Analyzing JavaScript for Security Risks through Malware Ads

Chris Frisz
cjfrisz@indiana.edu

Brennon York
boyork@indiana.edu

Thiago Rebello
thirebel@indiana.edu

Abstract

Advertisements have become a crucial point of revenue within the current business model of the Internet. This scheme of leveraging third party content providers as a means of boosting sales while at the same time bidding on ad space to increase the individual ranking is quite unique. Many issues have arisen from the competitive model though. Advertisements have become bulky, intrusive, and even privacy-violating which causes problems for typical users. This has spawned a plethora of ad-based blocking mechanisms although none are proactive or based on ground truth as to what an advertisement is. This work intends to expand on these topics by developing a ground truth for what an advertisement is on the Internet. We show that the top ten most common filters matched on a variation of the string "ad." Further, the top five ad-displaying sites observed accounted for 1.7% of all ads seen. Additionally, we have leveraged the Mozilla Firefox browser to implement, what we believe, to be initial work into a behavior-based analysis to differentiate advertisements from non-advertisements in relation to JavaScript files. We demonstrate that there are characteristics inherent to advertisements that grant the ability to separate those files into ad-, and non-ad-, related bins. Advertisements fall into two distinct categories based on their content provider with an invocation of 31.1% of all functions loaded per file. Future work in this area will hopefully expand on these characteristics with additional metrics to build a browser-based plugin for proactive advertisement detection.

1 Introduction

Increasingly, Internet websites and applications can distribute dynamic content that caters to the needs of both consumers and content owners. This has occurred in large part due to the evolution of technologies such as the JavaScript language. JavaScript allows content providers not only to provide rich and interactive content to users, but gives them flexibility in how to present advertising to subsidize the cost of the content.

This has led to an advertising model in which first-party content providers often do not directly interface with the advertisers whose content appears on the content providers sites. Rather, content providers reserve spaces in the layout of their web pages for advertisements and sell the space to advertising agencies. These agencies in turn make arrangements with advertisers who actually generate the ad content and store it on servers belonging to the advertising agency. When users navigate to a web page, the advertisements on the page load from the advertisement agencies servers. The advertisements may in turn execute some JavaScript code to make the content interactive or fetch some related data.

While this three-part model has provided first-party content providers with a simple means of generating advertising revenue, it imposes dangers on users. Because of the process by which the advertising agencies make the arrangements with the advertisers, the content providers do not verify the content or execution of the advertisements on their websites. Thus if the advertising agency fails to recognize that an advertisement displays unwanted content or executes harmful behavior through JavaScript, content providers may inadvertently serve users with malicious ads that can hijack the users computers or violate their privacy [15].

The obfuscation of advertisement content through this common model of advertisement service and the possibility of exposing users to malicious content motivated the project. Our aim was to develop an algorithmic method for isolating malicious advertising content at the compiler level. Of particular interest to our project were malicious JavaScript code

behaviors that violate user privacy. These behaviors include stealing cookie data, location data from the URL string, and history information from the users browser cache [17]. Additionally, JavaScript code may record user behavior such as keyboard and mouse input and time examining particular page elements, which we also wished to block.

In our investigation of advertisement and JavaScript behavior, we took on two tasks. For the first, we performed a survey of ad-blocking and privacy-protecting software, both freely available and commercial, paid applications. We additionally instrumented two web browsers, HtmlUnit and Firefox 4, to log information regarding ad-related content, particularly JavaScript files and behavior.

The remainder of the report is laid out as follows. Section 2 describes our survey of ad blocking software. Section 3 deals with the two web browsers that we instrumented. In section 4 we present experimental data and analysis for the effectiveness of existing ad blockers and information gathered from the instrumented version of Firefox 4 regarding JavaScript loading and execution. In section 5 we present related work. We describe future work in section 6. Section 7 concludes.

2 Identifying Ads

With the constant growth in Internet usage, it is only expected that hosts open up their sites to trackers, who gather information from users in order to target them with advertisements. This not only poses a privacy issues, but also threatens security in the web as many advertisements might contain malware and such. While concerning, this situation also provides an opportunity for developers to design tools to block ads and prevent trackers from stealing personal information. For this section of our project, we present a study on many different ad blockers freely available and their functionality. More precisely, we focus on Adblock Plus, Ghostery, IE9 Tracking Protection Lists, and Ad Muncher. We have developed a list with regular expressions using all of these blockers, converting them to Adblock Plus format. In total, we have come up with over 40,000 filters from all of the aforementioned blockers and trackers. Our goal is to use this list to paint a graph (that is generated through an algorithm), identifying what is an ad and needs to be blocked. With this in place we can stop these ads from being loaded, providing better security and privacy to users.

