variable names	counts the corresponding regular expression matches
	18	keywords in function names	counts the corresponding regular expression matches
	19	keywords in strings	counts the corresponding regular expression matches
	20	features flagged	counts the number of features flagged

Table 1: Features distinguishing ad-related scripts from other scripts.

2.4 Function Call Distribution

A high number of calls to specific functions could indicate certain behavioral patterns. For instance, functions that manipulate the DOM, such as `document.write`, `document.referrer`, and `document.writeln` might be more likely in ad-related code. Similarly, calls to `eval()` on strings mean synthesizing and executing new code dynamically, which is a more likely behavior for ad-related scripts. In fact, the `eval()` function is one of the greatest reasons for JavaScript's notorious problems with security [19, 20, 23, 24], because it allows for the injection of foreign code.

Based on this intuition, we identified 19 built-in JavaScript functions that were more commonly observed in ad-related scripts. We also identified 14 built-in JavaScript functions that were more commonly in other types of scripts. (These functions are enumerated in the appendix.) We treated `eval()` differently due to its security implications. In total, we designed three features around these observations, which are shown in Table 1. Note that even though static analysis of JavaScript cannot accurately determine the runtime frequency of calls to certain functions, it is possible to estimate relative frequencies relatively accurately. For instance, a function called inside a loop is likely to be more often called than one outside of a loop. The analysis can also account for nested calls. For example, if function f_1 calls f_2, where f_1 is called at many places in the program, but f_2 is only called once, within f_1, we would still conclude that f_2 is called as often as f_1. However, in the interest of simplicity in this preliminary work, we only performed flow-insensitive analysis, i.e., we counted the occurrences of each of the functions in the JavaScript files without concern for their relative call frequencies. Note that a flow-sensitive analysis could increase the accuracy of the analysis, for example, by determining that certain branches are not reachable and not counting functions called within those branches.

2.5 Event Handling

We observed that it is common for ad-related JavaScript code to frequently register certain kinds of events. One example is the *onload* event, which several types of ad-related scripts favor because it helps to ensure that a portion of the page has been fully loaded before beginning, for example, an audio track or animation. On the other hand, non ad-related JavaScript code also try to register several kinds of events with distinguishing regularity. For example, registering *onclick* allows handling events such as when a user clicks on a button or selects from a pull down menu, submits a form, or generally interacts with a page's interface. Clicking on an ad, however, usually just redirects the user to a new website, an action that does not require an event handler.

Overall, we found 8 event handler names to be more commonly used in ad-related scripts and 19 more to be more commonly used in other types of scripts (listed in the appendix). The two features that count the occurrences of these event handlers are listed in Table 1 as features 13-14.

2.6 Script Origin

Modern web pages are complex, with portions of many web pages loaded from external servers. Thus, even though JavaScript runs in a single environment within the browser for each page, a complete JavaScript program is often built out of multiple URLs and multiple inlined code (within the `script` HTML tag). However, programs that load and display ads and collect user data use standard third-party sites to load portions of their functionalities much more commonly than code that is unrelated to ads. This stems from the common use of libraries provided by analytics services and advertising agencies. We designed a boolean feature to capture this observation. The feature sets a flag if a JavaScript is loaded from a third-party server (see Table 1).

2.7 Presence of Generalized Keywords

This category of features most closely resembles the methods that Adblock and other existing ad blocking tools employ. The major difference is that existing tools work by

matching against URLs and HTML elements only, while our approach catches keywords that exist anywhere in a JavaScript. Specifically, we look for a small number of keywords related to ads through generalized regular expressions designed to spot them in URLs, function names, variable names and strings. These regular expressions are mentioned in the appendix. The four features in this category are shown in Table 1.

3. EXPERIMENTAL RESULTS

We tested our approach in two steps. In the first step, we trained and tested a classifier on a mix of ad-related and non ad-related scripts using the features described in Section 2. The scripts were derived from both popular and relatively unpopular websites from all Alexa [4] categories and a range of different countries, in order to capture website diversity. In the second step, we ran the classifier on a new, untested set of websites derived from Alexa.

