ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection

Charlie Curtsinger
Univ. of Mass., Amherst

Benjamin Livshits and Benjamin Zorn
Microsoft Research

Christian Seifert
Microsoft

Abstract

JavaScript malware-based attacks account for a large fraction of successful mass-scale exploitation happening today. Attackers like JavaScript-based attacks because they can be mounted against an unsuspecting user visiting a seemingly innocent web page. While several techniques for addressing these types of exploits have been proposed, in-browser adoption has been slow, in part because of the performance overhead these methods incur.

In this paper, we propose ZOZZLE, a low-overhead solution for detecting and preventing JavaScript malware that is fast enough to be deployed in the browser.

Our approach uses Bayesian classification of hierarchical features of the JavaScript abstract syntax tree to identify syntax elements that are highly predictive of malware. Our experimental evaluation shows that ZOZZLE is able to detect JavaScript malware through mostly static code analysis effectively. ZOZZLE has an extremely low false positive rate of 0.0003%, which is less than one in a quarter million. Despite this high accuracy, the ZOZZLE classifier is fast, with a throughput of over one megabyte of JavaScript code per second.

1 Introduction

In the last several years, we have seen mass-scale exploitation of memory-based vulnerabilities migrate towards heap spraying attacks. This is because more traditional vulnerabilities such as stack- and heap-based buffer overruns, while still present, are now often mitigated by compiler techniques such as StackGuard [7] or operating system mechanisms such as NX/DEP and ALSR [12]. While several heap spraying solutions have been proposed [8, 9, 21], arguably, none are lightweight enough to be integrated into a commercial browser.

However, a browser-based detection technique is still attractive for several reasons. Offline scanning is often used in modern browsers to check whether a particular site the user visits is benign and to warn the user otherwise. However, because it takes a while to scan a very large number of URLs that are in the observable web, some URLs will simply be missed by the scan. Offline scanning is also not as effective against transient malware that appears and disappears frequently.

ZOZZLE is a *mostly static* JavaScript malware detector that is fast enough to be used in a browser. While its *analysis* is entirely static, ZOZZLE has a runtime component: to address the issue of JavaScript obfuscation, ZOZZLE is integrated with the browser's JavaScript engine to collect and process JavaScript code that is created at runtime. Note that *fully* static analysis is difficult because JavaScript code obfuscation and runtime code generation are so common in both benign and malicious code.

Challenges: Any technical solution to the problem outlined above requires overcoming the following challenges:

- **performance**: detection is often too slow to be deployed in a mainstream browser;
- **obfuscated malware**: because both benign and malicious JavaScript code is frequently obfuscated, *purely* static detection is generally ineffective;
- **low false positive rates**: given the number of URLs on the web, while false positive rates of 5% are considered acceptable for, say, static analysis tools, rates even 100 times lower are not acceptable for in-browser detection;
- **malware transience**: transient malware compromises the effectiveness of offline-only scanning.

Because it works in a browser, ZOZZLE uses the JavaScript runtime engine to expose attempts to obscure malware via uses of `eval`, `document.write`, etc. by hooking the runtime and analyzing the JavaScript just before it is executed. We pass this *unfolded* JavaScript to a static classifier that is trained using features of the JavaScript

AST (abstract syntax tree). We train the classifier with a collection of labeled malware samples collected with the NOZZLE dynamic heap-spraying detector [21]. Related work [4, 6, 14, 22] also classifies JavaScript malware using a combination of static and dynamic features, but relies on emulation to deobfuscate the code and to observe dynamic features. Because we avoid emulation, our analysis is faster and, as we show, often superior in accuracy.

Contributions: this paper makes these contributions:

- **Mostly static malware detection.** We propose ZOZZLE, a highly precise, lightweight, mostly static JavaScript malware detector. ZOZZLE is based on extensive experience analyzing thousands of real malware sites found while performing dynamic crawling of millions of URLs using the NOZZLE runtime detector.
- **AST-based detection.** We describe an AST-based technique that involves the use of hierarchical (context-sensitive) features for detecting malicious JavaScript code. This context-sensitive approach provides increased precision in comparison to naïve text-based classification.
- **Fast classification.** Because fast scanning is key to in-browser adoption, we present fast multi-feature matching algorithms that scale to hundreds or even thousands of features.
- **Evaluation.** We evaluate ZOZZLE in terms of performance and malware detection rates, both false positives and false negatives. ZOZZLE has an extremely low false positive rate of 0.0003%, which is less than one in a quarter million, comparable to five commericial anti-virus products we tested against. To obtain these numbers, we tested ZOZZLE against a collection of over 1.2 million benign JavaScript samples. Despite this high accuracy, the classifier is very fast, with a throughput at over one megabyte of JavaScript code per second.

Classifier-based tools are susceptible to being circumvented by an attacker who knows the inner workings of the tool and is familiar with the list of features being used, however, our preliminary experience with ZOZZLE suggests that it is capable of detecting many thousands of malicious sites daily in the wild. We consider the issue of evasion in Section 6.

Paper Organization: The rest of the paper is organized as follows. Section 2 gives some background information on JavaScript exploits and their detection and summarizes our experience of performing offline scanning with NOZZLE on a large scale. Section 3 describes the implementation of our analysis. Section 4 describes our experimental methodology. Section 5 describes our experimental evaluation. Section 6 provides a discussion of the limitations and deployment concerns for ZOZZLE. Section 7 discusses related work, and, finally, Section 8 concludes.

```
<html>
  <body>
    <button id="butid" onclick="trigger();"
                          style="display:none"/>
    <script>
    // Shellcode
    var shellcode=unescape('\%u9090\%u9090\%u9090\%u9090...');
    bigblock=unescape('\%u0D0D\%u0D0D');
    headersize=20;
    shellcodesize=headersize+shellcode.length;
    while(bigblock.length<shellcodesize){bigblock+=bigblock;}
    heapshell=bigblock.substring(0,shellcodesize);
    nopsled=bigblock.substring(0,
                      bigblock.length-shellcodesize);
    while(nopsled.length+shellcodesize<0x25000){
      nopsled=nopsled+nopsled+heapshell
    }

    // Spray
    var spray=new Array();
    for(i=0;i<500;i++){spray[i]=nopsled+shellcode;}

    // Trigger
    function trigger(){
      var varbdy = document.createElement('body');
      varbdy.addBehavior('#default#userData');
      document.appendChild(varbdy);
        try {
        for (iter=0; iter<10; iter++) {
            varbdy.setAttribute('s',window);
        } catch(e){ }
        window.status+='';
      }
        document.getElementById('butid').onclick();
   }
    </script>
</body>
</html>
```

Figure 1: Heap spraying attack example.

Appendices are organized as follows. Appendix A discusses some of the hand-analyzed malware samples. Appendix B explores tuning ZOZZLE for better precision. Appendix C shows examples of non-heap spray malware and also anti-virus false positives.

2 Background

This section gives overall background on JavaScript-based malware, focusing specifically on heap spraying attacks.

2.1 JavaScript Malware Background

Figure 1 shows an example of real JavaScript malware that performs a heap spray. Such malware consists of three relatively independent parts. The shellcode is the portion of executable machine code that will be placed on the browser heap when the exploit is executed. It is typical to precede the shellcode with a block of NOP instructions (so-called *NOP sled*). The sled is often quite large compared to the size of the subsequence shellcode, so that a random jump into the process address space is likely to hit the NOP sled and slide down to the start of the shellcode. The next part is the spray, which allocates many copies of the NOP sled/shellcode in the browser heap. In JavaScript, this is easily accomplished using an array of strings. Spraying of this sort can be used to de-

feat address space layout randomization (ASLR) protection in the operating system. The last part of the exploit triggers a vulnerability in the browser; in this case, the vulnerability is a well-known flaw in Internet Explorer 6 that exploits a memory corruption issue with function `addBehavior`.

Note that the example in Figure 1 is entirely unobfuscated, with the attacker not even bothering to rename variables such as `shellcode`, `nopsled`, and `spray` to make the attack easier to spot. In practice, many attacks are obfuscated prior to deployment, either by hand, or using one of many available obfuscation kits [11]. To avoid detection, the primary technique used by obfuscation tools is to use `eval` *unfolding*, i.e. self-generating code that uses the `eval` construct in JavaScript to produce more code to run.