2.1 Adblock Plus

Adblock Plus is perhaps the most famous ad blocker in the web, used in Mozilla Firefox browsers. From a technical standpoint, Adblock Plus makes use of Gecko, an engine that operates on top of Firefox, Thunderbird, etc. It allows applications to do content policing [8]. Once a page is requested and begins to download objects, it calls a JavaScript that looks up the address from where the object is being loaded and decides whether or not to block it. This idea is similar to blocking certain web pages amongst web browsers. In fact, when one manually configures an image to be blocked by Adblock, you are modifying the content policy to have an address where it should block objects from. Previous work shows serious improvement to the tool [1]. As of 2004, AdBlock did not provide users with the ability to select certain elements and choose to block them; in fact there were not even lists to subscribe to and users had to create their own regular expressions. It now contains several subscriptions (with Fanboy and EasyList being the most subscribed lists) with over 10,000 filters and allows users to create and maintain their own blocking lists. At that point, Adblock could also only block ⟨IMG⟩, ⟨EMBED⟩, ⟨OBJECT⟩, and ⟨IFRAME⟩ HTML tags [1]. To address the lack of elements being blocked by Adblock, [1] describes an HTML tag that is to be blocked as a DOM path starting at the ⟨HTML⟩ root node. The path body:1/table:2/tr:3, for example, identifies the third row in the second table within the documents first body. A problem with this approach is that nothing can be blocked until a page is fully loaded so if a page does not finish loading for any reason, the user is still exposed to the ad. In our project, we plan to make sure that we can block ads before they load, while the page is loading, rather than after it is already loaded.

In comparison to the past, Adblock Plus can block the following today [2]:

- script – external scripts loaded via HTML script tag
- image – regular images, typically loaded via HTML img tag
- background – background images, often specified via CSS

- stylesheet – external CSS stylesheet files
- object – content handled by browser plugins, e.g. Flash or Java
- xbl – XBL bindings (typically loaded by -moz-binding CSS property)
- xmlhttprequest – requests started by the XMLHttpRequest object
- object-subrequest – requests started plugins like Flash
- dtd – DTD files loaded by XML documents
- subdocument – embedded pages, usually included via HTML frames
- document – the page itself (only exception rules can be applied to the page)
- elemhide – for exception rules only, similar to document but only disables element hiding rules on the page rather than all filter rules

One obvious drawback from Adblock Plus is its reliance on blacklists, rather than automatic identification and blocking of ads. Previous work [1] aimed at moving towards algorithmically blocking ads, developing an algorithm for automatically generating wildcarded URLs, an extension that allows for web updates of the blacklist, and an algorithm for blocking arbitrary HTML page sections. Given two strings and comparing them, the project aimed to keep as many similarities as possible, but replace differences with wildcards in order to create filters. The goal is to determine a longest common subsequence of two strings and insert wildcards between the fragments which form the subsequence. The process uses two nested loops to fill a table with dimensions `(n+1)` by `(m+1)`, where n and m are the lengths of the two strings. Whenever two characters match, the value of the cell is increased by one; if the characters do not match, then the larger of the two values is chosen. Instead of returning a longest common subsequence as one string, it returns a list of subsequence fragments that make up the subsequence. A wildcarded URL can then be generated by inserting wildcard characters in between those fragments. This task is not optimal as the result of a general expression could be "http://*" which would block unwanted content. Also, another problem with this algorithm is that they are order-dependent, meaning that a new expression is created as soon as a new URL is added to the list, rather than the list updating itself for each site that a user decides to block.

Aside from ad blocking, Adblock Plus also contains an add-on for element hiding that allows for removal of certain part of web pages that a user does not want to see [3]. This can be particularly useful in web pages that have embedded code for text ads, enabling Adblock Plus to completely remove them. There are many different ways to block ads through Adblock and if none of the regular expression building techniques are used, then it will match against anything, which can block unwanted content, rather than being useful. If used correctly, it is particularly effective, both visually and technically.

2.2 Ghostery

Ghostery is an ad blocker designed by the major companies that operate trackers, ad servers, analytics services, page widgets, and other page elements. Like Adblock Plus, it uses a list to block advertisements, but rather than allowing for multiple subscriptions or creation of personal lists, Ghostery only blocks content from the companies that it supports (and only the ones that users choose to block). Once Ghostery detects companies with page elements present on the site you are browsing, it notifies you by listing those company names in a bubble, avoiding its loading within the page. It can also prevent domains in our library from creating browser cookies. Ghostery currently possesses 486 of what they call "bugs" and 314 cookie protection items, which are currently just experimental and not guaranteed to work [10].

Ghostery relies on user data and user submissions. One of its services, called Ghost Rank [10], allows users to anonymously participate in an information-gathering panel designed to improve Ghostery performance and create a census of advertisements, tracking beacons, and other page scripts across the web. The data collected is used only in aggregate, contains no personally identifiable information, and is never used to target advertising.