3.1 Experimental Setup

Alexa organizes websites in 16 categories. From each category, we picked between 14 and 20 websites, avoiding any duplicates resulting from websites showing up in multiple categories and ensuring that websites from top, middle and bottom of each category were picked. This process resulted in 254 diverse websites in the 16 categories combined, as shown in Table 2. To this group, we added websites from Alexa's international sites directory and selected a set of web pages in countries outside of the United States. We picked 24 countries from all continents, except Antarctica, and added between four and six web pages from each. The process resulted in 85 additional websites shown in Table 3. We trained our classifier on the top-level pages of this group of 339 websites, which was different from the 100 web pages we used to identify distinguishing features. As in the case of feature identification, we visited each website and used the Firebug extension to identify ad-related and other scripts. We also used the combination of SpiderMonkey's AST, RubyWrite and the unparser described in Section 2, to derive values of each of the features. Note that only the step involving Firebug needs to be done while visiting websites. The steps involving SpiderMonkey's output can be done offline if the scripts corresponding to each website are saved for analysis.

The 339 web pages in our data set contained many repeated scripts. Sometimes, the same JavaScript file would be loaded multiple times on a single website. Often, scripts repeated across websites as well, such as when Google's ads are loaded on multiple websites through the `show_ads.js` JavaScript, or when common libraries, such as `jQuery`, are loaded. In order to ensure that a variety of different types of scripts are represented without biasing the classifier in favor of popular scripts, we picked 250 unique ad-related and 250 non ad-related scripts to train the classifier.

Category	*Web pages*	*Category*	*Web pages*
Adult	14	News	17
Art	18	Recreation	15
Business	14	Reference	16
Computers	16	Regional	16
Games	20	Science	17
Health	14	Shopping	13
Home	17	Society	18
Kids and Teens	15	Sports	14
Total			254

Table 2: Web pages from various Alexa categories.

Country	*Web pages*	*Country*	*Web pages*
Afghanistan	5	Malaysia	4
Azerbaijan	5	Mexico	4
Bahrain	6	Pakistan	4
Brazil	4	Romania	5
Chile	4	Russia	4
China	6	South Africa	4
Egypt	4	Sweden	4
France	4	Thailand	4
India	4	Turkey	4
Indonesia	4	UK	5
Italy	4	Vietnam	4
Japan	5	Yemen	4
Total			85

Table 3: International web pages from Alexa.

For the classifier, we chose the supervised machine learning-based Support Vector Machine (SVM) classifier in its default configuration from the LIBSVM [7] package. We supplied the classifier with normalized values for each of the features described in Section 2.

Most of the features were normalized using a linear function on the script's length, where the coefficient is fit via the testing set. What is actually meant by length depended on the feature. Features that dealt with function call distribution used the total number of function calls in the script; features that dealt with strings used the total number of strings in the script; other features used the literal length of the script, defined here for consistency as the number of nodes in the AST representation. A simpler definition of length, such as counting bytes, can generate skewed results when applied to programs that contain long strings or variable names. This method was used for 16 of the 20 features.

The remaining 4 features are the four code structure features (6-9 in Table 1). For each of these, we wished to take into account the relative rarity of each programming construct or syntactic idiom we were flagging. To do this, we utilized a well-known normalization technique called tf*idf (term frequency-inverse document frequency). This is a technique that is standard in the field of natural language processing, usually for document classification. It works by offseting the number of times a feature appears by the frequency of that feature in a supplied data set (in this case, the training set of web pages).

The *accuracy* of the classifier is defined as:

$$\frac{(\text{true positives} + \text{true negatives})}{(\text{true positives} + \text{true negatives} + \text{false positives} + \text{false negatives})}$$

We performed a five-fold cross validation, where we trained the classifier with 80% of the scripts of each kind and tested on the remaining 20% until each of the five possibilities were exhausted. Following this, the classifier was ready for testing.