2.2 Characterizing Malicious JavaScript

ZOZZLE training is based on results collected with the NOZZLE heap spraying detector. To gather the data we use to train the ZOZZLE classifier and evaluate it, we employed a web crawler to visit many randomly selected URLs and process them with NOZZLE to detect if malware was present.

Once we determine that JavaScript is malicious, we invested a considerable effort in examining the code by hand and categorizing in various ways. One of the insights we gleaned from this process is that once unfolded, most malware does not have that much variety, following the traditional long tail pattern. We discuss some of the hand-analyzed samples in Appendix A.

Any offline malware detection scheme must deal with the issues of transience and cloaking. Transient malicious URLs go offline or become benign after some period of time, and cloaking is when an attack hides itself from a particular user agent, IP address range, or from users who have visited the page before. While we tried to minimize these effects in practice by scanning from a wider range of IP addresses, in general, these issues are difficult to fully address.

Figure 2 summarizes information about malware transience. To compute the transience of malicious sites, we re-scan the set of URLs detected by Nozzle on the previous day. This procedure is repeated for three weeks (21 days). The set of all discovered malicious URLs were re-scanned on each day of this three week period. This means that only the URLs discovered on day one were re-scanned 21 days later. The URLs discovered on day one happened to have a lower transience rate than other days, so there is a slight upward slope toward the end of the graph.

Any offline scanning technique will have difficulty keeping up with malware exhibiting such a high rate of transience–Nearly 20% of malicious URLs were gone after a single day. We believe that in-browser detection is desirable, in order to be able to detect new malware before it has a chance to affect the user regardless of whether the URL being visited has been scanned before.

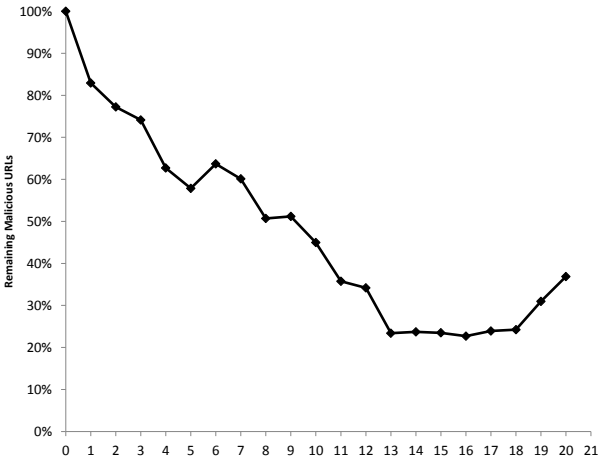


Figure 2: Transience of detected malicious URLs after several days. The number of days is shown of the x axis, the percentage of remaining malware is shown on the y axis.

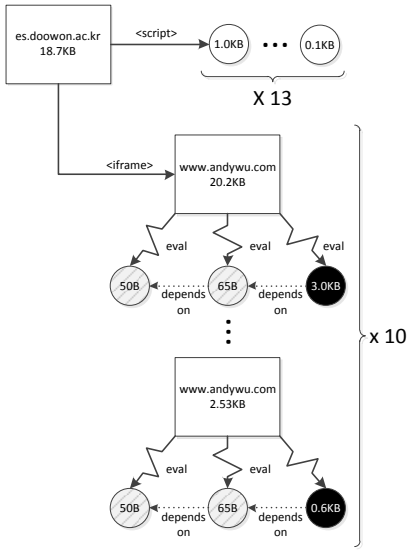


Figure 3: Unfolding tree: an example. Rectangles are documents, and circles are JavaScript contexts. Gray circles are benign, black are malicious, and dashed are "co-conspirators" that participate in deobfuscation. Edges are labeled with the method by which the context or document was reached. The actual page contains 10 different exploits using the same obfuscation.

2.3 Dynamic Malware Structure

One of the core issues that needs to be addressed when talking about JavaScript malware is the issue of *obfusca-*

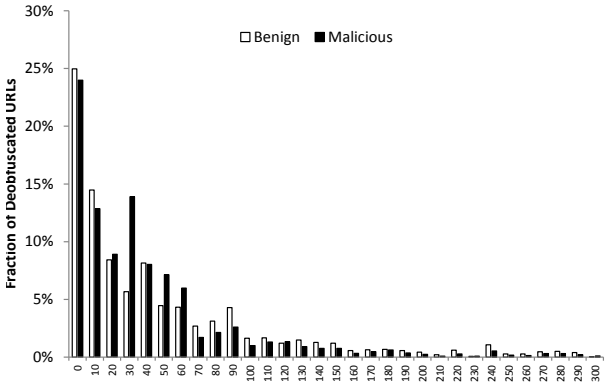


Figure 4: Distribution of context counts for malware and benign code.

tion. In order to avoid detection, malware writers resort to various forms of JavaScript code obfuscation, some of which is done by hand, other with the help of many available obfuscation toolkits [11]. While many approaches to code obfuscation exist, in our experience we see `eval` unfolding as the most commonly used. The idea is to use the `eval` language feature to generate code at runtime in a way that makes the original code difficult to pattern-match. Often, this form of code unfolding is used repeatedly, so that many levels of code are produced before the final, malicious version emerges.

Example 1 Figure 3 illustrates the process of code unfolding using a specific malware sample obtained from a web site `http://es.doowon.ac.kr`. At the time of detection, this malicious URL flagged by NOZZLE contained 10 distinct exploits, which is not uncommon for malware writers, who tend to "over-provision" their exploits: to increase the changes to successful exploitation, they may include multiple exploits within the same page. Each exploit in our example is pulled in with an `<iframe>` tag.

Each of these exploits is packaged in a similar fashion. The leftmost context is the result of an `eval` in the body of the page that defines a function. Another `eval` call from the body of the page uses the newly-defined function to define another new function. Finally, this function and another `eval` call from the body exposes the actual exploit. Surprisingly, this page also pulls in a set of benign contexts, consisting of page trackers, JavaScript frameworks, and site-specific code. □

Note, however, that the presence of `eval` unfolding does *not* provide a reliable indication of malicious intent. There are plenty of perfectly benign pages that also perform some form of code obfuscation, for instance, as a weak form of copy protection to avoid code piracy. Many commonly used JavaScript library frameworks do the same, often to save space through client-side code generation.

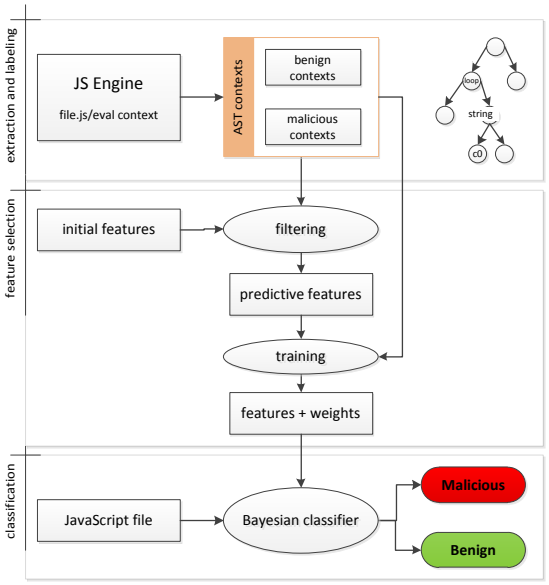


Figure 5: ZOZZLE training illustrated.

We instrumented the ZOZZLE deobfuscator to collect information about which code context leads to other code contexts, allowing us to collect information about the number of code contexts created and the unfolding depth. Figure 4 shows a distributions of JavaScript context counts for benign and malicious URLs. The majority of URLs have only several JavaScript code contexts, however, many can be have 50 or more, created through either `<iframe>` or `<script>` inclusion or `eval` unfolding. Some pages, however, may have as many as 200 code contexts. In other words, a great deal of dynamic unfolding needs to take place before these contexts will "emerge" and will be available for analysis.

It is clear from the graph in Figure 4 that, contrary to what might have been thought, the number of contexts is *not* a good indicator of a malicious site. Context counts were calculated for all malicious URLs from a week of scanning with NOZZLE and a random sample of benign URLs over the same period.

3 Implementation

In this section, we discuss the details of the ZOZZLE implementation.