Its list of filters contains only URLs that are a part of Ghostery. In order to use these in our project, we simply extracted the URLs associated with these ad providers and added to our list of filters.

2.3 IE9 Tracking Protection

Internet Explorer 9 is the latest browser from Microsoft. It contains several innovations, including Tracking Protection, which allows users to block advertising and tracking based on lists provided by some companies. The advantage of Tracking Protection is that it is a built-in function of IE, rather than a plug-in like Adblock Plus. However, users still have to subscribe to lists. As IE9 has become more popularamongst browsers since its release, more lists have been added. At the time IE9 became available to users, there were only 5 tracking lists, but now there are over 10. EasyList is perhaps the most popular list, as it is based on Adblock Plus list as well as Fanboy, which includes add-on lists for foreign filters which are particularly useful for us to identify advertisements. Because IE9 is quite new, it does not have an extensive list of filters if compared to Adblock Plus and because of that it is not as effective.

Technically, IE9's filter list is extremely simple. The first line of each list starts with "msFilterList" and each existing line after that is a rule. A "-d" means that the rule blocks traffic from the domain on that line that contains the substring shown after the domain (a "-" sign works the same way). A "+d" means that requests to the domain on the same line are allowed and when multiple lists target the same domain and substring, the Allow ("+d") rule wins [11]. Table 1 from [11] shows the effectiveness of IE9 Tracking Protection as of February 2011.

Aside from lists, IE9 also allows for automatic blocking of some trackers, although there are no details about how IE9 handles this specifically. Essentially, as users browse the web, when websites contain content from the same provider, they can choose to block that content.

Also, IE9 provides a feature called "Do Not Track HTTP header transmission" which basically tells websites not to track a user. This allows users to choose not to be tracked [12]. This is obviously not a guarantee that sites will not track users, but it is definitely a step in the right direction.

Publisher	Block	Allow
EasyList	2,189	47
PrivacyChoice	463	1
Abine	94	0
TRUSTe	0	3,958

Table 1: Effectiveness of filter lists for IE9 Tracking Protection

2.4 Ad Muncher

Ad Muncher was another ad blocker examined by us. It combines the use of lists with built-in functions to remove links to URLs, block retrieval to URL, remove images with "alt" text, remove divs/spans with text, remove tables with text, etc. [7] Ad Muncher does not require manual setup as a proxy server, but it instead intercepts winsock calls in memory and redirects Internet traffic through itself. What is more, like Adblock Plus, it allows for custom lists, letting the user choose, for example, a particular image within a web page that he/she wants to block. It is visually effective as 1st and 3rd party ads are blocked, and embedded text advertisements are also removed from view.

Rather than blocking and completely removing some ads, it displays a [Munched] message that lets users know that that section of the page is potentially an ad, but still allows them to visit the link and view the ad. There is, however, an option that allows for filtering out links to protect users from accessing potentially unsafe advertisement links. The user also has the option to report a link or webpage that he/she might think of as harmful.

One disadvantage with Ad Muncher, however, is the fact that it is not free of charge, and like all previously discussed blockers, it depends on a list to block ads. Our goal is to avoid lists and constant updates to them, allowing users to block ads on the fly, rather than depend on other people to identify ads, contact subscription companies, and wait for updates.

3 Browser Implementations

3.1 HtmlUnit And Rhino

For our first attempt at instrumenting a web browser to log ad-related files, including JavaScript, we started by modifying HtmlUnit, a Java-based testing framework for web interactions. It is dubbed the "GUI-less" web browser and is built off the the JUnit testing framework commonly used for unit testing Java applications and shares much of the syntax and functionality. It aims to simulate all the interactions involved in loading web pages, including downloading resources, parsing HTML files, and ex-

ecuting JavaScript content. To achieve the latter, HtmlUnit utilizes the Rhino JavaScript engine implemented in Java.

Our project started from previous work by one of the authors for another class in which he also utilized HtmlUnit to attempt to discover characteristics of ad-related web content with respect to JavaScript. The previous work involved instrumenting a version of the PAW open source server modified by the HtmlUnit team. The server used filters in conjunction with the version of Rhino used in HtmlUnit to log the names of functions defined in JavaScript files loaded for a particular web page. It then output a log of when each function was invoked during loading and interaction with the web page. In the previous work, one of the authors further modified this version of PAW to perform logging over a set period of time and output a histogram of the function names and their invocation counts. While the modified PAW server provided some insight into JavaScript behavior, it was lacking in several respects. For one, it would not log native JavaScript functions or the libraries provided by the web browser. This was particularly severe because much malicious JavaScript behavior stems from abuse of certain built-in functions, such as 'eval.' Additionally, the modified PAW server could not log arguments passed to functions, another potential marker for malicious JavaScript behavior.