In the next step, we took the classifier and ran it against 25 new randomly chosen websites from the top 100,000 Alexa

	Website	*Ad-related scripts*	*Other scripts*	*True positives*	*True negatives*	*False positives*	*False negatives*
1	`9gag.com`	26	30	24	29	1	2
2	`adultfriendfinder.com`	7	50	7	50	0	0
3	`aintitcool.com`	20	21	19	21	1	0
4	`aizhan.com`	0	11	0	11	0	0
5	`booking.com`	3	22	3	21	1	0
6	`chinanews.com`	3	23	3	23	0	0
7	`constantcontact.com`	8	66	3	66	0	5
8	`fandom.com`	55	35	54	34	1	1
9	`fedex.com`	0	16	0	16	0	0
10	`gamesfreek.com`	12	34	12	33	1	0
11	`godaddy.com`	2	39	2	38	1	0
12	`ikea.com/us/en`	0	20	0	20	0	0
13	`mop.com`	0	51	0	50	1	0
14	`onet.pl`	32	99	31	99	0	1
15	`tinterest.com`	0	8	0	8	0	0
16	`pof.com`	5	6	5	6	0	0
17	`salesforce.com`	12	39	12	37	2	0
18	`soundcloud.com`	2	20	2	20	0	0
19	`surveymonkey.com`	4	11	4	11	0	0
20	`target.com`	11	40	11	39	1	0
21	`taringa.net`	28	26	28	25	1	0
22	`torrentz.com`	0	5	0	5	0	0
23	`vimeo.com`	3	12	3	12	0	0
24	`youjizz.com`	16	18	16	18	0	0
25	`youpornAAAAA.com`	9	24	8	23	1	1
Total		258	726	246	716	12	10

Table 4: Performance of the classifier on top-level pages of 25 randomly chosen websites.

websites, ensuring that neither the 100 websites we used to learn the features, nor the 339 websites we trained and tested the classifier on were included. As we visited the top-level web page of each website, we used Firebug as before to create ground truth for each script to determine whether it was ad-related or not. The important difference in this step compared to when we experimented with the features or trained the classifier was that for each script we dumped its AST and unparsed it using RubyWrite to retrieve a pretty-printed script in a single online step. Doing so allowed us to study the features flagged for each script precisely while sidestepping any issues with the dynamic nature of modern web pages, which often change content on reloads.

3.2 Results

Table 4 shows the result of analyzing 984 scripts found on each of the 25 test web pages. Details of true positives, true negatives, false positives and false negatives for each page are also shown. The overall accuracy of 97.76% is a little higher than what we obtained while testing and training the classifier. The reason is that our training and test set was built with a variety of script examples, without repetitions. On the other hand, randomly selected websites tend to use certain JavaScript libraries more frequently. Once our classifier bins those libraries correctly, its overall accuracy improves for all the scripts that use those libraries.

Of the 12 false positives, 7 were the same Google API script, `cb=gapi.loaded_0.js`. This script deals with managing cookies and generating HTML. As a result, it looks suspiciously similar to ad-related code. One possible way to eliminate such false positives is to include the context in which the API is used and try to classify it together with the context, such as the code from where the API methods are invoked. Another option is to classify individual functions within a script, instead of trying to classify an entire script. Since a library might provide varied functions, with a mix of functions that may or may not provide features relevant to ad-related code, it would be useful to analyze functions within a script individually. For example, we may wish to take certain actions, if desired, only on ad-related functions without impacting other features supported by a library.

Although the results show 10 false negatives, these actually amounted to only 4 unique scripts. The five false negatives at `constantcontact.com` were all the same script (all analytics scripts originating from `btstatic.com`), loaded five separate times. There were an additional two other pairs of identical scripts, and one singleton script. All the false negatives were attributable to JavaScript code that was specific to analytics, which is code to track users. Such code often accompanies ads but sometimes does appear by itself. Although our goal is to also be able to identify analytics-related code, and we are therefore counting misidentified scripts among the false negatives, the focus of our current work is on ads. We plan to develop features for analytics-related JavaScript in future work, as discussed in Section 5.

