Zozzle: Low-overhead Mostly Static JavaScript Malware Detection

Charles Curtsinger
UMass at Amherst

Benjamin Livshits and Benjamin Zorn
Microsoft Research

Christian Seifert
Microsoft

Microsoft Research Technical Report

MSR-TR-2010-156

Abstract

JavaScript malware-based attacks account for a large fraction of successful mass-scale exploitation happening today. From the standpoint of the attacker, the attraction is that these drive-by attacks 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 tend to incur.

In this paper, we propose ZOZZLE, a low-overhead solution for detecting and preventing JavaScript malware that can be deployed in the browser. Our experience also suggests that ZOZZLE may be used as a lightweight filter for a more costly detection technique or for standalone offline malware detection.

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 extensive experimental evaluation shows that ZOZZLE is able to detect JavaScript malware through mostly static code analysis effectively. In our experiments, the most accurate classifier did not produce any false positives, implying a false positive rate of below 0.01%. Despite this high accuracy, the classifier is very fast, with a throughput at over 1 MB of JavaScript code per second.

I. 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 [11] or operating system mechanisms such as NX/DEP and ALSR [18]. While several heap spraying solutions have been proposed [12, 34], arguably, none are lightweight enough to be integrated into a commercial browser.

Specifically, Nozzle is an effective heap spraying prevention technique with a low false positive and false negative rate. However, the overhead of this runtime technique may be 10% or higher, which is too much overhead to be acceptable in today's competitive browser market [34]. This paper is based on our experience using Nozzle for offline (HoneyMonkey-style) heap spray detection. By deploying Nozzle on a large scale, in the context of a dynamic web crawler, we are able to scan millions of URLs daily and find thousands of heap spraying attacks in the process. Offline URL scanning is a practical alternative to online, in-browser detection. Malicious URLs detected in this manner are used in the following two ways: to enhance a browser-based blacklist of malicious URLs as well as a blacklist of sites not to serve with a search engine.

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, often at new URLs. In this paper we present a third application of offline scanning techniques: we show how to use Nozzle offline scanning results on a large scale to train Zozzle, a lightweight heap spraying JavaScript malware detector.

Zozzle is a *mostly static* detector that is able to examine a page and decide if it contains a heap spray exploit. 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.

This paper discusses the design and evaluation of Zozzle and proposes two deployment strategies. We show how to integrate such a detector into the JavaScript parser of a browser in a manner that makes the runtime performance overhead minimal. We also suggest how to combine Zozzle with a secondary detection technique such as Nozzle within the browser, as well as using it for offline scanning. In our experience, Zozzle finds many more malware pages than Nozzle, in part because Nozzle requires an attack to be initiated for a successful detection, whereas with Zozzle, it is enough for the underlying JavaScript code to appear malicious.

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 thousands of malicious sites daily in the wild.

While we acknowledge that a determined attacker may be able to design malware that Zozzle will not detect, our focus in this paper is on creating a very low false positive, low-overhead scanner. Just as is the case with anti-virus tools, often the penalty for a single false positive — marking a benign site as malicious — for a malware detector is considerably higher than the cost of a false negative. Consequently, the main focus of both our design and experimental evaluation is on ensuring that the false positive rate of Zozzle remain very low. In our experiments conducted on a scale of millions of URLs, we show that Zozzle produces false positive rates of a fraction of 1%, while incurring only a small runtime overhead, usually not exceeding a few milliseconds per JavaScript file.

A. Contributions

This paper makes the following contributions:

- **Mostly static malware detection.** We propose Zozzle, a highly precise lightweight mostly static JavaScript malware detection approach. 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 [34].
- **AST-based detection.** We describe an AST-based technique that involves the use of hierarchical (context-sensitive) features for detecting malicious JavaScript code, which scales to over 1,000 features, while remaining fast enough to use on-the-fly, in the context of a web browser. This is in part due to our fast, parallel feature matching that is a considerable improvement over the naïve sequential feature matching approach.
- **Evaluation.** We evaluate Zozzle in terms of performance and malware detection rates (both false positives and false negatives) using thousands of labeled code samples. Our most accurate classifier did not produce any false positives, implying a false positive rate of below 0.01%. Despite this high accuracy, the classifier is very fast, with a throughput at over 1 MB of JavaScript code per second for deployable classifier configurations.
- **Staged analysis.** We advocate the use of Zozzle as a first stage for Nozzle or anti-virus-based detection. In this setting, the use of Zozzle drastically reduces the expected end-user overhead for in-browser protection. We also evaluate Zozzle extensively by using it in the context of offline scanning.

B. Paper Organization

The rest of the paper is organized as follows. Section II gives some background information on JavaScript exploits and their detection and summarizes our experience of performing

```
<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>
```

Fig. 1: Heap spraying attack example.

