"NoFus: Automatically Detecting" + String.fromCharCode(32) + "ObFuSCateD ".toLowerCase() + "JavaScript Code"

Scott Kaplan
Amherst College

Benjamin Livshits and Benjamin Zorn
Microsoft Research

Christian Siefert
Microsoft

Charlie Curtsinger
University of Massachusetts Amherst

Microsoft Research Technical Report

MSR-TR-2011-57

Microsoft® Research

Abstract

Obfuscation is applied to large quantities of benign and malicious JavaScript throughout the web. In situations where JavaScript source code is being submitted for widespread use, such as in a gallery of browser extensions (e.g., Firefox), it is valuable to require that the code submitted is not obfuscated and to check for that property. In this paper, we describe NOFUS, a static, automatic classifier that distinguishes obfuscated and non-obfuscated JavaScript with high precision. Using a collection of examples of both obfuscated and non-obfuscated JavaScript, we train NOFUS to distinguish between the two and show that the classifier has both a low false positive rate (about 1%) and low false negative rate (about 5%).

Applying NOFUS to collections of deployed JavaScript, we show it correctly identifies obfuscated JavaScript files from Alexa top 50 websites. While prior work conflates obfuscation with maliciousness (assuming that detecting obfuscation implies maliciousness), we show that the correlation is weak. Yes, much malware is hidden using obfuscation, but so is benign JavaScript. Further, applying NOFUS to known JavaScript malware, we show our classifier finds 15% of the files are unobfuscated, showing that not all malware is obfuscated.

1. Introduction

In many ways, JavaScript has become the new assembly language. It is used for complex web sites such as Facebook and also for reasons as diverse as creating browser extensions and operating system widgets. JavaScript is a highly dynamic, expressive language, which means that much of the code that is executed can be generated at runtime. However, the expressiveness of JavaScript is often misused by attackers to create obfuscated malware, which has led to research in trying to detect malware using both static and runtime techniques [1, 3, 5, 8, 10, 21].

Of course, there is plenty of *benign* code that has been obfuscated, often as a light form of intellectual property protection, by toolkits such as JavaScript Obfuscator [15]. In many contexts, it is valuable to distinguish between "good" and "bad" JavaScript code, for instance, when accepting an application to a centralized software repository such as a browser extension gallery for a browser such as Firefox or Google Chrome. Some techniques such as AdSafe [6], Caja [17], and Gatekeeper [13] severely restrict the expressiveness of allowed JavaScript to allow analysis. Often such strict restrictions are not tolerable for large-scale industrial use.

In this paper, we present NOFUS, a tool that uses automatic techniques for determining whether a piece of JavaScript code has been obfuscated for any purpose, malicious or otherwise. NOFUS provides an *obfuscation score*, which shows how likely a particular input JavaScript file is to be obfuscated; this value can be thresholded in practice.

Previous work assumes that obfuscated code is also malicious and consequently conflates the two concepts. This approach can have negative consequences on the false positive rate of malware detectors. For example, our recent experience with the Norman Antivirus engine [19] indicates that it sometimes triggers on obfuscated JavaScript that is not necessarily malicious, resulting in possible false positives. In a web crawl that resulted in 517 Norman AV alerts total, we observed that 195 (37.7%) of them were caused by obfuscated JavaScript.

Unlike all prior work, we focus on defining a forgiving metric of whether JavaScript is amenable to further analysis, either manual or through automation. If a piece of code passes our fast initial filter, previously proposed static techniques [4, 8, 13] can be used to determine whether is benign.

Much of the previous work attempts to classify malware broadly, and does not attempt to distinguish between obfuscation and other forms of malicious intent. The little work that does, considers obfuscation specifically [3], and attempts to measure obfuscation through string pattern analysis. Our approach differs in that it is based on whole-program classification using a Baysian classifier over the JavaScript AST similar to what is done in the Zozzle project [8].

Using hand-categorized training sets, we train a static classifier to compute the obfuscation score metric and show that it has both a low false positive rate (about 1%) and low false negative rate (about 5%). Beyond measuring its effectiveness, we apply our classifier to several large collections of existing JavaScript and consider the degree of obfuscation present in these collections. To give readers more intuition about what obfuscated code actually looks like, we show some obfuscated code that our classifier correctly detected in JavaScript extracted from Alexa top 50 websites in Figure 2.

```
// VERSION 5.2.15-27
_="\u0009ubq#el\u0071wzlmf#>#xpn\u0071Wjnf9#m\u0066t
#Gbwf+133\u0036/3/26*\u002ddfwWjnfylmfLeep\u0066w+*/tm
\u0077qWjnf9#mf\u0074#Gbwf+1336/\u0035/26*-dfwWjnf
\u0079l\u006dfLeep\u0066w+\u002a/‘lnsp\u0039#X\u005e
/b‘wjuf[#\u0039#x\u0009!Eobpk!#9#\u0058#!Pkl‘
htbufEobp [...]";
eval("_\u005f=\u0027’;f\u006fr(i\u003d0;i<\u005f.l [...]
\u0072Co\u0064e\u0041t(i)\u005e3);e\u0076\u0061l
(\u005f_)");
document.write((function(){a=’’;for(i in s)
{a+=(s.charAt(i)+32);};return a;})())"
```

Figure 1: Fragment of Obfuscated JavaScript Discovered in Alexa Top 50 Website.