Overall, the classifier only produced 6 unique false positives and 4 unique false negatives.

3.2.1 *Effectiveness of features*

Next, we went on to determine the effectiveness of various features we used in the classification process. Toward this goal, we analyzed each of the 984 scripts we extracted from

Feature	*Feature number*	*% Ad-related scripts*	*% Other scripts*
Folded strings	1	30.4	13.3
JavaScript in strings	2	12.6	1.7
HTML in strings	3	48.7	20.4
URLs in strings	4	47.0	23.8
Long strings	5	30.1	6.8
Program constructs in ad scripts	6	23.9	13.6
Program constructs in other scripts	7	23.9	40.2
Syntactic idioms in ad scripts	8	23.5	13.6
Syntactic idioms in other scripts	9	19.6	44.0
Instances of `eval()`	10	17.4	9.4
Function calls in ad scripts	11	63.9	37.9
Function calls in other scripts	12	20.0	38.6
Event handlers in ad scripts	13	22.2	12.2
Event handlers in other scripts	14	13.4	21.0
Script is loaded from third party	15	76.1	41.5
Keywords in URLs	16	67.0	0.0
Keywords in function names	18	43.5	5.2
Keywords in variables/strings	17, 19	74.3	5.4
Features flagged[1]	20	*N/A*	*N/A*
Total scripts		*258*	*726*

Table 5: Percentage of ad-related and other scripts flagged for each feature. Feature numbers refer to numbers in Table 1. Grey rows indicate features that are expected to occur more frequently in non ad-related scripts.

the 25 websites used for testing. Specifically, we looked for which features got flagged, and at what frequency for what kinds of scripts. Table 5 shows the percentage of each kind of script that got flagged for each of the features listed in Table 1.

A few things are noteworthy in Table 5. First, all features are discriminating in the manner expected. In particular, features 7, 9, 12 and 14 are found more commonly in non ad-related scripts than ad-related scripts, as expected. The same is true of the rest of the features, which are more commonly found in ad-related scripts. All features serve to discriminate between ad-related scripts and other scripts, but some are more important than others. The keyword-related features (15-19) are among the most discriminating, as well as scripts being loaded from third-party websites (feature 15).

4. RELATED WORK

There have been several attempts to use feature-based analysis techniques for the identification of JavaScript malware on the Internet. We highlight a representative sample first and then outline the differences involved in applying similar techniques to ads.

Cova et al. [8] developed JSAND, a tool that uses instrumented versions of HTMLUNIT, a Java-based browser simulator, and Rhino, a JavaScript interpreter developed by Mozilla, to identify JavaScript malware using a set of ten static and dynamic features. These features all represent characteristics of drive-by-download attacks. JSAND also has a publically available web interface.

Prophiler [6] is a static analysis-based system presented by Canali et al. for the identification of malicious web pages, which builds upon the idea of feature-based identification, and utilizes a combination of JavaScript, HTML, and URL features. They decrease runtime by quickly identifying benign web pages, so that more time can be spent on malicious pages.

Cujo [21] is a system described by Rieck et al. that uses both static and dynamic analyses in combination with an SVM-based classifier framework for the detection of drive-by-download attacks. After parsing the JavaScript source code into tokens, Cujo is able to automatically extract groups of these tokens (called Q-grams) as features that are indicative of malicious code. Cujo has a high rate of successful identification, and achieves significantly better runtime performance than to JSAND.

Curtsinger et al. [9] describe ZOZZLE, a JavaScript malware detector that is meant to be used in the browser, and is therefore optimized for speed. It uses Bayesian classification and statically generated syntax features to achieve a significantly lower false positive rate than previous works. Training was done on both a hand-picked set of a features, which were all specific to malicious code, and an automatically selected set of features, which also included features specific to benign code. The analysis is "mostly static", but has some dynamic aspects for deobfuscation.