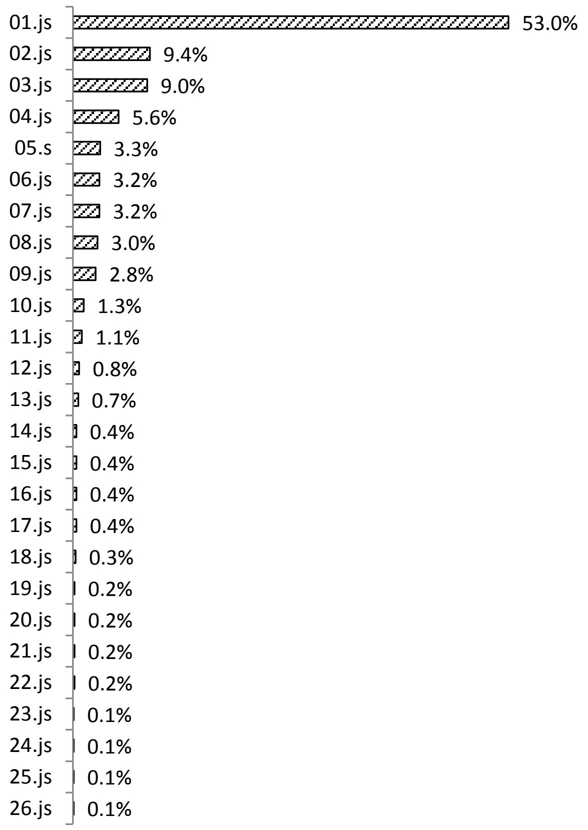


Fig. 2: Distribution of different exploit samples in the wild.

offline scanning with NOZZLE on a large scale. Section III describes the implementation of our analysis. Section IV describes our experimental methodology. Section V describes our experimental evaluation. Sections VI and VII discusses related and future work, and, finally, Section VIII concludes.

II. OVERVIEW

This section is organized as follows. Section II-A gives overall background on JavaScript-based malware, focusing specifically on heap spraying attacks. Section II-B summarizes our experience of offline scanning using NOZZLE and categorizes the kind of malware we find using this process. Section II-C digs deeper into how malware is usually structured, including the issue of obfuscation. Finally, Section II-D provides a summary of the ZOZZLE techniques.

A. JavaScript Malware Background

Figure 1 shows an example of malicious JavaScript containing a typical heap-spraying attack. Such an attack 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 may be easily accomplished using an array of strings. Spraying of this sort can be used to defeat address space layout randomization (ASLR) protection in the operating system [18]. 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 [17]. To avoid detection, the primary techniques 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.

B. Characterizing Malicious JavaScript

To gather the data we use to train the ZOZZLE classifier and evaluate it, we employed a web crawler to visit many ranomly selected URLs and process them with NOZZLE to

Sample Name	Shellcode	Spray	CVE
01.js	unescape	yes	2009-0075
02.js	unescape	yes	2009-1136
03.js	unescape	yes	2010-0806
04.js	unescape	yes	2010-0806
05.js	none	no	2010-0806
06.js	hex, unescape	yes	none
07.js	replace, unescape	no	none
09.js	unescape	yes	2009-1136
08.js	replace, hex, unescape	yes	2010-0249
10.js	custom, unescape	yes	2010-0806
11.js	unescape	yes	none
12.js	replace, array	yes	2010-0249
13.js	unescape	yes	none
14.js	unescape	yes	2009-1136
15.js	replace, unescape	no	none
16.js	replace, unescape	yes	none
17.js	unescape	yes	2010-0249
18.js	unescape	yes	2010-0806
19.js	hex, unescape	yes	2008-0015
20.js	unescape	no	none
21.js	replace, unescape	no	none
22.js	unescape, array	yes	2010-0249
23.js	replace, unescape	yes	2010-0806
24.js	replace, unescape	yes	2010-0806
25.js	replace, unescape	yes	none
26.js	replace, unescape	no	none

Fig. 3: Malware samples described.

detect if malware was present. Note that because we use NOZZLE to identify malicious JavaScript, the malware we detect always contains a heap-spray in addition to shellcode and the vulnerability.

Once we determine that JavaScript is malicious, we invested a considerable effort in examining the code by hand and categorizing it in various ways. Here we present summary statistics about the body of malicious JavaScript that we have identified and classified.

Figure 2 shows the observed distribution of distinct malware instances based on 169 malware samples we investigated.