1.1 Paper Contributions

This paper makes the following contributions:

- This is the first paper the considers the problem of detecting JavaScript obfuscation without conflating such obfuscation with malicious intent. We make the case for why such detection is valuable and present an effective solution.
- We present NOFUS, an obfuscation detector that uses a static classifier, and show that it has low false positive and false negative rates.
- We apply NOFUS to a collection of existing JavaScript code to present an assessment of the quantity of obfuscated JavaScript present in collections such as the Alexa top 50, popular JavaScript libraries, and known malware.
- We demonstrate that malware is not always obfuscated if enough unfolding is applied before analysis. Similarly, we find uses of obfuscation in benign JavaScript, demonstrating that obfuscation detectors that assume malicious intent will likely have many false positives.

1.2 Paper Organization

The rest of the paper is organized as follows. Section 3 provides background for this work. Section 4 describes our implementation, which is largely based on previous work [8]. Section 5 presents the methodology we use for evaluation, while Section 6 presents our results. We discuss related work in Section 7, and conclude in Section 8.

2. Introduction

In many ways, JavaScript has become the new assembly language. It is used for complex web sites such as Facebook

and also for reasons as diverse as creating browser extensions and operating system widgets. JavaScript is a highly dynamic, expressive language, which means that much of the code that is executed can be generated at runtime. However, the expressiveness of JavaScript is often misused by attackers to create obfuscated malware, which has led to research in trying to detect malware using both static and runtime techniques [1, 3, 5, 8, 10, 21].

Of course, there is plenty of *benign* code that has been obfuscated, often as a light form of intellectual property protection, by toolkits such as JavaScript Obfuscator [15]. In many contexts, it is valuable to distinguish between "good" and "bad" JavaScript code, for instance, when accepting an application to a centralized software repository such as a browser extension gallery for a browser such as Firefox or Google Chrome. Some techniques such as AdSafe [6], Caja [17], and Gatekeeper [13] severely restrict the expressiveness of allowed JavaScript to allow analysis. Often such strict restrictions are not tolerable for large-scale industrial use.

In this paper, we present NOFUS, a tool that uses automatic techniques for determining whether a piece of JavaScript code has been obfuscated for any purpose, malicious or otherwise. NOFUS provides an *obfuscation score*, which shows how likely a particular input JavaScript file is to be obfuscated; this value can be thresholded in practice.

Previous work assumes that obfuscated code is also malicious and consequently conflates the two concepts. This approach can have negative consequences on the false positive rate of malware detectors. For example, our recent experience with the Norman Antivirus engine [19] indicates that it sometimes triggers on obfuscated JavaScript that is not necessarily malicious, resulting in possible false positives. In a web crawl that resulted in 517 Norman AV alerts total, we observed that 195 (37.7%) of them were caused by obfuscated JavaScript.

Unlike all prior work, we focus on defining a forgiving metric of whether JavaScript is amenable to further analysis, either manual or through automation. If a piece of code passes our fast initial filter, previously proposed static techniques [4, 8, 13] can be used to determine whether is benign.

Much of the previous work attempts to classify malware broadly, and does not attempt to distinguish between obfuscation and other forms of malicious intent. The little work that does, considers obfuscation specifically [3], and attempts to measure obfuscation through string pattern analysis. Our approach differs in that it is based on whole-program classification using a Baysian classifier over the JavaScript AST similar to what is done in the Zozzle project [8].

Using hand-categorized training sets, we train a static classifier to compute the obfuscation score metric and show that it has both a low false positive rate (about 1%) and low false negative rate (about 5%). Beyond measuring its effectiveness, we apply our classifier to several large collections

```
// VERSION 5.2.15-27
_="\u0009ubq#el\u0071wzlmf#>#xpn\u0071Wjnf9#m\u0066t
#Gbwf+133\u0036/3/26*\u002ddfwWjnfylmfLeep\u0066w+*/tm
\u0077qWjnf9#mf\u0074#Gbwf+1336/\u0035/26*-dfwWjnf
\u00791\u006dfLeep\u0066w+\u002a/‘lnsp\u0039#X\u005e
/b‘wjuf[#\u0039#x\u0009!Eobpk!#9#\u0058#!Pkl‘
htbufEobp [...]";
eval("_\u005f=\u0027’;f\u006fr(i\u003d0;i<\u005f.l [...]
\u0072Co\u0064e\u0041t(i)\u005e3);e\u0076\u0061l
(\u005f_)");
document.write((function(){a=’’;for(i in s)
{a+=(s.charAt(i)+32);};return a;})())"
```

Figure 2: Fragment of Obfuscated JavaScript Discovered in Alexa Top 50 Website.

of existing JavaScript and consider the degree of obfuscation present in these collections. To give readers more intuition about what obfuscated code actually looks like, we show some obfuscated code that our classifier correctly detected in JavaScript extracted from Alexa top 50 websites in Figure 2.