Caffeine Monkey [13] is a tool based on extensions to SpiderMonkey that is designed for the analysis of obfuscated malicious JavaScript code. The authors discuss the use of function call distribution for the identification of malicious JavaScript; this is one of the features we use to identify ads.

AdSentry [10] is a framework for the detection and isolation of malicious JavaScript ads, which is based on SpiderMonkey. However, it makes no attempt to identify or

[1]The presence of this feature (20) indicates that one or more ad-identifying features were flagged. The generalized nature of the features means that even scripts entirely unrelated to ads typically exhibit at least one feature. Therefore, a feature which counts the number of other features present relies more than any of the others on intensity, rather than the simple fact of existence, making percentages meaningless in the context of this table.

classify malicious ads, and instead provides a shadow engine for code execution.

Identifying ads is trickier than identifying malware for several reasons:

1. Ads cover a wider range of behaviors. As a result, it is much more difficult to create a concrete definition of ad than malware. Many past efforts on detecting malware have focused on specific types of attacks, such as drive-by-download attacks [6, 8, 21], which allows a narrow, but well-defined, characterization of malware.

2. Some efforts, such as [6], have attempted to classify malware at web page granularity. This allows them to consider features that would not make sense at file granularity, such as the distribution of HTML tags on a page. This sort of classification would not work for ads, because they appear virtually exclusively as embedded elements in a page.

3. There already exist several data sets of known JavaScript malware files that can be used for testing and training purposes, which are used by several of the related works [8, 9, 21]. Nothing similar exists for ads.

5. DISCUSSION

Our study indicates that it is possible to come up with a set of criteria that distinguish ad-related scripts from other webpage content with a high degree of accuracy. This functionality can be used as a basic building block for automatically blocking ads but several issues remain. We are in the process of examining these issues, which we describe next.

Determining Ads.

Humans rely on contextual information and several other subtle clues to determine what is an ad on a webpage. Nevertheless, deciding that a part of a page is an ad can still sometimes be confusing at first. Unless an ad is marked with keywords, the problem of determining whether a portion of content is ad is inherently difficult to solve algorithmically due to the need to rely on contextual and semantic information to arrive at the decision. It is important to emphasize that while we have attempted to solve the problem of identifying ad-related JavaScript, we have not tried to address the problem of determining ads in this paper.

Script Hierarchies.

Our current approach focuses on flagging scripts related to ads. Using a flagged script and inferring if it would block an ad and which one in the case where a webpage contains multiple ads turns out to be a difficult problem. Among the key issues complicating the matter is that modern webpages often contain hierarchies of scrips. In particular, it is possible for a script to load other scripts that, in turn, could load more scripts, and so on. At each step a script could modify the DOM in addition to loading more scripts or HTML objects. This leads to a natural hierarchy of scripts, representing the scripts as a tree[2], where an edge indicates that the parent script loaded the child script. Note that if script S_1 loads script S_2, which loads S_3, and S_3 loads an ad, then successfully identifying any of the three scripts is sufficient to block the ad, if desired. On the other hand, preempting an upstream script might have an adverse impact on the functionality of a webpage. Therefore, in order to perform a systematic evaluation of the potential impact of our classifier, it is critical to know the relationship among scripts. Further, knowing if blocking a script would lead to blocking an ad requires knowing which parts of the DOM tree are impacted by each script. The latter requires identifying ads on webpages and the portions of the DOM tree corresponding to each ad.

As mentioned above, determining what is an ad is a difficult problem to solve. It turns out that even identifying the DOM tree corresponding to an ad is a non-trivial problem.

Identifying Ad-related DOM Subtree.