3.1 Overview

Much of ZOZZLE's design and implementation has in retrospect been informed by our experience with reverse engineering and analyzing real malware found by NOZZLE. Figure 5 illustrates the major parts of the ZOZZLE

architecture. At a high level, the process evolves in three stages: JavaScript context collection and labeling as benign or malicious, feature extraction and training of a naïve Baysian classifier, and finally, applying the classifier to a new JavaScript context to determine if it is benign or malicious. In the following section, we discuss the details of each of these stages in turn.

3.2 Training Data Extraction and Labeling

ZOZZLE makes use of a statistical classifier to efficiently identify malicious JavaScript. The classifier needs training data to accurately classify JavaScript source, and we describe the process we use to get that training data here. We start by augmenting the JavaScript engine in a browser with a "deobfuscator" that extracts and collects individual fragments of JavaScript. As discussed above, exploits are frequently buried under multiple levels of JavaScript `eval`. Unlike Nozzle, which observes the behavior of running JavaScript code, ZOZZLE must be run on an unobfuscated exploit to reliably detect malicious code.

While detection on obfuscated code may be possible, examining a fully unpacked exploit is most likely to result in accurate detection. Rather than attempt to decipher obfuscation techniques, we leverage the simple fact that an exploit must unpack itself to run.

Our experiments presented in this paper involved instrumenting the Internet Explorer browser, but we could have used a different browser such as Firefox or Chrome instead. Using the Detours binary instrumentation library [13], we were able to intercept calls to the `Compile` function in the JavaScript engine located in the `jscript.dll` library. This function is invoked when `eval` is called and whenever new code is included with an `<iframe>` or `<script>` tag. This allows us to observe JavaScript code at each level of its unpacking just before it is executed by the engine. We refer to each piece of JavaScript code passed to the `Compile` function as a *code context*. For purposes of evaluation, we write out each context to disk for post-processing. In a browser-based implementation, context assessment would happen on the fly.

3.3 Feature Extraction

Once we have labeled JavaScript contexts, we need to extract features from them that are predictive of malicious or benign intent. For ZOZZLE, 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. 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.

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 expression and variable declarations 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 benign or malicious 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 pre-selected 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.

3.4 Feature Selection

As illustrated in Figure 5, after creating an initial feature set, ZOZZLE performs a filtering pass to select those features that are 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 (benign or malicious). The χ^2 test (for one degree of freedom) is described below:

$$A = \text{malicious contexts with feature}$$

$$B = \text{benign contexts with feature}$$

$$C = \text{malicious contexts without feature}$$

$$D = \text{benign contexts without feature}$$

$$\chi^2 = \frac{(A*D - C*B)^2}{(A+C)*(B+D)*(A+B)*(C+D)}$$

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.

3.5 Classifier Training

ZOZZLE uses a naïve Bayesian classifier, one of the simplest statistical classifiers available. When using naïve Bayes, all features are assumed to be statistically independent. While this assumption is likely incorrect, the independence assumption has yielded good results in the

past. Because of its simplicity, this classifier is efficient to train and run.

The probability assigned to label L_i for code fragment containing features $F_1, \ldots, F_n$ may be computed using Bayes rule as follows:

$$P(L_i|F_1, \ldots, F_n) = \frac{P(L_i)P(F_1, \ldots, F_n|L_i)}{P(F_1, \ldots, F_n)}$$

Because the denominator is constant regardless of L_i we ignore it for the remainder of the derivation. Leaving out the denominator and repeatedly applying the rule of conditional probability, we rewrite this as:

$$P(L_i|F_1, \ldots, F_n) = P(L_i) \prod_{k=1}^{n} P(F_k|F_1, \ldots, F_{k-1}, L_i)$$

Given that features are assumed to be conditionally independent, we can simplify this to:

$$P(L_i|F_1, \ldots, F_n) = P(L_i) \prod_{k=1}^{n} P(F_k|L_i)$$

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. It is clear from the definition that classification may 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. The processes of collecting and hand-categorizing JavaScript samples and training the ZOZZLE classifier are detailed in Section 4.

3.6 Fast Pattern Matching

An AST node contains a feature if the feature's text is a substring of the AST node. With a naïve approach, each feature must be matched independently against the node text. To improve performance, we construct a state machine for each context that reduces the number of character comparisons required. There is a state for each unique character occurring at each position in the features for a given context.

A pseudocode for the fast matching algorithm is shown in Figure 7. State transitions are selected based on the next character in the node text. Every state has a bit mask with bits corresponding to features. The bits are set only for those features that have the state's incoming character at that position. At the beginning of the matching, a bitmap is set to all ones. This mask is ANDed with the mask at each state visited during matching. At the end of matching, the bit mask contains the set of features present in the node. This process is repeated for each position in the node's text, as features need not match at the start of the node.

Example 2 An example of a state machine used for fast pattern matching is shown in Figure 6. This string matching state machine can identify three patterns: `alert`, `append`, and `insert`. Assume the matcher is running on input text `appert`. During execution, a bit array of size three, called the matched list, is kept to indicate the patterns that have been matched up to this point in the input. This bit array starts with all bits set. From the leftmost state we follow the edge labeled with the input's first character, in this case an `a`.

The match list is bitwise-anded with this new state's bit mask of `110`. This process is repeated for the input characters `p`, `p`, `e`. At this point, the match list contains `010` and the remaining input characters are `r`, `t`, and `null` (also notated as `\0`). Even though a path to an end state exists with edges for the remaining input characters, no patterns will be matched. The next character consumed, an `r`, takes the matcher to a state with mask `001` and match list of `010`. Once the match list is masked for this state, no patterns can possibly be matched. For efficiency, the matcher terminates at this point and returns the empty match list.

The maximum number of comparisons required to match an arbitrary input with this matcher is 17, versus 20 for naïve matching (including null characters at the ends of strings). The worst-case number of comparisons performed by the matcher is the total number of distinct edge inputs at each input position. The sample matcher has 19 edges, but at input position 3 two edges consume the same character ('e'), and at input position 6 two edges consume the null character. In practice, we find that the number of comparisons is reduced significantly more than for this sample, due to the large number of features because of the pigeonhole principle. □

For a classifier using 100 features, a single position in the input text would require 100 character comparisons with naïve matching. Using the state machine approach, there can be no more than 52 comparisons at each string position (36 alphanumeric characters and 16 punctuation symbols), giving a reduction of nearly 50%. In practice there are even more features, and input positions do not require matching against every possible input character.

Figure 8 clearly shows the benefit of fast pattern matching over a naïve matching algorithm. The graph shows the average number of character comparisons performed per-feature using both our scheme and a naive approach that searches an AST node's text for each pattern individually. As can be seen from the figure, the fast matching approach has far fewer comparisons, de-

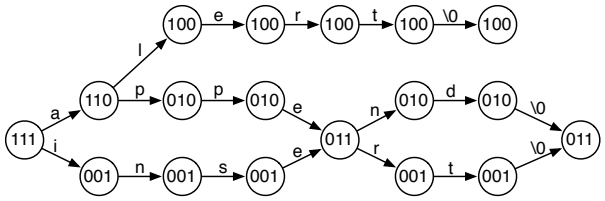


Figure 6: Fast feature matching illustrated.

$matchList \leftarrow \langle 1, 1, \ldots, 1 \rangle$
$state \leftarrow 0$
for all c in $input$ do
  $state \leftarrow matcher.getNextState(state, c)$
  $matchList \leftarrow matchList \wedge matcher.getMask(state)$
  if $matchList\langle 0, 0, \ldots, 0 \rangle$ then
    return $matchList$
  end if
end for
return $matchList$

Figure 7: Fast matching algorithm.

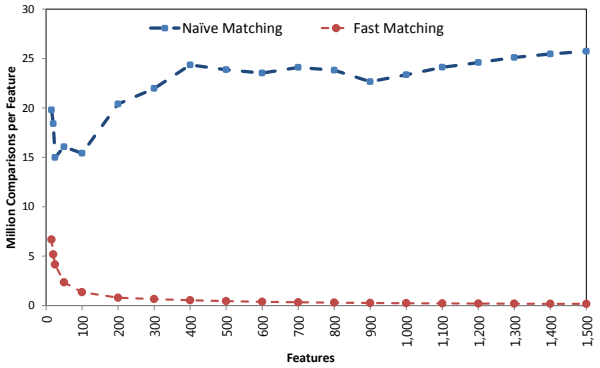


Figure 8: Comparisons required per-feature with naïve vs. fast pattern matching. The number of features is shown on the x axis.

creasing asymptotically as the number of features approaches 1,500.