2.1 Paper Contributions

This paper makes the following contributions:

- This is the first paper the considers the problem of detecting JavaScript obfuscation without conflating such obfuscation with malicious intent. We make the case for why such detection is valuable and present an effective solution.
- We present NOFUS, an obfuscation detector that uses a static classifier, and show that it has low false positive and false negative rates.
- We apply NOFUS to a collection of existing JavaScript code to present an assessment of the quantity of obfuscated JavaScript present in collections such as the Alexa top 50, popular JavaScript libraries, and known malware.
- We demonstrate that malware is not always obfuscated if enough unfolding is applied before analysis. Similarly, we find uses of obfuscation in benign JavaScript, demonstrating that obfuscation detectors that assume malicious intent will likely have many false positives.

2.2 Paper Organization

The rest of the paper is organized as follows. Section 3 provides background for this work. Section 4 describes our implementation, which is largely based on previous work [8]. Section 5 presents the methodology we use for evaluation, while Section 6 presents our results. We discuss related work in Section 7, and conclude in Section 8.

3. Background

This section provides additional background on the JavaScript language and obfuscation techniques used in that context. The relevance of this topic comes from the fact that JavaScript code is everywhere. Indeed, JavaScript is used for large-scale web site development, with libraries such as

jQuery and Prototype increasing adoption. JavaScript is also used in the mobile context; for instance, Palm OS (now webOS) is largely JavaScript-based.

However, often JavaScript code is used as an *extension language*. For instance, browser extensions in Google Chrome and Firefox are written in JavaScript. It is also used for desktop operating system widgets in the Windows sidebar and Yahoo widgets (Konfabulator) dashboard. JavaScript is used in Facebook applications [16, 17] and to program rich advertisements [6]. In all these contexts, there is both an opportunity and a need to *check* the JavaScript code before it is accepted. This paper advocates NOFUS, a flexible and lightweight filter that can be used for this purpose.

3.1 Centralized Software Repositories

There is a pronounced trend in the industry towards a centralized software distribution model, where the platform manufacturer is responsible for validating, maintaining, and hosting the software being distributed. Some prominent examples of this model include mobile apps in the app store hosted by Apple, Google, and Microsoft for their respective mobile operating systems. Browser manufacturers such as Mozilla and Google host extension galleries for their products. Operating systems manufacturers such as Microsoft offer software through sites such as Windows Marketplace.

In the context of a centralized software repository, when the software malfunctions, due to either a genuine bug or a security exploit, the blame is often placed on the owner of the platform. For instance, the fact that many Firefox extensions slow down the browser startup process makes Mozilla Firefox look bad. As such, there is often a need to check the software before allowing it to be hosted. Apple performs a largely manual review process before allowing apps into the app store. Mozilla relies on a community-based beta phase for extensions. NOFUS presented in this paper provides a degree of automation when it comes to enforcing whether submitted JavaScript may contain obfuscation. The filtering performed by NOFUS is flexible: we realize that there is no one-size-fits-all approach to this kind of filtering and allow the platform manufacturer to set the rules.

3.2 JavaScript Runtime

JavaScript is a single-threaded language that uses an event-based execution model. At runtime, one event handler is picked up from the queue and is executed to completion. As part of execution, it might create other handlers, generate new code, etc.

Code generation at runtime is fairly common in JavaScript. The most widely used technique is the use of `eval`, but other approaches such as calling `document.write` and `setTimeout` are also widely used to convert runtime strings into invocable code. Note that, while there are frequently better ways to express the same functionality in JavaScript, these techniques are used in completely benign code: `eval` is used for JSON string evaluation; `document.write` is used to inject new frames into the document, etc.

When trying to understand runtime execution of JavaScript code, it is therefore useful to think about the *unfolding tree*, a tree of code contexts that represents code creation (or "unfolding") at runtime. One approach to obfuscation is to hide the intent of code using such unfoldings (based on `eval` or `document.write`). As with other forms of obfuscation, however, the use of these mechanisms does not imply malicious intent. In addition, obfuscation through unfolding is relatively easy to defeat if one has access to the underlying JavaScript runtime and can see the unfolded code appear as it is generated. For the remainder of this paper, we consider the problem of classifying JavaScript fragments irrespective of whether they are folded or unfolded.

3.3 JavaScript Obfuscation Techniques

This section provides a very brief overview of some of the obfuscations techniques used in the wild; a more complete survey can be found in [14].

String obfuscation Much of the difficulty of applying standard anti-virus techniques to JavaScript comes from the fact that any form of static string content is easily obfuscated. Some techniques in this space include

- **% encoding**: using the JavaScript `unescape` function to convert an URL-encoded string to avoid it having to appear as pliantext. A similar tecnique involves using `debase64` to decode base 64-encoded text.
- **Unicode encoding**: using a Unicode representation of a string such as `\u0048\u0065...` to avoid it having to appear as plaintext.
- **String manipulation**: using string concatenation, either in isolation or together with `document.write` or `eval`.
- **Character substitution**: involves running a regular expression replace on a string.

This is by no means an exhaustive list, but it suggests some good building blocks that may be combined to derive more complicated obfuscation strategies such as

```
for(i=0;i<"iDMMNvNSME".length;i++) {
  document.write(
     String.fromCharCode(
        "iDMMNvNSME".charCodeAt(i)\^0x21));
}
```

shown in Howard [14].