Once we have determined what the ads are on a page, the next step is to build the script hierarchy and determine the relationship between scripts and the portions of the DOM tree they impact. This relationship is complicated by the presence of event handlers that are triggered on loading of certain scripts. Consider the idea of loading scripts requested from a page, one at a time, and examining the changes to the DOM tree after each load to identify the portions of the DOM impacted by each script. However, some scripts might install event handlers that may load more scripts, trigger already loaded scripts, modify DOM trees, or do all of these in unpredictable ways. Indeed, it turns out to be common for scripts to run only after certain other scripts have been loaded to satisfy dependencies. As a result, merely examining the scripts being loaded and tracking changes to the DOM tree is not sufficient. The asynchronous and concurrent loading of URLs further complicates the problem. We are currently building a Firefox extension that can automatically build a relationship graph for scripts on a webpage and also identify the portions of DOM tree affected by each script by carefully monitoring the events and examining their impact. However, the use of that information is complementary to the approach presented in this paper.

Granularity of Classification.

The most obvious application of classifying scripts as ad-related is to block the execution of those scripts in order to block the ads. The idea of blocking ads leads to the question of the granularity at which the blocking should occur. If an entire script is blocked then it would also block any desirable aspects of that script which may be present. Therefore, it seems worthwhile to classify not just entire scripts, but various functions inside a script as ad-related. This is especially useful in the context of large libraries which are often loaded by scripts. We have not explored blocking scripts in this work but the fundamental techniques we have presented will continue to be applicable whether the blocking granularity is partial or total.

Comparison with Ad-blocking Techniques.

A natural thing to look at after identifying ad-related scripts is how effectively we can use that information to actually block ads, as compared to existing brute-force pattern matching-based approaches used by, for example, Adblock

[2]Technically, it could be a directed acyclic graph (DAG) if a script is loaded multiple times, or even a cyclic graph if a script attempts to load one of its ancestors. Even though multiple loads, leading to a DAG, are not uncommon, cycles are rare.

Plus. It is not a simple matter of loading a page with Adblock Plus enabled and then comparing it by reloading it with some kind of JavaScript filter based on our classifier because invariably, ads change with every reload of the page, even changing the scripts that get loaded. Therefore, we would need a technique that can determine the effectiveness of Adblock Plus as well as our analysis-based technique in just one load of a webpage.

We are currently in the process of developing such a technique by implementing our own regular expression matchers that mimic the behavior of Adblock Plus and Ghostery. Combined with the knowledge of which portions of a DOM tree are impacted by different scripts, we can compare the success rates of the two approaches at blocking ads in a single load of a webpage. In this paper, we have focused on developing the approach for identifying the pertinent scripts, rather than using that information to block ads. As discussed before, the granularity of classification and the data collection methodology has to be carefully tuned to make such a comparison possible and scientifically sound.

Addressing Analytics.

Advertisers often track Web users in order to serve targeted ads. Consequently, it is common for scripts related to analytics to accompany ads. We have implied an inclusive meaning of the term "ad-related" in this paper to also admit scripts that perform analytics functions, as well as those which display ads. Indeed, all of our false negatives are analytics scripts. However, it should be possible to identify more features, or even feature categories, by paying special attention to the behavior of analytics scripts. It may also be desirable in certain circumstances to identify only analytics scripts, rather than all ad-related scripts. The classifier and the feature set will need to be tuned to make that possible.

Other Multimedia Ads.

We have not made an attempt at analyzing ads that are delivered through mechanisms other than JavaScript, such as Flash or other video formats. We believe that the approach we have developed for JavaScript can be extended to include other major ad delivery formats.

Evasion.

Our approach accounts for ad providers keeping the behavior of ad-related scripts the same but altering the URLs that load ads. The latter will cause regular expressions of the type used by Adblock Plus and Ghostery tools to fail until they are updated. However, if ad providers significantly alter the functionality of their scripts for evasion purposes or otherwise, our classifier would have to be retrained. While this situation is less than ideal, we note that it is similar in nature to the issues faced by classifiers used for identifying email or Web spam, or even malware.