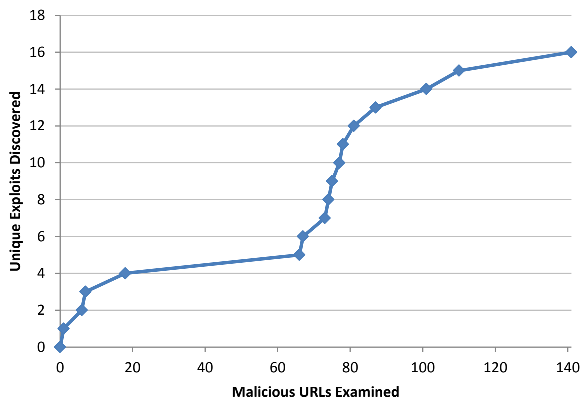


Fig. 4: Saturation: discovering more exploit samples over time. The x axis shows the number of examined malware samples, the y axis shows the number of unique ones.

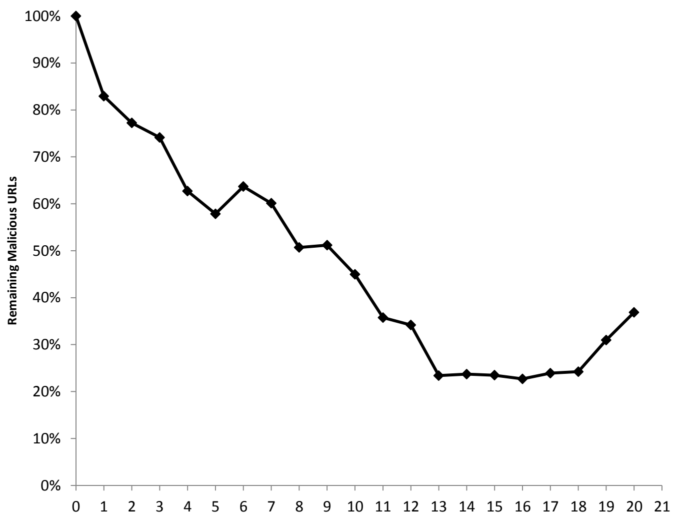


Fig. 5: Transience of detected malicious URLs after several days. The number of days is shown of the x axis, the percentage of remaining malware is shown on the y axis.

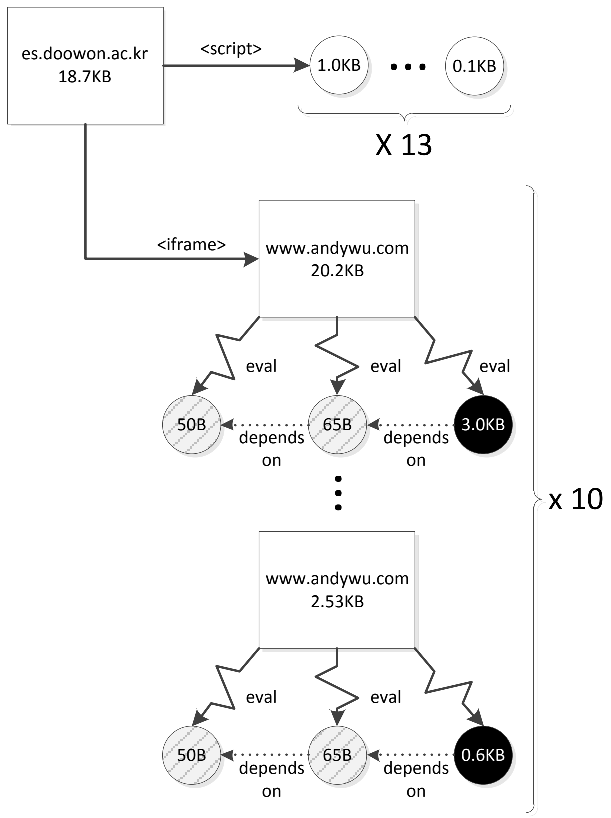


Fig. 6: Unfolding tree: an example. Rectangles are documents, and circles are JavaScript contexts. Gray circles are benign, black are malicious, and dashed are “co-conspirators” that participate in deobfuscation. Edges are labeled with the method by which the context or document was reached. The actual page contains 10 different exploits using the same obfuscation.

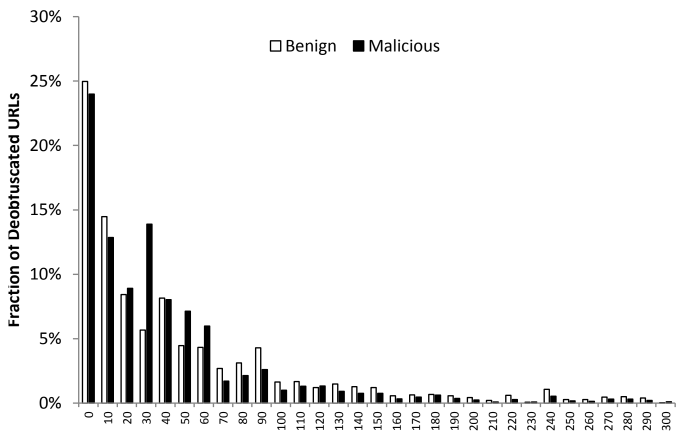


Fig. 7: Distribution of context counts for malware and benign code.