3.7 Future Improvements

In this section, we describe additional algorithmic improvements not present in our initial implementation.

3.7.1 Automatic Malware Clustering

Using the same features extracted for classification, it is possible to automatically cluster attacks into groups. There are two possible approaches that exist in this space: supervised and unsupervised clustering.

Supervised clustering would consist of hand-categorizing attacks, which has actually already been done for about 1,000 malicious contexts, and assigning new scripts to one of these groups. Unsupervised clustering would not require the initial sorting effort, and is more likely to successfully identify new, common attacks. It is likely that feature selection would be an ongoing process; selected features should discriminate between different clusters, and these clusters will likely change over time.

3.7.2 Substring Feature Selection

For the current version of ZOZZLE, automatic feature selection only considers the entire text of an AST node as a potential feature. While simply taking all possible substrings of this and treating those as possible features as well may seem reasonable, the end result is a classifier with many more features and little (if any) improvement in classification accuracy.

An alternative approach would be to treat certain types of AST nodes as "divisible" when collecting candidate features. If the entire node text is not a good discriminative feature, its component substrings can be selected as candidate features. This avoids introducing substring features when the full text is sufficiently informative, but allows for simple patterns to be extracted from longer text (such as `%u` or `%u0c0c`) when they are more informative than the full string. Not all AST nodes are suitable for subdivision, however. Fragments of identifiers don't necessarily make sense, but string constants and numbers could still be meaningful when split apart.

3.7.3 Feature Flow

At the moment, features are extracted only from the text of the AST nodes in a given context. This works well for whole-script classification, but has yielded more limited results for fine-grained classification (that is, to identify that a specific part of the script is malicious). To prevent a particular feature from appearing in a particularly informative context (such as `COMMENT` appearing inside a loop, a component the Aurora exploit [19]) an attacker can simply assign this string to a variable outside the loop and reference the variable within the loop. The idea behind feature flow is to keep a simple lookup table for identifiers, where both the identifier name *and* its value are used to extract features from an AST node.

By ignoring scoping rules and loops, we can get a reasonable approximation of the features present in both the identifiers and values within a given context with low overhead. This could be taken one step further by emulating simple operations on values. For example, if two identifiers set to strings are added, the values of these strings could be concatenated and then searched for features. This would prevent attackers from hiding common shellcode patterns using concatenation.

4 Experimental Methodology

In order to train and evaluate ZOZZLE, we created a collection of malicious and benign JavaScript samples to use as training data and for evaluation.

Gathering Malicious Samples: To gather the results for Section 5, we first dynamically scanned URLs with a browser running both NOZZLE and the ZOZZLE JavaScript deobfuscator. In this configuration, when NOZZLE detects a heap spraying exploit, we record the URL and save to disk *all* JavaScript contexts seen by the deobfuscator. All recorded JavaScript contexts are then hand-examined to identify those that contain any malware elements (shellcode, vulnerability, or heap-spray).

Malicious contexts can be sorted efficiently by first grouping by their md5 hash value. This dramatically reduces the required effort because of the lack of exploit diversity explained first in Section 2 and relatively few identifier-renaming schemes being employed by attackers. For exploits that do appear with identifier names changed, there are still usually some identifiers left unchanged (often part of the standard JavaScript API) which can be identified using the `grep` utility. Finally, hand-examination is used to handle the few remaining unsorted exploits. Using a combination of these techniques, 919 deobfuscated malicious contexts were identified and sorted in several hours.

Gathering Benign Samples: To create a set of benign JavaScript contexts, we extracted JavaScript from the `Alexa.com` top 50 URLs using the ZOZZLE deobfuscator. The 7,976 contexts gathered from these sites were used as our benign dataset.

Feature Selection: To evaluate ZOZZLE, we partition our malicious and benign datasets into training and evaluation data and train a classifier. We then apply this classifier to the withheld samples and compute the false positive and negative rates. To train a classifier with ZOZZLE, we first need a define a set of features from the code. These features can be hand-picked, or automatically selected (as described in Section 3) using the training examples. In our evaluation, we compare the performance of classifiers built using hand-picked and automatically selected features.

The 89 hand-picked features were selected based on experience and intuition with many pieces of malware detected by NOZZLE and involved collecting particularly "memorable" features frequently repeated in malware samples.

Feature
`try` : `unescape`
`loop` : `spray`
`loop` : `payload`
`function` : `addbehavior`
`string` : `0c`

Figure 9: Examples of hand-picked features used in our experiments.

Feature	Present	M : B
`function` : `anonymous`	✓	1 : 4609
`try` : `newactivexobject("pdf.pdfctrl")`	✓	1309 : 1
`loop` : `scode`	✓	1211 : 1
`function` : `$(this)`	✓	1 : 1111
`if` : `"shel" + "l.ap" + "pl" + "icati" + "on"`	✓	997 : 1
`string` : `%u0c0c%u0c0c`	✓	993 : 1
`loop` : `shellcode`	✓	895 : 1
`function` : `collectgarbage()`	✓	175 : 1
`string` : `#default#userdata`	✓	10 : 1
`string` : `%u`	✗	1 : 6

Figure 10: Sample of automatically selected features and their discriminating power as a ratio of likelihood to appear in a malicious or benign context.

Automatically selecting features typically yields many more features as well as some features that are biased toward benign JavaScript code, unlike hand-picked features that are all characteristic of malicious JavaScript code. Examples of some of the hand-picked features used are presented in Figure 9.

For comparison purposes, samples of the automatically extracted features, including a measure of their discriminating power, are shown in Figure 10. The middle column shows whether it is the presence of the feature (✓) or the absence of it (✗) that we are matching on. The last column shows the number of malicious (M) and benign (B) contexts in which they appear in our training.

In addition to the feature selection methods, we also varied the types of features used by the classifier. Because each token in the Abstract Syntax Tree (AST) exists in the context of a tree, we can include varying parts of that AST context as part of the feature. Flat features are simply text from the JavaScript code that is matched without any associated AST context. We should emphasize that flat features are what are typically used in various *text* classification schemes. What distinguishes our work is that, through the use of *hierarchical features*, we are taking advantage of the contextual information given by the code structure to get better precision.

Hierarchical features, either 1- or n-level, contain a certain amount of AST context information. For example, 1-level features record whether they appear within a loop, function, conditional, `try`/`catch` block, etc. Intuitively, a variable called `shellcode` declared or used right after the beginning of a function is perhaps less indicative of malicious intent than a variable called `shellcode` that is used with a loop, as is common in the case of a spray. For n-level features, we record the entire stack of AST contexts such as

$$\langle \text{a loop, within a conditional, within a function, } \ldots \rangle$$

Features	Hand-Picked	Automatic	Features
flat	95.45%	99.48%	948
1-level	98.51%	99.20%	1,589
n-level	96.65%	99.01%	2,187

Figure 11: Classifier accuracy for hand-picked and automatically selected features.

Features	Hand-Picked		Automatic	
	False Pos.	False Neg.	False Pos.	False Neg.
flat	4.56%	4.51%	0.01%	5.84%
1-level	1.52%	1.26%	0.00%	9.20%
n-level	3.18%	5.14%	0.02%	11.08%

Figure 12: False positives and false negatives for flat and hierarchical features using hand-picked and automatically selected features.

The depth of the AST context presents a tradeoff between accuracy and performance, as well as between false positives and false negatives. We explore these tradeoffs in detail in Section 5.

5 Evaluation

In this section, we evaluate the effectiveness of ZOZZLE using the benign and malicious JavaScript samples described in Section 4. To obtain the experimental results presented in this section, we used an HP xw4600 workstation (Intel Core2 Duo E8500 3.16 Ghz, dual processor, 4 Gigabytes of memory), running Windows 7 64-bit Enterprise.

5.1 False Positives and False Negatives

Accuracy: Figure 11 shows the overall classification accuracy of ZOZZLE when evaluated using our malicious and benign JavaScript samples[1]. The accuracy is measured as the number of successful classifications divided by total number of samples. In this case, because we have many more benign samples than malicious samples, the overall accuracy is heavily weighted by the effectiveness of correctly classifying benign samples.

In the figure, the results are sub-divided first by whether the features are selected by hand or using the automatic technique described in Section 3, and then subdivided by the amount of context used in the classifier (flat, 1-level, and n-level).