6. CONCLUSION

We have presented a static analysis-based approach to the identification of ad-related JavaScript scripts. Unlike the existing regular expressions-based filters, our technique is highly scalable and adaptive, and unlike techniques that rely on information only available at runtime, our purely static approach enables us to identify offending scripts before they have a chance to run. We carefully identified 20 distinct features in 7 different categories that could be used to distinguish ad-related scripts from others. These features are extracted from a given script by hooking into SpiderMonkey to parse the script, and then statically analyzing the AST using our homegrown tool, called RubyWrite, for analyzing and manipulating ASTs. We selected 250 unique scripts that were loaded from a total of 339 websites and used those as the training set for an SVM-based classifier based on these features. We then tested our classifier on scripts from 25 randomly chosen websites from Alexa Top 100,000 and obtained 98% accuracy in correctly classifying the scripts.

This paper has shown that a systematic, behavior-based approach to detecting ad-related scripts is feasible and could be used as a robust mechanism to filter ads. While we selected the features manually and empirically for this study, the selection process itself could be automated, which would improve our approach. We believe that the concept can be extended to other ad-delivery mechanisms, such as Flash videos.

7. ACKNOWLEDGEMENTS

This material is based in part upon work supported by the U.S. Department of Homeland Security under Grant Award Number 2006-CS-001-000001, under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. Department of Homeland Security, the I3P, or Dartmouth College.

This material is also based upon work supported by the National Science Foundation under Grant Numbers CNS-1018617 and CNS-0834722. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] EasyBlog - EasyList statistics: August 2011. `https://easylist.adblockplus.org/blog/2011/09/01/easylist-statistics:-august-2011`, 2011.

[2] Adblock Plus - for annoyance-free web surfing. `http://adblockplus.org/en/`, 2012.

[3] EasyList Mercurial changelogs. `https://hg.adblockplus.org/easylist/`, April 2012.

[4] Alexa Internet. The top 500 sites on the Web. `http://www.alexa.com/topsites`, May 2012.

[5] Dasient Blog. Q1'10 web-based malware data and trends. `http://blog.dasient.com/2010/05/q110-web-based-malware-data-and-trends.html`, May 2012.

[6] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *International World Wide Web Conference (WWW)*, 2011.

[7] Chih-Chung Chang and Chih-Jen Lin. LIBSVM – A Library for Support Vector Machines. `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`, 2012.

[8] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download

attacks and malicious JavaScript code. In *International World Wide Web Conference (WWW)*, 2010.

[9] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. ZOZZLE: fast and precise in-browser JavaScript malware detection. In *USENIX Security Symposium*, 2011.

[10] Xinshu Dong, Minh Tran, Zhenkai Liang, and Xuxian Jiang. AdSentry: comprehensive and flexible confinement of JavaScript-based advertisements. In *Annual Computer Security Applications Conference (ACSAC)*, 2011.

[11] Standard ECMA-262. ECMAScript language specification, edition 5.1. Technical Report ISO/IEC 16262:2011, ECMA International, June 2011.

[12] Evidon, Inc. Ghostery. `http://www.ghostery.com/`, 2012.

[13] Ben Feinstein and Daniel Peck. Caffeine Monkey: Automated collection, detection and analysis of malicious JavaScript. In *Black Hat USA*, 2007.

[14] Mozilla Foundation. SpiderMonkey (JavaScript-C) Engine. `http://www.mozilla.org/js/spidermonkey/`.

[15] Dongseok Jang, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[16] Andrew Keep, Arun Chauhan, Chun-Yu Shei, and Pushkar Ratnalikar. RubyWrite: A Ruby-embedded domain-specific language for high-level transformations. School of Informatics and Computing, Indiana University, Bloomington, 2009.

[17] Pedro G. Leon, Blase Ur, Rebecca Balebako, Lorrie Faith Cranor, Richard Shay, and Yang Wang. Why Johnny can't opt out: A usability evaluation of tools to limit online behavioral advertising. Technical Report CMU-CyLab-11-017, Carnegie Mellon University, Pittsburgh, PA 15213, October 2011.