Obfuscated Field References In JavaScript, instead of the traditional `o.f` notation used for object access, one can instead resort to a bracket form `o["f"]`. The problem, of course, is that the index expression may be computed and, as such, is subject to the string obfuscation strategies outlined above. So, `document.write`, which might otherwise be suspect may be obfuscated using

```
document["wr"+"i"+"15t355e3".replace(/[0-9]/g, "")]
```

Environment Interactions Within a web browser, JavaScript code runs in the context of a *browser binding*. It is the binding that exposes JavaScript objects such as the `Document` that actually makes JavaScript programs useful. The DOM, in particular, allows several types of obfuscation opportunities.

One approach is to make scripts difficult to reverse engineer by scattering them across the HTML using multiple `<script>` blocks. Another technique involves storing program data (often, the payload) within the DOM and then being subsequently extracted from it. This way, the payload will not be clearly discernable from the code itself.

3.4 Obfuscation and Minimization

There are a number of commercial and free tools available on the web that will obfuscate JavaScript (e.g.,see [9, 15, 20, 23]. A different class of tools is available for the purpose of minimizing and/or compressing JavaScript (e.g., see [7, 11, 12]). There is an important distinction between these two types of tools that is relevant for our classification approach. While minification tools do reduce the readability of JavaScript code, they do not agressively attempt to hide the intent and/or content of the code to an interested reader using the kinds of techniques mentioned above. In this paper, the NOFUS classifier we examine is not trained to recognize minified code as obfuscated. Such training is possible if desired, but we leave that for future work.

3.5 Deploying NOFUS

There are a number of scenarios in which NOFUS could be deployed. First, as mentioned, it could be used as part of an automatic filtering process for determining if JavaScript code submitted to a repository was sufficiently unobscure. After passing the NOFUS test, other kinds of automated static analysis could be applied, including checking for JavaScript subsets, policy violations, etc. [13]. Second, it could be used in a malware context where a file detected as obfuscated by NOFUS is examined with greater scrutiny by more intrusive and expensive static and dynamic analysis tools. Note that labeling JavaScript detected as obfuscated by NOFUS does not necessarily imply that it is malicious, but greater care can be taken with such files once the fast NOFUS filter detects obfuscation.

4. NOFUS Implementation

NOFUS makes use of a statistical classifier to efficiently identify obfuscated JavaScript. Figure 3 illustrates the overall NOFUS architecture. Note that our approach follows our previous work, Zozzle, where we use a similar method to statically classify malware. As a result, we only summarize the approach here and refer readers to futher details in our previous paper [8].

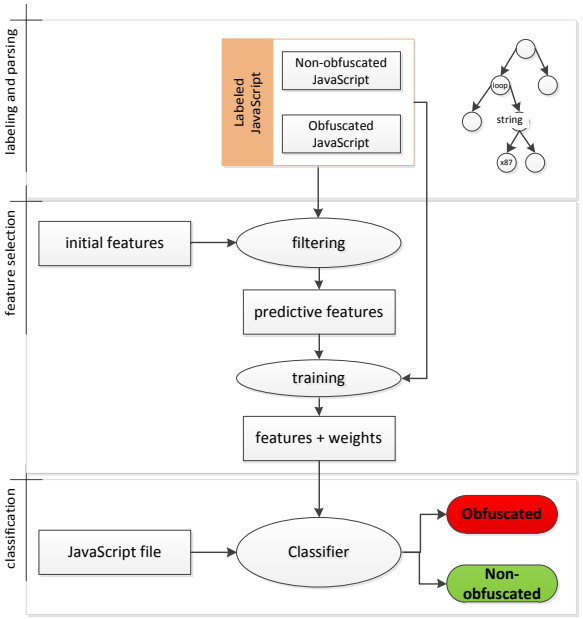

Figure 3: NOFUS architecture.

4.1 Feature Extraction

We start with a set of training data that contains examples of both obfuscated and unobfuscated JavaScript source. Section 5 describes how we obtained hand-labeled files for training and testing purposes. With labeled JavaScript files, we automatically extract features from them that are predictive of obfuscation. For NOFUS, we create features based on the hierarchical structure of the JavaScript abstract syntax tree (AST). Specifically, a feature consists of two parts: a context in which it appears (such as a loop, conditional, `try/catch` block, etc.) and the text (or some substring) of the AST node. In our evaluation, we consider different amounts of context, including none, 1-level (just the innermost surrounding context), and n-level (all enclosing contexts). For a given JavaScript context, we only track whether a feature appears or not, and not the number of occurrences. To efficiently extract features from the AST, we traverse the tree from the root, pushing AST contexts onto a stack as we descend and popping them as we ascend.

Figure 4 illustrates the kinds of features that our classifier uses to determine if a file is obfuscated. It contains some of the operations we already mentioned, such as `document.write` as well as some unexpected values, such peculiar expressions such as `c&0x1f` and `z >>> 5`. Apparently, most unobfuscated files fail to contain these expressions, whereas obfuscated files sometimes will.