[1]Unless otherwise stated, for these results 25% of the samples were used for classifier training and the remaining files were used for testing. Each experiment was repeated five times on a different randomly-selected 25% of hand-sorted data.

	ZOZZLE	AV1	AV2	AV3	AV4	AV5
Samples	1,275,033			1,275,078		
True pos.	5	3	0	3	1	3
False pos.	4	2	5	5	4	3
FP rate	**3.1E-6**	**1.6E-6**	**3.9E-6**	**2.9E-6**	**3.1E-6**	**2.4E-6**

Figure 13: False positive rate comparison.

The table shows that overall, automatic feature selection significantly outperforms hand-picked feature selection, with an overall accuracy above 99%. Second, we see that while some context helps the accuracy of the hand-picked features, overall, context has little impact on the accuracy of automatically selected features. We also see in the fourth column the number of features that were selected in the automatic feature selection. As expected, the number of features selected with the n-level classifier is significantly larger than the other approaches.

Hand-picked vs. Automatic: Figure 12 expands on the above results by showing the false positive and false negative rates for the different feature selection methods and levels of context. The rates are computed as a fraction of malicious and benign samples, respectively. We see from the figure that the false positive rate for all configurations of the hand-picked features is relatively high (1.5-4.5%), whereas the false positive rate for the automatically selected features is nearly zero. The best case, using automatic feature selection and 1-level of context, has no false positives in any of the randomly-selected training and evaluation subsets. The false negative rate for all the configurations is relatively high, ranging from 1–11% overall. While this suggests that some malicious contexts are not being classified correctly, for most purposes, having high overall accuracy and low false positive rate are the most important attributes of a malware classifier.

Best classifier: In contrast to the lower false positive rates, the false negative rates of the automatically selected features are higher than they are for the hand-picked features. The insight we have is that the automatic feature selection selects many more features, which improves the sensitivity in terms of false positive rate, but at the same time reduces the false negative effectiveness because extra benign features can sometimes mask malicious intent. We see that trend manifest itself among the alternative amounts of context in the automatically selected features. The n-level classifier has more features and a higher false negative rate than the flat or 1-level classifiers. Since we want to achieve a very low false positive rate with a moderate false negative rate, and the 1-level classifier provided the best false positive rate in these experiments, in the remainder of this section, we consider the effectiveness of the 1-level classifier in more detail.

ZOZZLE	JSAND	AV1	AV2	AV3	AV4	AV5
9%	15%	24%	28%	34%	83%	42%

Figure 14: False negative rate comparison.

5.2 Comparison with AV & Other Techniques

Previous analysis has been performed on a relatively small set of benign files. As a result, our 1-level classifier does not produce any false alarms on about 8,000 benign samples, but using a set of this size limits the precision of our evaluation. To fully understand the false positive rate of ZOZZLE, we have obtained a large collection of over 1.2 million benign JavaScript contexts taken from manually white-listed web sites.

Investigating false positives further: Figure 13 shows the results of running both ZOZZLE and five state-of-the-art anti-virus products on the large benign data set. Out of the 1.2 million files, only 4 were incorrectly marked malicious by ZOZZLE. This is fewer than one in a quarter million false alarms. The four false positives flagged by ZOZZLE fell into two distinct cases and both cases were essentially a single large JSON-like data structure that included many instances of encoded binary strings. Adding a specialized JSON data recognizer to ZOZZLE could eliminate these false alarms.

Even though anti-virus products attempt to be extremely careful about false positives, in our run, the five anti-virus engines produced 29 alerts when applied to 1,275,078 JavaScript samples.

Our of these, over half, 19 alerts turn out to be false positives. We investigated these further and found several reasons for these errors. The first is assuming that some `document.write` of an unescaped string could be malicious when they in fact were not. The second reason is flagging *unpackers*, i.e. pieces of code that convert a string into another one through character code translation. Clearly, these unpackers *alone* are not malicious. We show examples of these mistakes in Appendix B. The third reason is overly aggressively flagging phishing sites that insert links into the current page; this is because the anti-virus is unable to distinguish between them and malware. The figure also shows cases where we found true malware in the large data set (listed as true positives), despite the fact that the web sites that the JavaScript was taken from were white-listed. We see that ZOZZLE was also better at finding true positives than the anti-virus detectors, finding a total of five out of the 1.2 million samples. We also note that the number of samples used in the anti-virus and ZOZZLE results in this table are slightly different due to the fact that on some of the samples either the anti-virus or ZOZZLE aborts due to ill-formed JavaScript syntax and those samples are not included in the total.

In summary, ZOZZLE has a false positive rate of 0.0003%, which is comparable to the five anti-virus tools in all cases and is better than some of them.

Investigating false negatives further: Figure 14 shows a comparison of ZOZZLE and the five anti-virus engines discussed above. We fed the anti-virus engines the 919 hand-labeled malware samples used in the previous evaluation of ZOZZLE.[2] Additionally, we include JSAND [6], a recently published malware detector that has a public web interface for malware upload and detection. In the case of JSAND, we only used a small random sample of 20 malicious files due to the difficulty of automating the upload process, apparent rate limiting, and the latency of JSAND evaluation. The figure demonstrates that all of the other products have a higher false negative rate compared to ZOZZLE. JSAND is the closest, producing a false negative rate of 15%. We feel that these high false negative rates for the anti-virus products are likely caused by the tendency of such products to be conservative and trade low false positives for higher false negatives. This experiment illustrates the difficulty that traditional anti-virus techniques have classifying JavaScript, where self-generated code is commonplace. We feel that ZOZZLE excels in both dimensions.

5.3 Classifier Performance

Figure 15 shows the classification time as a function of the size of the file, ranging up to 10 KB. We used automatic feature selection, a 1-level classifier trained on .25 of the hand-sorted dataset with no hard limit on feature counts to obtain this chart. This evaluation was performed on a classifier with over 4,000 features, and represents the worst case performance for classification. We see that for a majority of files, classification can be performed in under 4 ms. Moreover, many contexts are in fact `eval` contexts, which are generally smaller than JavaScript files downloaded from the network. In the case of `eval` contexts such as that, the classification overhead is usually 1 ms and below.

Figure 16 displays the overhead as a function of the number of classification features we used and compares it to the average parse time of .86 ms. Despite the fast feature matching algorithm presented in Section 3, having more features to match against is still quite costly. As a result, we see the average classification time grow significantly, albeit linearly, from about 1.6 ms for 30 features to over 7 ms for about 1,300 features. While these numbers are from our unoptimized implementation, we believe that ZOZZLE's static detector has a lot of potential for fast on-the-fly malware identification.

[2]The ZOZZLE false negative rate listed in Figure 14 is taken on our cross-validation experiment in Figure 12.

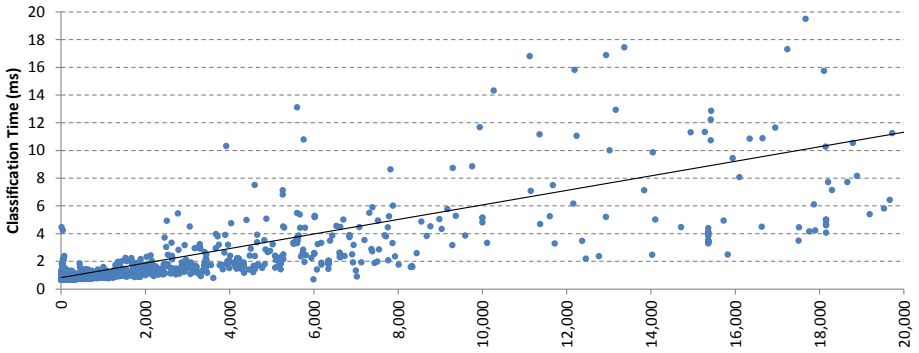


Figure 15: Classification time as a function of JavaScript file size. File size in bytes is shown on the x axis and the classification time in ms is shown on the y axis.

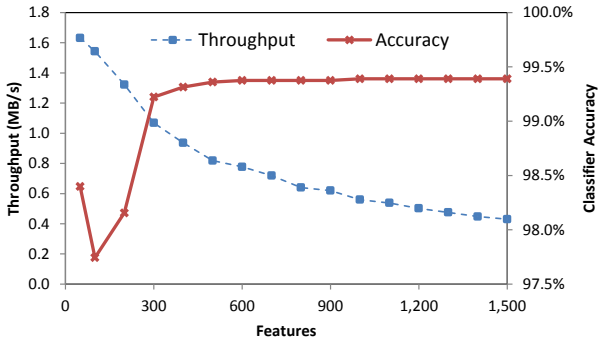


Figure 16: Classifier throughput and accuracy as a function of the number of features, using 1-level classification with .25 of the training set size.