[18] Mozilla. Firebug. `http://getfirebug.com/`, 2012.

[19] Gregor Richards, Christian Hammer, Brian Burg, and Jan Vitek. The eval that men do: A large-scale study of the use of eval in JavaScript applications. In *European conference on Object-Oriented Programming (ECCOP)*, 2011.

[20] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. *ACM SIGPLAN Not.*, 45:1–12, June 2010.

[21] Konrad Rieck, Tammo Krueger, and Andreas Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Annual Computer Security Applications Conference (ACSAC)*, pages 31–39, 2010.

[22] Tom Simonite. Ad Men and Browser Geeks Collide Over Web Protocols. `http://www.technologyreview.com/news/428050/ad-men-and-browser-geeks-collide-over-web/`, June 2012.

[23] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. JavaScript instrumentation for browser security. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2007.

[24] Chuan Yue and Haining Wang. Characterizing insecure JavaScript practices on the web. In *International World Wide Web Conference (WWW)*, 2009.

APPENDIX

This section lists the details of some of the features we used to distinguish ad-related scripts from other scripts.

Code Structure

Code structure is determined using the AST nodes, instead of original source text, to eliminate the noise caused by whitespace and unusually long strings. In the following lists, we use the form `x/y` to indicate the occurrence of an AST node of type `y` in the context of `x`. For example, `return / comma`, means a `comma` operator occurring inside a `return` statement.

Following are the program structures used by the classifier.

Program constructs found in ad-related scripts

try
bitxor
shift

Program constructs found in other scripts

array
throw

Following are the syntactic idioms used by the classifier.

Idioms found in ad-related scripts

comma/decrement
bitxor/decrement
bitxor/function call
bitand/shift
ternary operator/assignment
in/left bracket
body/colon
body/body (nested complete statements)
left bracket/plus
left bracket/array
left bracket/string
if/comma
return/comma
throw/comma

Idioms found in other scripts

ternary operator/bitand
and/dot
statement/new
statements/delete
colon/string
left bracket/primary (`true`/`false`/`this`/`null`)
delete/left bracket
for/comma
throw/new
bitand/dot
or/minus
statement/unary operator
statement/dot
decrement/left bracket
if/bitand
if/bitxor
while/assign
while/dot
throw/plus

Event Names

Scripts capture types of events with different frequencies, using the functions `attachEvent` and `addEventListener`.

Event names used commonly in ad-related scripts	*Event names used commonly in other scripts*
onload	onclick
onbeforeunload	onmouseout
onerror	onmouseover
onDOMReady	onunload
mousemove	onmessage
onmousedown	onkeyup
	onkeydown
	onscroll
	onblur
	onfocus
	onfocusin
	onfocusout
	onresize
	onreadystatechange
	DOMSubtreeModified
	DOMContentLoaded

Function Calls

Note that although most of these functions are built in to JavaScript, some of them may come from popular JavaScript libraries such as jQuery.

Functions used commonly in ad-related scripts	*Functions used commonly in other scripts*
`document.write`	`isArray`
`document.referrer`	`makeArray`
`document.writeln`	`getMonth`
`String.format`	`getYear`
`get`	`removeClass`
`push`	`addClass`
`getTime`	`hasClass`
`loadScript`	`createTextNode`
`cookie`	`isFunction`
`escape`	`getAttribute`
`toGMTString`	`handleObj`
`attachEvent`	`scrollTop`
`charAt`	`preventDefault`
`substring`	

Regular Expressions

We use the following generalized regular expressions to identify keywords. Note that `TLD` in the first regex refers to any existing TLD (top-level domain), including international TLDs.

Regex used for URLs

/(\.{TLD})\/ad([szx]?|(vt)?)?[.\/_?=-])/
/(\/ads?\.(js|php|pl|htm|html|asp|aspx|cgi|css)\??$)/

Regex used for function and variable names

/(.*A|^a)ds?([A-Z_-].*)?$/