Following Zozzle, to limit the possible number of features, we only extract features from specific nodes of the AST: expressions and variable declarations. At each of the

Feature	Probability
typeerror	0.8972
/%/g	0.8972
document.write	0.8885
z >>> 5	0.8291
xxtea$_d$ecrypt	0.8291
utf8to16	0.8291
delta = 0x9e3779b9;	0.8291
c&0x1f	0.8291
yycwy	0.7443

Figure 4: Example features selected by NOFUS.

Collection	Source	Files
Obfuscated (labeled)	Nozzle non-malicious	563
Non-obfuscated (labeled)	Firefox extensions	3,954
JavaScript libraries	Google's CDN	20
Alexa Top 50	alexa.com	7,977
Google Gadgets	Google	1,170
Windows Live Widgets	live.com	2,700
Sideshow Gadgets	Microsoft	4,468
Tool-obfuscated code	Libraries and extensions	22
Malware files	Nozzle malicious	755

Figure 5: Sources of JavaScript code for training and analysis

expression and variable declaration nodes, a new feature record is added to that script's feature set.

If we use the text of every AST expression or variable declaration observed in the training set as a feature for the classifier, it will perform poorly. This is because most of these features are not informative (that is, they are not correlated with either non-obfuscated or obfuscated training set). To improve classifier performance, we instead pre-select features from the training set using the χ^2 statistic to identify those features that are useful for classification. A preselected feature is added to the script's feature set if its text is a substring of the current AST node and the contexts are equal. The method we used to select these features is described in the following section.

4.2 Feature Selection

As illustrated in Figure 3, after creating an initial feature set, NOFUS performs a filtering pass to select those features that are most likely to be most predictive. For this purpose, we used the χ^2 algorithm to test for correlation. We include only those features whose presence is correlated with the categorization of the script (obfuscated or unobfuscated). We selected features with $\chi^2 \geq 10.83$, which corresponds with a 99.9% confidence that the two values (feature presence and script classification) are not independent.

4.3 Classifier Training

Classifying a fragment of JavaScript requires traversing its AST to extract the fragment's features, multiplying the constituent probabilities of each discovered feature (actually implemented by adding log-probabilities), and finally multiplying by the prior probability of the label. Classification can be performed in linear time, parameterized by the size of the code fragment's AST, the number of features being examined, and the number of possible labels.

5. Methods

In this section, we decribe out methods for evaluating NOFUS. Our evaluation is based on cross validation using hand-labeled sets of obfuscated and non-obfuscated files.

5.1 JavaScript Collectons

Figure 5 indicates the size of the our JavaScript collections and where we acquired them. The first two sets of files (Obfuscated and Non-obfuscated) are collections of JavaScript files that are hand labeled. The Obfuscated files are taken from a collection of JavaScript files collected during a deployment of Nozzle [21], which is a dynamic heap-spraying detection system. In the course of our Nozzle deployment, we extracted JavaScript fragments that were being interpreted by the JavaScript runtime and saved them. In post-processing those fragements by hand, we discovered that only some of them contained truly malicious code (e.g., with an identifiable heap spray, shellcode, and browser vulerability), while other fragments were inserted soley for the purpose of obfuscatating the malicious code that was eventually generated. These fragments are not themselves malicious, and we collected such files to form the Obfuscated set. The Non-obfuscated JavaScript files were taken from a collection of FireFox browser extensions (all written in JavaScript). Because they contain hand-written JavaScript that is intended to be reviewed by humans [18], these files contain unobfuscated code by design.

In addition to our testing and training sets, Figure 5 lists collections of JavaScript that we evaluate with NOFUS. Specifically, we have collected JavaScript files from a variety of places, including the Alexa Top 50 websites, all presumably benign; from files detected by Nozzle that have been hand-labeled as malicious; from various JavaScript libraries, such as jQuery; from a collection of JavaScript widgets, etc. We also applied two available JavaScript obfuscators [9, 15] to a collection of unobfuscated files to determine if NOFUS is able to identify obfuscation from specific obfuscators. The intent of our investigation is to understand how many of the files in these collections are considered obfuscated and whether NOFUS does a reasonable job classifying them. One of our contentions is that malicious code is not necessarily obfuscated and we present evidence to support this.

5.2 Cross Validation

To evaluate the NOFUS classifier, we used the following approach. We chose a random 25% of our hand-labeled

datasets to use for training and the remaining 75% for testing. Because the files picked for training are chosen randomly, we repeated our experiments 9 times, picking a different random 25% each time, and averaged our results. Following the methods outlined in our Zozzle paper [8] we extract a set of predictive features automatically from the training set and then train a classifier with that set of features. Using the 25% of the files for training, we constructed a naive Bayesian classifier and then applied the resulting classifier to the remaining 75% of the files.

While there are a number of possible parameters that can be varied in such experiments, for the sake of brevity we focus on specific configurations in this paper. Zozzle experiments [8] show that a training set smaller than 25% can also be effective—we fix the size of the training set at 25%. Likewise, our prior work compares a set of hand-picked features against an automatically generated set of features—here we do not consider how effective a set of hand-picked features might be because our prior work indicates that automatic feature selection is uniformly better than hand-picked features.

6. Experimental Evaluation

In this section, we seek to answer the following questions:

- How effective is our static obfuscation classifier in terms of false positive and false negative results when it is applied to a collection of labeled samples?
- How much obfuscated code does the detector detect when applied to a collection of deployed JavaScript from Alexa top 50 web sites?
- What is the performance of the obfuscation classifier?