6 Discussion

Caveats and limitations: All classifier-based malware detection tools will fail to detect some attacks, such as exploits that do not contain any of the features present in the training examples. More importantly, attackers who have a copy of ZOZZLE as an oracle can devise variants of malware that are not detected by it. For example, they might rename variables, obscure strings by encoding them or breaking them into pieces, or substitute different APIs that accomplish the same task.

Evasion is made somewhat more difficult because any exploit that uses a known CVE must eventually make the necessary JavaScript runtime calls (e.g., detecting or loading a plugin) to trigger the exploit. If ZOZZLE is able to statically detect such calls, it will detect the attempted exploit. To avoid such detection, an attacker might change the context in which these calls appear by creating local variables that reference the desired runtime function, an approach already employed by some exploits we have collected.

In the future, for ZOZZLE to continue to be effective, it has to be adaptive against attempts to avoid detection. This adaptation takes two forms: improving its ability to reason about the malware, and adapting the feature set used to detect malware as it evolves. To improve ZOZZLE's detection capability, it needs to incorporate more semantic information about the JavaScript it analyzes. For example, as described in Section 3, feature flow could help ZOZZLE identify attempts to obfuscate the use of APIs necessary for malware to be successful. Adapting ZOZZLE's feature set requires continuous retraining based on collecting malware samples detected by deploying other detectors such as NOZZLE. With such adaptation, ZOZZLE would dramatically reduce the effectiveness of the copy-and-pasted attacks that make up the majority of JavaScript malware today. In combination with complementary detection techniques, such as NOZZLE, an updated feature set can be generated frequently with no human intervention.

Just as with anti-virus, we believe that ZOZZLE is one of several measures that can be used as part of a defense-in-depth strategy. Moreover, our experience suggests that in many cases attackers are slow to adapt to the changing landscape. Despite the wide availability of obfuscation tools, in our NOZZLE detection experiments we still find many sites not using any form of obfuscation at all. We also see little diversity in the exploits collected. For example, the top five malicious scripts account for 75% of the malware detected.

Deployment: The most attractive deployment strategy for ZOZZLE is in-browser deployment. ZOZZLE has been designed to require only occasional offline re-training so that classifier updates can be shipped off to the browser every several days or weeks. Figure 17 shows a proposed workflow for ZOZZLE in-browser deployment.

The *code* of the in-browser detector does not need to change, only the list of features and weights needs to be sent, similarly to updating signatures in an anti-virus product. Note that our detector is designed in a way that can be tightly integrated into the JavaScript parser, mak-

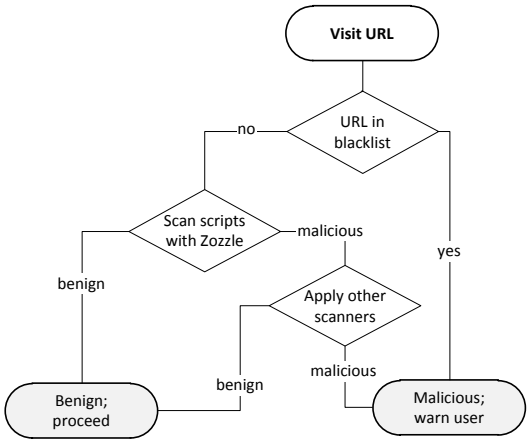


Figure 17: In-browser ZOZZLE deployment: workflow.

ing malware "scoring" part of the overall parsing process; the only thing that needs to be maintained as the parse tree (AST) is being constructed is the set of matching features. This, we believe, will make the incremental overhead of ZOZZLE processing even lower than it is now.

Another way to deploy ZOZZLE is as a filter for a more heavy-weight technique such as NOZZLE or some form of control- or dataflow integrity [1, 5]. As such, the *expected* end-user overhead will be very low, because both the detection rate of ZOZZLE and the rate of false positives is very low; we assume if an attack is prevented, the user will not object to additional overhead in that case.

Finally, ZOZZLE is suitable for offline scanning, either in the case of dynamic web crawling using a web browser, or in the context or purely static scanning that exposes some part of the JavaScript code to the scanner.

7 Related Work

Several recent papers focusing on static detection techniques for malware, specifically implemented in JavaScript. None of the existing techniques propose integrating malware classification with JavaScript execution in the context of a browser, as ZOZZLE does.

7.1 Closely-related Malware Detection Work

A quantitative comparison with closely related techniques is presented in Figure 18. It shows that ZOZZLE is heavily optimized for an extremely low rate of false positives — about one in quarter million — with the closest second being CUJO [22] with six times as many false positives.

ZOZZLE is generally faster than other tools, since the only runtime activity it performs is capturing JavaScript code. In its purely static mode, Cujo is also potentially quite fast, with running times ranging from .01 to 10 ms per URL, however, our rates are not directly comparable because URLs and code contexts are not one-to-one.

Project	Citation	FP rate	FN rate	Static	Dynamic
ZOZZLE		3.1E-6	9.2E-2	✓	-[3]
JSAND	[6]	1.3E-5	2E-3	✓	✓
Prophiler	[4]	9.8E-2	7.7E-3	✓	✓
CUJO	[22]	2.0E-5	5.6E-2	✓	✓

Figure 18: Quantitative comparison to closely related work.

Canali *et al.* [4] present Prophiler, a lightweight static filter for malware. It combines HTML-, JavaScript-, and URL-based features into one classifier that quickly filters non-malicious pages so that malicious pages can be examined more extensively. While their approach has elements in common with ZOZZLE, there are also differences. First, ZOZZLE focuses on classifying pages based on unobfuscated JavaScript code by hooking into the JavaScript engine entry point, whereas Prophiler extracts its features from the *obfuscated* code. Second, ZOZZLE automatically extracts *hierarchical* features from the AST, whereas Prophiler relies on a variety of statistical and lexical hand-picked features present in the HTML and JavaScript. Third, the emphasis of ZOZZLE is on very low false positive rates, whereas Prophiler, because it is intended as a fast filter, allows higher false positive rates in order to reduce the false negative rate.

Rieck *et al.* [22] describe Cujo, an system that combines static and dynamic features in a classifier framework based on support vector machines. They preprocess the source code into tokens and pass groups of tokens (Q-grams) to automatically extract Q-grams that are predictive of malicious intent. Unlike ZOZZLE, Cujo is proxy-based and uses JavaScript emulation instead of hooking into the JavaScript runtime in a browser. This emulation adds runtime overhead, but allows Cujo to use static as well as dynamic Q-grams in their classification. ZOZZLE differs from Cujo in that it uses the existing JavaScript runtime engine to unfold JavaScript contexts without requiring emulation reducing the overhead.

Similarly, Cova *et al.* present a system JSAND that conducts classification based on static and dynamic features [6]. In JSAND, potentially malicious JavaScript is emulated to determine runtime characteristics around deobfuscation, environment preparation, and exploitation, such as the number of bytes allocated through string operations. These features are trained and evaluated with known good and bad URLs. Like Cujo, JSAND uses emulation to combine a collection of static and dynamic features in their classification, as compared to ZOZZLE,

[3] The only part of ZOZZLE that requires dynamic intervention is unfolding.

which extracts only static features automatically. Also, because ZOZZLE leverages the existing JavaScript engine unfolding process, JSAND performance is significantly slower than ZOZZLE.

7.2 Other Projects

Karanth *et al.* identify malicious JavaScript using a classifier based on hand-picked features present in the code [14]. Like us, they use known malicious and benign JavaScript files and train a classifier based on features present. They show that their technique can detect malicious JavaScript with high accuracy and they were able to detect a previously unknown zero-day vulnerability. Unlike our work, they do not integrate their classifier into the JavaScript engine, and so do not see the unfolded JavaScript as we do.