To this end, the current project began by modifying the source code for HtmlUnit itself. One of the first modifications we implemented was a log of the URIs for each file requested in the process of loading a web page. This change intercepted the names of web page resources as they were loaded. We used these logs to generate graphs of ad-related content using a graphing program provided to us that used a set of pre-defined filters for ad-related content based on those used for AdBlock. We also instrumented the implementation of Rhino used in HtmlUnit to log the names of the JavaScript functions as they were loaded into the parser of the JavaScript engine, associating the functions to the files from which they originated.

At this stage in the project, we were able to observe some behavior of ad-related content, but our HtmlUnit implementation incurred some serious limitations on our ability to gain further insight. For one, since HtmlUnit does not have any visual elements (being a GUI-less web browser), we could not visually inspect ad content to determine whether our implementation correctly identified advertisements. Additionally, we could not clearly tell whether HtmlUnit requested all of the visual content for web pages (e.g. images and video) due to both the aforementioned lack of a GUI and because of an absense of URIs denoting visual content. Furthermore, the compilation of the Rhino JavaScript engine to Java bytecode and subsequent execution in the Java Virtual Machine made the low-level JavaScript function invocations opaque to us. These issues prompted us to migrate our implementation to Firefox.

```
Function(‘‘x’’, ‘‘return x + 1;’’);
Function(‘‘x’’, ‘‘return x - 1;’’);
```

Figure 1: Calls to the 'Function' constructor to create simple increment and decrement closures.

3.2 Firefox and SpiderMonkey

The second JavaScript engine we ended up choosing for instrumentation was SpiderMonkey. Since this needed a web browser front-end, and not only the JavaScript engine, we chose to utilize Mozilla Firefox. For our implementation we pulled down the latest version of the Firefox web browser (version 4.0 at the time of this writing). Since the focus was on the JavaScript engine this granted access to the enhanced JägerMonkey version.

Behavior within JavaScript becomes very difficult to define. Because we look at this through the lens of a user we can see everything from a client-side. It is this aspect that pushes our behavioral analysis to the file level, as web browsers work by requesting files. When looking at each JavaScript file requested it is broken down into its primary constituents and classified based on those. In this preliminary work we focus on the file name, functions loaded per file, and invocations of each function. Multiple issues arise when attempting to classify these because of the many powerful features of the JavaScript language, such as the 'eval' library call that executes an arbitrary string as JavaScript code. Among those, special cases still needed to be handled in the instance of two different files with the same function name. This would typically be seen in the case of an 'Anonymous' function call, created by invoking the 'Function' constructor library call which generates an unnamed closure. One could declare two functions that do separate things to two different variables, but, because of the JavaScript language,

they would both be labeled, at invocation, with a name of 'Anonymous' as seen in Figure 1. Because of this issue a unique ID was needed to ensure which function invocation was actually happening. Without this the function invocation recordings would be skewed with false positives, matching with the first instance it found. In the above example, the first occurrence of an 'Anonymous' function call would represent the invocation count for all 'Anonymous' functions loaded and invoked in a given file.

To gather those function calls, invocations, and files a few modifications of the SpiderMonkey source code were needed. The first parts of these tuples of information to be collected were the file names. Each JavaScript file must first be requested during typical website loading (e.g. an HTTP POST request). The point where those files get sent to the SpiderMonkey engine is where they are captured. The actual instrumentation point begins within the parser (jsparse.cpp) at the time of new parser instantiation. SpiderMonkey works by instantiating a new parser for each file it receives then iterating over that file loading in all functions into the environment regardless of its assignment or invocation style. To gather each individual function an additional modification to the parser was needed. Because the parser will iterate over the file, it would automatically declare the type of any given object seen. This iteration process is where each function name can be gathered, considering the parser will predefine which objects are functions. At that same time the unique ID is collected as well. At this stage in the project, the unique ID consists of a function name and decompiled source code pair. We obtain the function name during parsing of each JavaScript file, putting it into a tuple along with the name of that file. This tuple is made visible to the interpreter portion of the JavaScript engine, which can decompile a function's source code. We pair the source code back to the function name on the first invocation of a function, and increment the invocation count for each time the name and source code pair is observed. Technically speaking, this will not differentiate between functions that share completely identical name and source code pairs. We ignore this occurrence both because it is highly unlikely to happen and because considering two such functions would not invalidate our analysis of JavaScript behavior.

To handle individual data collection we used a shared memory signaling system. When the web browser launches it generates a specific section of shared memory at a size of one byte. This one byte holds the character that is checked against when the JavaScript engine is invoked. If a specific character is read in then all of the contents of each tuple will be dumped into a human-readable file. These files can then be statistically analyzed to determinebehavior. To actually signal the shared memory a separate application was built. This enabled reading and dumping of data when needed which became increasingly important as more data was collected from each file. Additionally, Firefox utilizes the SpiderMonkey engine extensively on startup which can generate behavior. To actually signal the shared me a massive amount of unnecessary data that would need to be trimmed off without the use of signaling application.