6.1 Classifier Effectiveness

Using the methods described in the previous section, we evaluate the effectiveness of the NOFUS classifier. For these evaluations, we vary the number of features included in the classifier (prioritizing those that have the greatest discrimination value) and also consider classifiers that include varying levels of AST hierarchy (flat, 1-level, and n-level). The false negative rate measures the fraction of files that our classifier labels as unobfuscated in the 75% of Obfuscated files withheld for evaluation. Figure 6 shows the false negative rate of different NOFUS classifiers as a function of number of features used and amount of AST hierarchy used. As the figure shows, the false negative rate increases as more features are included in the classifier, reaching at high as 40% for n-level classifiers with 1000 features. This result suggests that including more features leads to greater ambiguity in classifying many of the obfuscated files, perhaps because they also contain features that suggest that they are unobfuscated as well. The figure also shows that greater amounts of AST context typically hurt the false negative rate. One reason for this result is that the presence of context (e.g., that a particular attempt at obfuscation only appears in specific contexts, such as in a loop) may not be predictive. In which case, the hierarchy just dilutes the predictiveness of a particular feature, requiring more features, which we see leads to higher false negative rates.

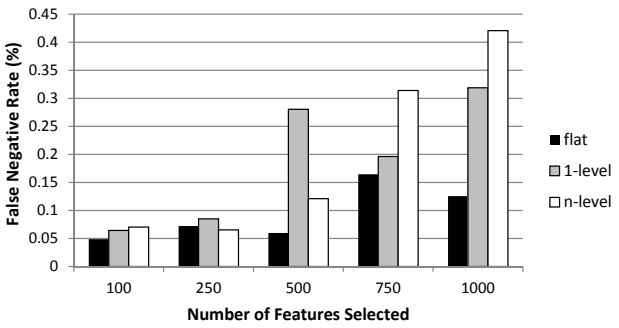

Figure 6: Classification false negative rate as a function of feature set size for flat, 1-level, and n-level classifiers.

The false positive rate measures the fraction of the Non-obfuscated files that our classifier labels as obfuscated. Figure 7 shows the false positive rates of various classifiers. Overall, the false positive rate is very low and trends the opposite of the false negative rate—the more features used the lower the false positive rate. The suggestion here is that with many features, a larger number of features need to be present to classify a file as obfuscated. We note that the precision of our false positive results is limited by the number of files in our Obfuscated test set (422 files). Consequently, there is more variation in our results caused by randomness in the training set selection. While the numbers are sufficiently noisey to deny strong claims about the impact of features or hierarchy, we do observe that the flat classifier has a uniformly low false positive rate and appears effective even with only 100 features. Given that additional features increase the classification time for the rest of this evaluation, we use a NOFUS classifier with a flat hierarchy and 100 features.

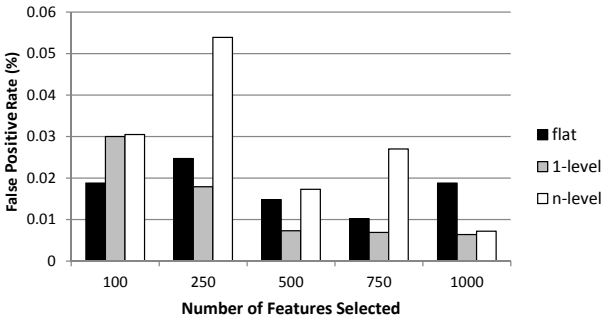

Figure 7: Classification false positive rate as a function of feature set size for flat, 1-level, and n-level classifiers.

Obfuscation Score	JavaScript libraries	Alexa Top 50	Google Gadgets	Live Widgets	Windows Sidebar
0.0 < x < 0.1	20	7974	1170	2700	4468
0.1 < x < 0.2	0	1	0	0	0
0.2 < x < 0.3	0	0	0	0	0
0.3 < x < 0.4	0	0	0	0	0
0.4 < x < 0.5	0	0	0	0	1
0.5 < x < 0.6	0	2	0	0	0
0.6 < x < 0.7	0	0	0	0	0
0.7 < x < 0.8	0	0	0	0	0
0.8 < x < 0.9	0	0	0	0	1
0.9 < x < 1.0	0	1	0	0	21

Figure 8: Classifier results for different JavaScript collections.

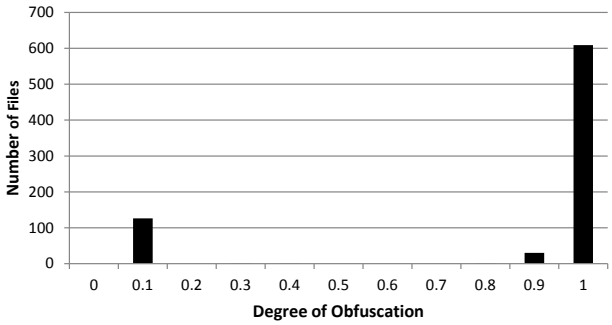

Figure 9: Classifier results for Nozzle-labeled malware.