High-interaction client honeypots have been at the forefront of research on drive-by-download attacks. Since they were first introduced in 2005, various studies have been published [15, 20, 25, 30–32]. High-interaction client honeypots drive a vulnerable browser to interact with potentially malicious web page and monitor the system for unauthorized state changes, such as new processes being created. The detection of drive-by-download attacks can also occur through the analysis of the content retrieved from the web server. When captured at the network layer or through a static crawler, the content of malicious web pages is usually highly obfuscated opening the door to static feature based exploit detection [10, 20, 24, 27, 28]. While these approaches, among others, consider static JavaScript features, ZOZZLE is the first to utilize hierarchical features extracted from ASTs.

Besides static features focusing on HTML and JavaScript, shellcode injection exploits also offer points for detection. Existing techniques such as Snort [23] use pattern matching to identify attacks in a database. Polymorphic attacks that vary shellcode on each exploit attempt can avoid pattern-based detection unless improbable properties of shellcode are used to detect such attacks, as in Polygraph [17]. Like ZOZZLE, Polygraph utilizes a naïve bayes classifier, but only applies it to the detection of shellcode.

Abstract Payload Execution (APE) by Toth and Kruegel [29], STRIDE by Akritidis *et al.* [2, 18], and NOZZLE by Ratanaworabhan, Livshits and Zorn [21] all focus on analysis of the shellcode and NOP sled used by a heap spraying attack. Such techniques can detect heap sprays with low false positive rates, but incur higher runtime overhead than is acceptable for always-on deployment in a browser (10-15% is farily common).

Dynamic features have been the focus of several groups. Nazario, Buescher, and Song propose systems that detect attacks on scriptable ActiveX components [3, 16, 26]. They capture JavaScript interactions and use vulnerability specific signatures to detect attacks. This method is effective in detecting attacks due to the relative homogeneous characteristic of the attack landscape. However, while they are effective in detecting known existing attacks on ActiveX components, they fail to identify attacks that do not involve ActiveX components, which ZOZZLE is able to detect.

8 Conclusions

This paper presents ZOZZLE, a highly precise, mostly static detector for malware written in JavaScript. ZOZZLE is a versatile technology that is suitable for deployment in a commercial browser, staged with a more costly runtime detector like NOZZLE. Designing an effective in-browser malware detector requires overcoming technical challenges that include achieving high performance, generating precise results, and overcoming attempts at obfuscating attacks. Much of the novelty of ZOZZLE comes from its hooking into the the JavaScript engine of a browser to get the final, expanded version of JavaScript code to address the issue of deobfuscation. Compared to other classifier-based tools, ZOZZLE uses contextual information available in the program abstract syntax tree (AST) to perform fast, scalable, yet precise malware detection.

This paper contains an extensive evaluation of our techniques. We evaluated ZOZZLE in terms of performance and malware detection rates (both false positives and false negatives) using over 1.2 million pre-categorized code samples. ZOZZLE has an extremely low false positive rate of 0.0003%, which is less than one in a quarter million. Despite this high accuracy, the ZOZZLE classifier is fast, with a throughput at over one megabyte of JavaScript code per second.

Acknowledgments

This work would not have been possible without the help of many people, including Sarmad Fayyaz, David Felstead, Michael Gamon, Darren Gehring, Rick Gutierrez, Engin Kirda, Jay Stokes, and Ramarathnam Venkatesan. We especially thank Rick Bhardwaj for working closely with malware samples to help us understand their properties in the wild.

References

[1] M. Abadi, M. Budiu, Úlfar Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the Conference on Computer and Communications Security*, 2005.

[2] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. G. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In *Proceedings of Security and Privacy in the Age of Ubiquitous Computing*, 2005.

[3] A. Buescher, M. Meier, and R. Benzmueller. Monkey-Wrench - boesartige webseiten in die zange genommen. In *Deutscher IT-Sicherheitskongress*, Bonn, 2009.

[4] D. Canali, M. Cova, G. Vigna, and C. Krügel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the International World Wide Web Conference*, Mar. 2011.

[5] M. Castro, M. Costa, and T. Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2006.

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

[7] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the USENIX Security Symposium*, January 1998.

[8] Y. Ding, T. Wei, T. Wang, Z. Liang, and W. Zou. Heap Taichi: exploiting memory allocation granularity in heap-spraying attacks. In *Proceedings of Annual Computer Security Applications Conference*, 2010.

[9] M. Egele, P. Wurzinger, C. Krügel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of the Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2009.

[10] B. Feinstein and D. Peck. Caffeine Monkey: Automated collection, detection and analysis of malicious JavaScript. In *Proceedings of Black Hat USA*, 2007.

[11] F. Howard. Malware with your mocha: Obfuscation and anti-emulation tricks inmalicious JavaScript. `http://www.sophos.com/security/technical-papers/malware_with_your_mocha.pdf`, Sept. 2010.

[12] M. Howard. Address space layout randomization in Windows Vista. `http://blogs.msdn.com/b/michael_howard/archive/2006/05/26/address-space-layout-randomization-\\in-windows-vista.aspx`, May 2006.

[13] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the USENIX Windows NT Symposium*, 1999.

[14] S. Karanth, S. Laxman, P. Naldurg, R. Venkatesan, J. Lambert, and J. Shin. Pattern mining for future attacks. Technical Report MSR-TR-2010-100, Microsoft Research, 2010.

[15] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware on the web. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2006.

[16] J. Nazario. PhoneyC: A virtual client honeypot. In *Proceedings of the Usenix Workshop on Large-Scale Exploits and Emergent Threats*, Boston, 2009.

[17] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2005.

[18] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Emulation-based detection of non-self-contained polymorphic shellcode. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection*, 2007.

[19] Praetorian Prefect. The "aurora" IE exploit used against Google in action. `http://praetorianprefect.com/archives/2010/01/the-aurora-ie-exploit-in-action/`, Jan. 2010.

[20] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us. In *Proceedings of the USENIX Security Symposium*, 2008.

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

[22] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Proceedings of the Annual Computer Security Applications Conference*, 2010.

[23] M. Roesch. Snort – lightweight intrusion detection for networks. In *Proceedings of the USENIX Conference on System Administration*, 1999.

[24] C. Seifert, P. Komisarczuk, and I. Welch. Identification of malicious web pages with static heuristics. In *Proceedings of the Austalasian Telecommunication Networks and Applications Conference*, 2008.

[25] C. Seifert, R. Steenson, T. Holz, B. Yuan, and M. A. Davis. Know your enemy: Malicious web servers. 2007.

[26] C. Song, J. Zhuge, X. Han, and Z. Ye. Preventing drive-by download via inter-module communication monitoring. In *Proceedings of the Asian Conference on Comuting and Communication Security*, 2010.

[27] R. J. Spoor, P. Kijewski, and C. Overes. The HoneySpider network: Fighting client-side threats. In *Proceedings of FIRST*, 2008.

[28] T. Stuurman and A. Verduin. Honeyclients - low interaction detection methods. Technical report, University of Amsterdam, 2008.

[29] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection*, 2002.

[30] K. Wang. HoneyClient. `http://www.honeyclient.org/trac`, 2005.

[31] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2006.

[32] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economy on the Chinese web. Technical report, University of Mannheim, 2007.

Shellcode obfuscation strategy	Spray	CVE
unescape	✓	2009-0075
unescape	✓	2009-1136
unescape	✓	2010-0806
unescape	✓	2010-0806
none	✗	2010-0806
hex, unescape	✓	none
replace, unescape	✗	none
unescape	✓	2009-1136
replace, hex, unescape	✓	2010-0249
custom, unescape	✓	2010-0806
unescape	✓	none
replace, array	✓	2010-0249
unescape	✓	none
unescape	✓	2009-1136
replace, unescape	✗	none
replace, unescape	✓	none
unescape	✓	2010-0249
unescape	✓	2010-0806
hex, unescape	✓	2008-0015
unescape	✗	none
replace, unescape	✗	none
unescape, array	✓	2010-0249
replace, unescape	✓	2010-0806
replace, unescape	✓	2010-0806
replace, unescape	✓	none
replace, unescape	✗	none

Figure 19: Malware samples dissected and categorized.

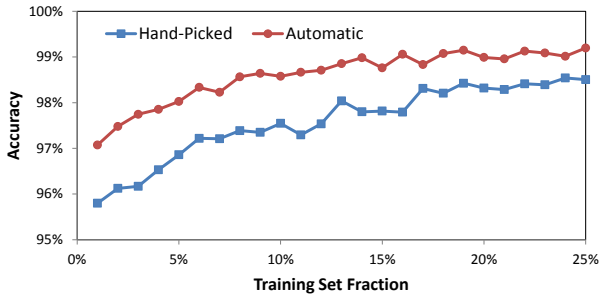


Figure 20: Classification accuracy as a function of training set size for hand-picked and automatically selected features.