Another set of data that was collected were the individual uniform resource identifiers (URI's) that each site requests to properly load. These were dumped from the source file nsDocLoader.cpp where each request is initially made. Again, as with the JavaScript information, each URI is dumped into a human-readable file to be parsed. This data was used more exclusively to understand the weight that each file type had within a typical webpage. These URI's and the DOM tree were areas explored in hopes of deterministically finding additional behavior mechanisms within a website that were disassociated from the JavaScript engine. The DOM tree, unlike the URI parser, has yet to be implemented which we note as future work.

4 Data Analysis

4.1 Ad Blocking Tools

In this part of the paper, we present an analysis of our combined list of ad blockers, providing information such as the frequency that a certain filter is matched based on Alexa's Top 1,000,000 sites. Due to time constraints, rather than applying our list to all of the 1M sites, we decided to use the top and bottom 1,000 sites instead. We believe that this should give us a representative set of ads based on their model as well as from where they come from (different countries, etc.).

In order to perform this analysis, the first step was to gather URLs from the 2,000 sites examined. We used HTTP Analyzer [9] for this task and then applied our filters to the list of URLs, determin-

ing how often a certain filter was blocked. To try and force Firefox to load a variety of ads rather than the same ads, we changed DNS servers as well as language settings within the browser. We believe that simulating a different geographical location will force ads from that location to be loaded into pages. To be more specific we used default settings and English, brahms.uol.com.br and Portuguese (Brazil), dns.sczn.de and German, and aussaguel.inmarsat.francetelecom.fr and French as the basis DNS servers and languages, respectively.

A lot of pages out of the 2000 sites that we visited from Alexa.com were search engines and social networking sites. Google for example, is popular in several different countries and although it loads URLs in its home pages that match our filters, it obviously does not retrieve as many URLs as it would in case a search query was performed. To address that, we chose to use the top 10 keywords from Google Insights [14] for year 2010 for each Google site in our list and retrieve the URLs from that search query. This listed included the keywords "iPad," "Justin Bieber," "fb," "facebook en espanol," "4shared," "www.facebook.com," "twitter," "facebook login," "taringa," and "face." For social networking sites, one does not see advertisements unless he or she logs in to their respective profile, where they will receive ads based on information that is publicly available through their profiles. We used personal log in information to log in to facebook.com, linkedin.com, and orkut.com.br since we already had profiles in those respective social networks. However, we did not have the time and resources to create different profiles in the many different social networking sites in the list and analyze how advertising is performed on them. We believe that in the future, further studies can be done in the areas of search engines and social networking sites in order to analyze where ads are coming from and how they are targeting users (geographical location, likes and interests, etc.).

From our analysis, the top 10 most common filters out of all ad blockers were ".ad.", "/ad", "/adj?", "/ads?", ".cnt?", ".ads.", "ads"/stat.?", ".com/ad?", and "/ads.". These together correspond to 31.9% of all ads found, showing that advertisements are commonly tied to variations of the word "ads". Individually, per list, Adblock Plus blocked "ads" the most (150,032 matches), followed by ".com/ad?" (108,288 matches), while Ghostery blocked ads from "advertising.com" 7,936 times, followed closely by "salesforce.com" with 7,648 matches. Ad Muncher and IE9 blocked ".ad." and ".com/ad?" the most with 566,032 and 108,288 matches, respectively. It should be noted here that both IE9 and Adblock Plus blocked the filter ".com/ad?" the same amount of times. This can be due to the fact that EasyList produces lists to both blockers.

The top 5 sites with the most advertisement matches were: "qq.com", "yahoo.co.jp", "163.com", "bbc.com.uk", and "xvideos.com". These sites together accounted for 1.7% of all ads seem. It is to be noted, however, that some sites out of the 2,000 examined might have some advertisements or trackers that could not be blocked by the filter list that we gathered, most likely because they were not up-to-date or possibly because the hosts founda way to avoid blockers. In either way, it strengthens our purpose for an algorithm rather than lists to block content from ad providers.

4.2 JavaScript Execution

When analyzing the JavaScript engine we chose to minimize the dataset from the original 2,000 websites to a subset of 10 sites. This reduction was necessary because of the gross amount of data that was collected per site as well as the in-depth behavioral analysis needed distinguish an advertisement against typical JavaScript. These ten sites were chosen based on social evaluation amongst peers of what constitutes as an 'annoying' advertisement paradigm. The culmination of these sites attempted to encompass the entire gamut of advertising schemes, ranging from advertisements that intentionally block content to movable ads that will 'all' from the top of the site. The ten sites are given, as well as their number of associated unique JavaScript files, in Table 2.