6.2 Classifying Deployed JavaScript

In this section, we consider the effectiveness of applying our classifier to existing collections of JavaScript code. Our goal is to understand two things: first, what the distribution of obfuscated code is in these collections, and second, is our classifier effective in discriminating obfuscated from unobfuscated code.

Figure 8 shows a table of NOFUS results when it is applied to various collections of JavaScript. Each row represents the number of files in each collection that showed a particular range of obfuscation scores, indicated in Column 1. For example, the figure shows that all 20 JavaScript library files had an obfuscation measure of less than 0.1. Similarly, 21 of the Sidebar files had an obfuscation score above 0.9.

The most striking thing about this table is that very few files appear to be obfuscated in any of the collections. For some of the collections, such as the JavaScript libraries, Google Gadgets, Live Widgets, and Windows Sidebar, this is understandable, because the code is written by humans and intended to be read. Surprisingly, while we thought that the Alexa Top 50 sites would include a mix of obfuscated and unobfuscated code, in fact we detected very little obfuscation. We hand-checked the few occurrences of some degrees of obfuscation in Alexa and found the most obfuscated file was the one we included in the Introduction. We hypothesize that high-traffic benign websites may avoid obfuscation as a general rule so that antivirus detectors don't accidently trigger on them, leading to a reduction in customer satisfaction. Another possibility is that NOFUS is being too careful in avoiding false positives and misclassifying many of the Alexa sites an unobfuscated, leading to false negatives.

The Sidebar widgets collection is the only one with a number of high-obfuscation files. We investigated these in greater detail, and found that most if not all of these files are false positives. The files in question contain code specifically targeted for encryption (e.g., md5, crypto, etc.), codecs, base64 manipulation, etc. The fact that these were detected tells us that our classifier is finding features that are indicative of obfuscation, but it also points out that such approaches are susceptible to false positives.

6.3 JavaScript Obfuscated with Known Tools

We applied NOFUS to a small collection of files explicitly obfuscated with two available obfuscators [9, 15] to determine its effectiveness in classifying obfuscation caused by a specific obfuscator. We used four of the obfuscated files to train NOFUS, and the remaining 18 files as a test suite. Having been trained explicitly with just a few files obfuscated with the particular tools, NOFUS correctly classifies all the other files obfuscated with those tools correctly. When NOFUS is applied to the original, unofuscated source of the same files, it incorrectly classifies only one as obfuscated (1/18 false positive rate).

6.4 Malware versus Obfuscated Code

Figure 9 presents results of NOFUS when applied to files previous hand-categorized as malicious for our Zozzle work [8]. The figure shows a histogram of the degree of obfuscation present in the collection. We see that while many files are tagged as obfuscated by NOFUS, a significant fraction (more than 15%) are not. We examined these files and found that they contain unobfuscated malicious code, typically the result of several levels of code unfolding. We conclude that assuming malicious code always contains attempts at obfuscation can significantly increase the false negative rate of malware detectors.

6.5 Performance

On average, we measure the NOFUS processing time to be approximately 5 megabytes per second (not including JavaScript parsing time), which is faster than the results reported for Zozzle (e.g., 0.4 to 1.6 megabytes per second). This performance improvement is likely due to the fact that are are using fewer features and the flat classified in our studies, whereas the Zozzle results used a 1-level classifier with 500 features.

7. Related Work

Because JavaScript has become a major platform for malicious attacks on browsers and other applications, such as Adobe Reader, there has been a variety of recent techniques

attempting to detect and prevent attacks based on malicious JavaScript (e.g., see [1, 3, 5, 8, 21]). Relatively little of this prior work focuses solely on detecting obfuscation specifically, and none of the prior work distinguishes obfuscation from maliciousness.

7.1 Detecting Obfusction

The most closely related work on obfuscation detection is that of Choi et al. [3], who observe that strings that appear in obfuscated, malicious web pages have qualities that distinguish them from unobfuscated pages. Specifically, they use properties that include the distribution of byte values in strings, the distribution of n-grams in strings, and the overall string length to classify the JavaScript in an web page as malicious or benign. Using 33 samples of malicious JavaScript, they show that a combination of their techniques will classify 30 out of the 33 files as malicious. In contrast, our classifier does not look for specific structures of contents of strings, but instead uses the content and structure of the entire JavaScript AST for classification.

Kim et al. [1] present findings illustrating the kinds of obfuscation techniques being applied to malicious JavaScript, and present similar results to Choi et al., indicating that malicious JavaScript has qualities different from benign JavaScript. Neither Choi or Kim distinguish between benign obfuscated JavaScript and malicious JavaScript, as we do, and our false positive and false negative rates are lower than those reported.

7.2 Building Malware Classifiers

Our previous work, Zozzle [8], uses static AST features and a Bayesian classifier to classify JavaScript malware based on training sets. Because the malware samples used to train Zozzle were collected using the Nozzle heap-spray detector [21], Zozzle is specifically effective at classifying malware with shellcode, heap sprays, and known vulnerabilities. NOFUS takes Zozzle forward by considered the specific problem of classifying obfuscated code and analyzing existing JavaScript collections with our new classifier.