A Hand-Analyzed Samples

In the process of training the ZOZZLE classifier, we hand-analyzed a number of malware samples. While there is a great deal of duplication, there is a diversity of malware writing strategies found in the wild.

Figure 19 provides additional details about each unique hand-analyzed sample. Common Vulnerabilities and Exposures (CVEs) are assigned when new vulnerabilities are discovered and verified, and these identifiers are listed for all the exploits in Figure 19 that target some vulnerability. Shellcode and nopsled type describe the method by which JavaScript or HTML values are converted to the binary data that is sprayed throughout the heap. Most shellcode and nopsleds are written as hexadecimal literals using the `\x` escape sequence. These cases are denoted by "hex" in Figure 19.

Many scripts use the `%u` encoding and are converted to binary data with the JavaScript `unescape` function. Finally, some samples include short fragments inserted repeatedly (such as the string `CUTE`, which appears in several examples) that are removed or replaced by a call to the JavaScript `replace` function.

In a few cases, the exploit sample does not contain one or more of the components of a heap spray attack (shellcode, spray, and vulnerability). In these cases, the script is delivered with one or more of the other samples for which it may provide shellcode, perform a spray, or trigger a vulnerability.

B Additional Experimental Data

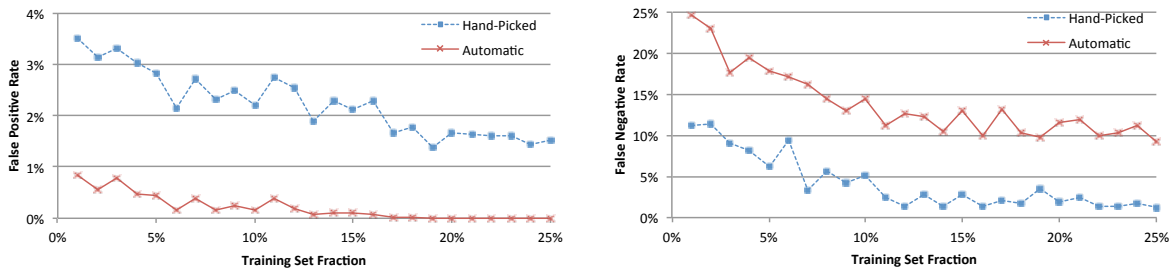
Training set size: To understand the impact of training set size on accuracy and false positive/negative rates, we trained classifiers using between 1% and 25% of our benign and malicious datasets. For each training set size, ten classifiers were trained using different randomly selected subsets of the dataset for both hand-picked and automatic features. These classifiers were evaluated with respect to overall accuracy in Figure 20 and false positives/negatives in Figure 21a.

The figures show that training set size does have an impact on the overall accuracy and error rates, but that a relative small training set ($< 5\%$ of the overall data set) is sufficent to realize most of the benefit. The false positive negative rate using automatic feature selection benefits the most from additional training data, which is explained by the fact that this classifier has many more features and benefits from more examples to fully train.

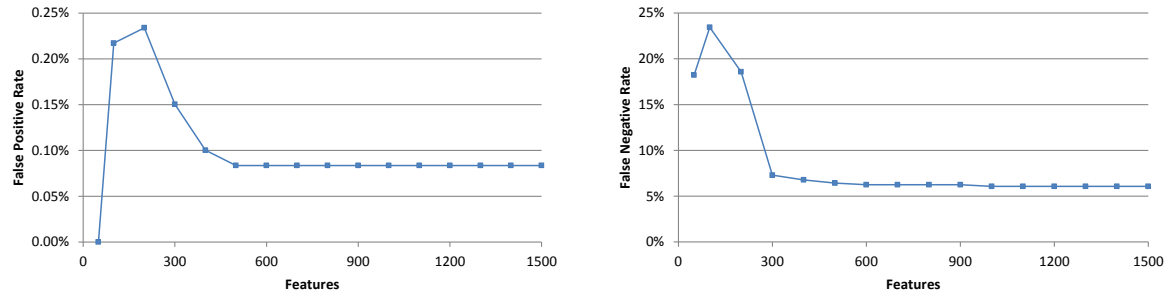
Feature set size: To understand the impact of feature set size on classifier effectiveness, we trained the 1-level automatic classifier, sorted the selected features by their χ^2 value, and picked only the top N features. For this experiment (due to the fact that the training set used is randomly selected), there were a total of 1,364 features originally selected during automatic selection.

Figure 21b shows how the false positive and false negative rates vary as we change the size of the feature set to contain 500, 300, 100, and 30 features, respectively.

The figures show that the false positive rate remains low (and drops to 0 in some cases) as we vary the feature set size. Unfortunately, the false negative rate increases



(a) as a function of training set size.



(b) as a function of feature set size.

Figure 21: False positive and false negative rates.

```
function dF(s)
{
    var s1 = unescape(s.substr(0,s.length - 1)),
        t = "";
    for(i = 0; i < s1.length; i++)
        t += String.fromCharCode(
                s1.charCodeAt(i) -
                s.substr(s.length - 1,1));
    document.write(unescape(t))
}
```

Figure 22: Code unpacker detected by anti-virus tools.

steadily with smaller feature set sizes. The implication is that while a small number of features can effectively identify some malware (probably the most commonly observed malware), many of the most obscure malware samples will remain undetected if the feature set is too small.

C Additional Code Samples

Figure 22 shows a code unpacker that is incorrectly flagged by overly eager anti-virus engines. Of course, the unpacker code itself is not malicious, even though the contents it may unpack could be malicious. Finally, Figure 23 shows an example of code that anti-virus engines overeagerly deem as malicious.

```
document.write(unescape('%3C%73%63...'));
dF('%264Dtdsjqu%264Fepdvnfou/xsjuf%2639
%2633%264Dtdsjqu%2631tsd%264E%266D%2633%2633
%2C%2633iuuq%264B00jutbmmcsfbltpgu/ofu0uet0jo/
dhj%264G3%2637tfpsfg%264E%2633
%2CfodpefVSJDpnqpofou%2639epdvnfou/sfgfssfs
%263%3A%2C%2633%2637qbsbnfufs%264E
%26351fzxpse%2637tf%264E%2635tf%2637vs
%264E2%2637IUUQ%60SFGFSFS%264E%2633%2C
%2631fodpefVSJDpnqpofou%2639epdvnfou/VSM
%263%3A%2C%2633%2637efgbvmu%601fzxpse
%264Eopuefgjof%2633%2C%2633%266D%2633
%264F%264D%266D0tdsjqu%264F%2633%263%3A
%264C%264D0tdsjqu%264F%261B%264Dtdsjqu%264F
%261Bjg%2639uzqfpg%2639i%263%3A%264E
%264E%2633voefgjofe%2633%263%3A%268C%261
%3A%261B%261%3Aepdvnfou/xsjuf%2639%2633
%264Djgsbnf%2631tsd%264E%2638iuuq
%264B00jutbmmcsfbltpgu/ofu0uet0jo/dhj%264G4
%2637tfpsfg%264E%2633%2CfodpefVSJDpnqpofou
%2639epdvnfou/sfgfssfs%263%3A%2C%2633
%2637qbsbnfufs%264E%26351fzxpse%2637tf
%264E%2635tf%2637vs%264E2%2637IUUQ%60SFGFSFS
%264E%2633%2C%2631fodpefVSJDpnqpofou
%2639epdvnfou/VSM%263%3A%2C%2633%2637efgbvmu
%601fzxpse%264Eopuefgjof%2638%2631xjeui
%264E2%2631ifjhiu%264E2%2631cpsefs%264E1
%2631gsbnfcpsefs%264E1%264F%264D0jgsbnf
%264F%2633%263%3A%264C%2631%261B%268E%261Bfmtf
%2631jg%2639i/joefyPg%2639%2633iuuq
%264B%2633%263%3A%264E%264E1%263%3A%268C%261B%261
%3A%261%3Axjoepx/mpdbujpo%264Ei%264C%261B
%268E%261B%264D0tdsjqu%264F1')
```

Figure 23: Anti-virus false positive. A portion of the file after `unescape` is removed to avoid triggering AV on the final PDF of this paper.