The detailed analysis of advertisement blockers and their implementations has provided us with an initial ground truth with which to compare the individual JavaScript files seen. Utilizing all of the ad blockers mentioned previously we were able to determine which JavaScript files would have been blocked had any of those blockers been running at the time of website request. This provided an initial binning of JavaScript files loaded in as either advertisement-related or advertisement-free. Our behavioral analysis begins with the attempt to distinctly separate these two groups.

One of the first behavioral traits seen within the data, through simple visual analysis, was a recursive

Web site	# JavaScript files
salon.com	44
cnn.com	35
sidereel.com	32
slate.com	32
nytimes.com	26
escapistmagazine.com	20
monster.com	20
ign.com	19
trailers.apple.com	8
vacaway.com	2
Average	23.8

Table 2: Total loaded JavaScript files for each of the 10 web sites examined

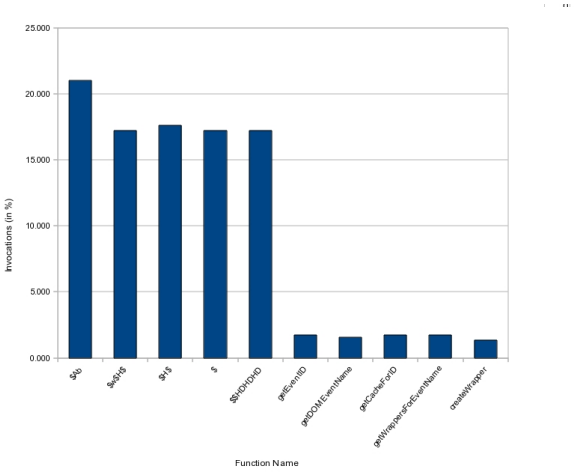

Figure 2: A graph of JavaScript invocation counts

function naming definition used within the Google advertisement delivery system. Seeing this in a text-based document over a typical JavaScript file provided a quick means of identification as it built a tree-like structure. A typical example of such would be the three functions such as "Ta," "PaTa," and "qaPaTa." The deepest invocation is seen first (e.g. qaPaTa) and at each additional function load the leftmost two characters are stripped off, leaving the rest as the new function definition. At this time it is unsure whether this is a form of obfuscation or merely a unique way that Google delivers it advertising content. Four of the ten sites loaded utilized this Google ad-serving technique. Another interesting point was that, although this plethora of functions was defined (typically between 38 and 52), on average only five to seven of those loaded were actually invoked. This behavior continues to show throughout most of the JavaScript files seen.

Nearly all JavaScript functions read in for any given site of any given file were not executed. On average 31.1% of all functions loaded were actually invoked at some time or another for any given site. Additionally, of those 31.1% executed, an even smaller subset of those are invoked a majority of the time. Figure 2 shows a list of the top ten functions invoked by the site http://trailers.apple.com/, taken from the file /trailers/global/scripts/lib/prototype.js, and measured in percent. To give more definition to the graph, the function in the top slot was invoked 217 times, with the bottom five functions at invocation counts between 14 and 18. This strong drop in the tail was seen throughout each file in each site measured. One caveat to this design, which will be explored more in future work, is the hands-off nature of our measurement. Many advertisements rely on mouse clicks, movement, and other interactions to activate additional features. Our current work merely collects data without any additional input from the user. It is plausible that these unused functions are waiting for additional interaction from the client before they can be used, which our work does not test. Conversely, these functions could be based on other issues such as browser language, Operating System, or geographic location. Again, each of those mentioned were static in this work.

When looking at these same JavaScript files and functions in the light of advertisements and non-advertisements a few unique features were seen. First, it seems that first party content delivery has a much higher rate of function execution. A primary example is that of http://cnn.com/. They have an advertisement distribution (e.g. http://ads.cnn.com/...) that runs directly on their site which loads and executes 22 different advertisement-related functions. This 100% load-and-invoke scenario is rarely seen elsewhere. Contrary, third party content providers seem to load a multitude of functions with a marginal invocation count. As stated above Google typically loads a large number of functions with a small minority actually executing. Graphing this by function count provides one nearly identical to Figure 2 with the exception that typically fewer than ten functions are actually executed. Further, JavaScript files that are binned into advertisements seem to have a lower overall function count. Pri-

mary examples are http://ad.doubleclick.net/ and http://pagead.googlesyndication.com/ where their average function load count is 3.27. These two parties range from 3 to 4 functions loaded although they tend to load multiple separate JavaScript files, with http://ad.doubleclick.net/ being the larger culprit. This shows that advertisement-related JavaScript files typically are smaller files with possibly more separate files being loaded in. It should be noted that in the previous two cases with Google the advertisements are served from different files, one being show_ads.js (with 3 to 4 functions) against show_ads_impl.js (with 38 to 52). These smaller files seem understandable if the current distribution scheme is to first provide a static loading function that then fetches the dynamic web content. This alleviates content providers from having to ensure that new advertisements were loading and pushes the fetching onto the ad provider.