Cova et al. [5] present a comprehensive approach to malware detection that includes efforts to anticipate and undo obfuscation techniques. They include features such as the relationship between string definitions and uses, the number of dynamic code executions (e.g., calls to eval), and the size of dynamically created code to guide classification. Because their work is much broader than our simple goal of obfuscation detection, their results are not directly comparable.

Canali et al. [2] present Prophiler, which also uses a lightweight static filter for malware detection. As with NOFUS, Prophiler uses static features of JavaScript, in addition to other features present in the HTML, to quickly classify a web page a potentially malicious. Unlike NOFUS, they use a collection of 25 hand-picked JavaScript features including things like the number of occurrences of eval, the number of long strings, the number of suspicious object names, etc.

Rieck et al. [22] describe an approach that combines static and dynamic features with a classifier framework based on support vector machines. They pre-process the source code into tokens and pass groups of tokens (so-called Q-grams) to automatically extract Q-grams that are predictive of malicious intent. Unlike NOFUS, their approach combines ad-hoc transformations (e.g., they encode string lengths into variable names) with automatic feature extraction. Further, they do not attempt to separate malicous from obfuscated code or classify whether code is obfuscated or not.

8. Conclusions

Increasingly, JavaScript code is being made available in repositories for widespread use, such as in a gallery of browser extensions. Maintainers of such repositories prefer that this code is unobfuscated so that they can evaluate it either automatically or by inspection.

In this paper, we describe NOFUS, a fully automatic classifier that distinguishs obfuscated and non-obfuscated JavaScript with high precision. Using a collection of examples of both obfuscated and non-obfuscated JavaScript, we train a classifier to distinguish between the two and show that the classifier has both a low false positive rate (about 1%) and low false negative rate (about 5%). We also show that NOFUS can correctly identify benign obfuscated code in collections such as Alexa top 50 websites.

While prior work conflates obfuscation with maliciousness (assuming that detecting obfuscation implies maliciousness), we show that this assumption is flawed. Particularly, assuming all obfuscated code is malicious can lead to increased false positive rates for malware detectors. Likewise, as we show, a significant amount of malware is unobfuscated if it is unfolded in the JavaScript runtime. Thus assuming all malware is obfuscated can increase false negative rates.

References

[1] H.-C. J. Byung-Ik Kim, Chae-Tae Im. Suspicious malicious web site detection with strength analysis of a JavaScript obfuscation. *International Journal of Advanced Science and Technology*, 2011.

[2] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. Technical Report 2010-22, University of California at Santa Barbara, 2010.

[3] Y. Choi, T. Kim, and S. Choi. Automatic detection for JavaScript obfuscation attacks in web pages through string pattern analysis. *Int. Journal of Security and Its Applications*, 2919.

[4] R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for JavaScript. In *Proc. of the Conference on Programming Language Design and Implementation*, June 2009.

[5] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proceedings of the International World Wide Web Conference*, Raleigh, NC, April 2010.

[6] D. Crockford. ADSafe. adsafe.org.

[7] D. Crockford. JSMin: The JavaScript minifier. http://www.crockford.com/javascript/jsmin.html.

[8] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Low-overhead mostly static JavaScript malware detection. Technical Report MSR-TR-2010-156, Microsoft Research, Jan. 2011.

[9] I. CuteSoft Components. Javascript obfuscator. http://javascriptobfuscator.com/, 2010.

[10] B. Feinstein and D. Peck. Caffeine monkey: Automated collection, detection and analysis of malicious JavaScript. http://craigchamberlain.com/library/blackhat-2007/Feinstien_and_Peck/Whitepaper/bh-usa-07-feinstien_and_peck-WP.pdf, 2007.

[11] C. Foster. JSCrunch: JavaScript cruncher. http://www.cfoster.net/jscrunch/.

[12] Google. minify: combines, minifies, and caches JavaScript and CSS files on demand to speed up page loads. http://code.google.com/p/minify.

[13] S. Guarnieri and B. Livshits. Gatekeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *Proc. of the Usenix Security Symp.*, Aug. 2009.

[14] F. Howard. Malware with your Mocha? obfuscation and anti emulation tricks in malicious JavaScript. http://www.sophos.com/security/technical-papers/malware_with_your_mocha.pdf, Sept. 2010.

[15] JavaScript-Source.com. Javascript obfuscator. http://javascript-source.com/, 2009.

[16] Microsoft. Microsoft web sandbox. http://websandbox.org/, 2010.

[17] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja - safe active content in sanitized JavaScript, October 2007. http://google-caja.googlecode.com/files/caja-spec-2007-10-11.pdf.

[18] Mozilla. Add-on developer hub. https://addons.mozilla.org/en-US/developers/docs/policies/reviews, 2011.

[19] Norman ASA. Antivirus and antispyware products. http://www.norman.com/products/antivirus_antispyware.

[20] P. J. O'Neil. Javascript Utility version 2. http://jsutility.pjoneil.net/, 2010.

[21] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the USENIX Security Symposium*, Montreal, Canada, August 2009.

[22] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In C. Gates, M. Franz, and J. P. McDermott, editors, *ACSAC*, pages 31–39. ACM, 2010.

[23] Stunnix. Javascript obfuscator & encoder. http://www.stunnix.com/prod/jo/, 2011.