Another metric that was originally going to be tested moving into this project was that of JavaScript naming conventions, more specifically a measurement of obfuscated function names compared to those that were humanly-readable. This, unfortunately, was unable to be tested because of the nature of the issue. Determining whether or not a function call is humanly-readable, in the terms of function names, can be much more difficult than typical natural language processing. This stems from the fact that developers might label their functions with camel casing (e.g. myVar), underscores (e.g. my_var), or another manner as a way to deliniate multiple words. Further, functions could be coded in an odd manner, such as file2EOF, which causes the natural language corpuses to fail on these accounts. Because of these unique features of function naming schemes we move this problem to future work.

5 Related Work

Many other tools have been developed to stop web advertising. As previously mentioned [1] talked about improving AdBlock Plus to allow for automated generation of regular expressions, sharing of filter lists and improve the sets of items that can be blocked. [1] developed an algorithm for automatically generating wild-carded URLs, an extension that allows for web updates for previously generated lists, and an algorithm for blocking arbitrary HTML page sections. AdBlock Plus has addressed all issues described in [1] and has also added the ability to block any content from a specific page using Element Hiding Helper [3].

[4] proposes Quero, a web browser-based content filter for detecting and removing online ads. The idea behind this software is that it only uses a small set of rules to filter ads. By examining some methods and other information used by common advertisements (banner ads, video ads, text ads, pop-ups, sticky ads, ad games, interstitials, and content sponsoring as listed [4]) Quero created rules for blocking all Flash-based content by default, blocking unwanted pop-ups, ad banners based on their size, content that comes from well-known ad providers, images based on ad-related keywords in their URL, and not blocking content on sites that are whitelisted.

[4] concludes that the URL is the most efficient indicator of an ad, proposing that perhaps lists would still be better for identifying those. However, as lists grow ([4] shows 425 filters for AdBlock at this point while there are over 10,000 as of this writing), it becomes less efficient to use blacklists, thus strengthening our purpose for a blocker that can detect behavior.

In [13] a group of researchers evaluated 10 popular anti-phishing tools that use different techniques to identify phishing sites. This is particularly appealing to our work as it describes how many different tools focus on lists to block phishing material, rather than automatic blocking phishing sites.

Out of the 10 anti-phishing tools, only SpoofGuard does not use whitelists or blacklists, but rather employs heuristics to identify phishing pages. The toolbar first checks the current domain name and compares it with sites that have been recently visited by the user to catch fraudulent websites that have a similar-looking domain name. Other tools investigated utilize variations of heuristics that examine location of domain, popularity of site, user reports and ratings, and blacklist data.

[13] discovered that Spoofguard had a catch rate of over 90%, but also had a 43% false positive rate, which is something we try to avoid in our work - we do not want to block things that shouldn't be considered ads. On the other hand, the other tools performed way subpar is compared to SpoofGuard, with IE7 having a catch rate of 60%, but having none to very low false positive rate (highest one was 2% from CallingID toolbar). Most tools also took a long time (hours or even a day) to identify phishing sites, meaning that their lists could not be updating

at a fast rate. SpoofGuard in the meantime, only needed to visit a page once to determine whether it was phishy or not. Our approach is somewhat similar: we want something fast and efficient, but again, need to be careful with false positives.

[17] created a framework utilizing a source code level rewriting strategy to track information flow to and from JavaScript functions and programs with respect to privacy-related data. We share a similar goal in tracking privacy-violating behavior in the web sites, particularly with respect to JavaScript execution. Our project differs from their work in that our analysis and proposed tool operate at the compiler level rather than the source code level. This provides us with a finer-grained view of JavaScript.

[15] instrumented the HtmlUnit framework to identify malicious JavaScript behavior with respect to drive-by downloads and malicious JavaScript code intended to attack the users computer (i.e. heap overflow attacks). Like [15], we utilized HtmlUnit for part of our analysis. [15] focuses on maliciousness with respect to the users machine rather than our focus on user privacy.

[16] developed Zozzle, a tool for statically analyzing JavaScript code for malicious behavior. The approach in [16] differs from ours in several ways. This includes that our analysis is dynamic, executing as the user navigates within the web browser. Zozzle also depends on a separate tool for de-obfuscation of JavaScript code before it can work effectively, whereas our implementation analyzes unmodified, possibly obfuscated code. In fact, the obfuscation of the code is used as one of the markers for possible malicious behavior.

[18] presented ConScript, a system for enforcing fine-grained security policies for JavaScript execution. Like our work, ConScript operates at the compiler level and in part aims to prevent malicious JavaScript code from obtaining private user data. In contrast to our project, ConScript depends on content providers to specify policies for JavaScript behavior, whereas our tool aims to ensure privacy automatically.

6 Future Work

Currently our JavaScript analysis implementation is limited to identifying the names of the functions invoked during interactions with a web page. While this provides a number of insights into JavaScript functionality, we intend to further instrument SpiderMonkey to provide broader details about JavaScript behavior. Among our next steps is logging the arguments passed to each invocation, since these can signify whether a function operates on private user data, such as strings including the browser version. We also intend to observe JavaScript in relation to the document object model (DOM) tree, comparing its original loading state to that after the application of ground truth. Additionally, the current scheme of using the function name and decompiled source code string presents a number of operational disadvantages. These stem both from high memory usage to store a large number of potentially long strings and from the amount of computation time needed to match these strings on function invocation. Thus we intend to find an alternative identifier, such as a unique memory address for the function object used in SpiderMonkey's interpreter.

A long term goal of the project is to develop a tool akin to the AdBlock Plus browser plugin that could be distributed to users for blocking malicious ad content. As noted throughout this report, we hypothesize that our behavioral analysis of JavaScript would allow us to create a tool that would have both higher accuracy in identifying ad content and operate without need for blocking lists. Hopefully such a tool could help ensure that users have the ability to maintain their privacy. This of course raises the issue of whether such a tool would circumvent the primary means of financing content on the web, but we leave this discussion for when we can make such a tool available.

7 Conclusions

These initial findings on our behavioral analysis of advertisement-related JavaScript files shows that there is conclusive evidence that they do differ in behavior. With the current metrics there is not enough dividing evidence to provide distinct behavioral bins although, as noted in section 6, we believe that this demonstrates first steps into the world of advertisement-related JavaScript binning. Our work was able to utilize current tools and measurements to build a comprehensive ground truth as to what constitutes an advertisement. Additionally, We were able to show that JavaScript files related to ads, and their providers, do provide certain discernable features that, coupled with future metrics, could prove accurate as a tool to distinguish an advertisement

on the Internet. More specifically, we demonstrated that ad-related files typically follow one of two different rulesets; one, a number of small files load a variety of functions into the JavaScript environment with few function invocations, and two, a single, large, file loads the entirety with a majority of its functions being invoked.

References

[1] Justin Crites and Mathias Ricken. Automatic Ad Blocking: Improving AdBlock for the Mozilla Platform. Rice University. Houston, TX.

[2] Writing Adblock Plus Filters. Adblockplus.org. Adblock Plus, n.d. Web. 2 May 2011.

[3] Element Hiding Helper. Adblockplus.org. Adblock Plus, n.d. Web. 2 May 2011.

[4] An Effective Defense against Intrusive Web Advertising. Viktor Krammer, Secure Business Austria, Vienna University of Technology, A-1040 Vienna, Austria

[5] NoScript. InformAction., n.d. NoScript JavaScript/Java/Flash blocker for a safer Firefox experience! what is it? InformAction. Web. 2 May 2011.

[6] Privoxy. Privoxy Developers, n.d. Privoxy Home Page. Web. 2 May 2011.

[7] Ad Muncher Ad Muncher Wiki. Admuncher.com. Ad Muncher, n.d. Web. 2 May 2011.

[8] FAQ Adblock Plus internals. Adblockplus.org. Adblock Plus, n.d. Web. 2 May 2011.

[9] IEInspector HTTP Analyzer HTTP Sniffer, HTTP Monitor, HTTP Trace, HTTP Debug. Ieinspector.com. IEInspector Software LLC, n.d. Web 2 May 2011.

[10] Ghostery. Evidon, Inc., n.d. Web. 2 May 2011.

[11] Privacy protection and IE9: who can you trust? Zdnet.com. Ed Bott, ZDNet, 14 Feb. 2011. Web. 2 May 2011

[12] IE9 follows Firefox 4s lead on Do Not Track Computerworld. Computerworld.com. Gregg Keizer, Computerworld, 16 Mar. 2011. Web. 2 May 2011

[13] Zhang, Yue, Serge Egelman, Lorrie Cranor, and Jason Hong. "Phinding Phish: Evaluating Anti-Phishing Tools." Carnegie Mellon University. Pittsburg, PA.

[14] Google Insights for Search. Google.com. Google, n.d. Web. 2 May 2011.

[15] Cova, Marco, Christopher Kruegel, and Giovanni Bigna. "Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code." Raleigh, North Carolina, 2010.

[16] Curtsinger, Charles, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. "Zozzle: Low-overhead Mostly Static JavaScript Malware Detection." Technical Report, Microsoft Research, 2010.

[17] Jang, Dongseok, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. "An Empirical Study of Privacy-Violating Information Flows in JavaScript Web Applications." Chicago, Illinois, 2010.

[18] Meyerovich, Leo and Benjamin Livshits. "ConScript: Specifying and Enforcing Fine-Grained Security Policies for JavaScript in the Browser." Oakland, California, 2010.