The figure shows that the malware we detected observes a highly skewed distribution, with one of the samples accounting for more than 50% of the total malware observed. We give each unique sample a label, and Figure 3 provides additional details about each sample, including the CVE number of the vulnerability being exploited, how the shellcode and NOP sled is constructed, and how the spray is executed.

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. The 'decode' type is an ad-hoc encoding scheme developed as a part of the exploit. Most shellcode and nopsleds are written using the `%u` or `\x` encoding and are converted to binary data with the JavaScript `unencode` function. Finally, some samples include short fragments inserted at many places (such as the string `CUTE`) that are removed or replaced with a call to the JavaScript `replace` function. In all but one case, the heap-spray is carried out using a `for` loop. The 'plugin' spray is executed by an Adobe Flash plugin included on the page, and not by JavaScript code. The CVE is the identifier assigned to the vulnerability exploited by the sample. CVEs (Common Vulnerabilities and Exposures) are assigned when new vulnerabilities are discovered and verified. This database is maintained by the MITRE Corporation.

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 5 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 cannot keep 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.

C. Dynamic Malware Structure

One of the core issues that needs to be addressed when talking about JavaScript malware is the issue of *obfuscation*. 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 [17]. Many approaches to code obfuscation exist. In practice, we often see `eval` unfolding as the most prominent. 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 6 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 exploit. To increase the chances of 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 a `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.

We instrumented the ZOZZLE deobfuscator to collect information about which code context leads to other code contexts, allowing us to collect information about code unfolding depth. Figure 7 shows a distribution of JavaScript context counts for benign and malicious URLs. The majority of URLs have only several JavaScript code contexts, however, many can 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

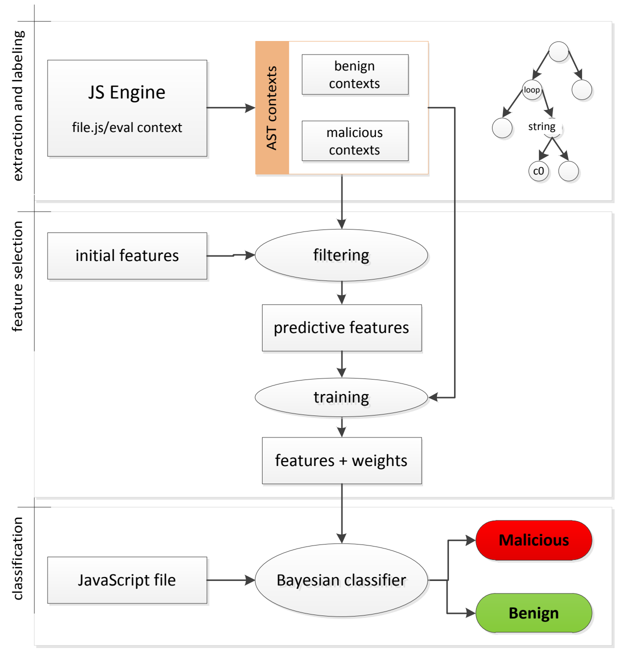


Fig. 8: ZOZZLE architecture.

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 7 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.

D. *ZOZZLE 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 8 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.

Caveats and limitations: As with any classifier-based tool, the issue of false negatives is difficult to avoid. In particular, knowing the list of features used by ZOZZLE, a determined attacker may be able to design exploits that would not be detected, especially if ZOZZLE is available as an oracle for experimentation purposes while the attacker is debugging her exploit. A weakness of any classifier-based approach that is easier to circumvent than runtime detection techniques such as NOZZLE, although we believe it is impossible for an attacker to remove all informative features from an attack, particularly JavaScript language constructs required for the exploit to succeed.

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 adopt to the changing detection landscape. Indeed, despite wide availability of obfuscation tools, in our NOZZLE detection experiments, we still find many sites not using any form of obfuscation at all, using obvious variable names such as `shellcode`, `nopsled`, etc.

Deployment: Several deployment strategies for ZOZZLE exist. The most attractive one, we believe, 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. 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, making 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, 9]. 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 do not necessarily object to the user taking a performance hit in the case of an actual exploit taking place.

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

III. IMPLEMENTATION

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

A. *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 [20], 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.

B. 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.

C. Feature Selection

As illustrated in Figure 8, after creating an initial feature set, ZOZZLE 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 (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.

D. 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 IV.

E. 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 10. 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 AND-ed 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 9. 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. This is 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 11 clearly shows the benefit of fast pattern matching over a naive 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, decreasing asymptotically as the number of features approaches 1,500.

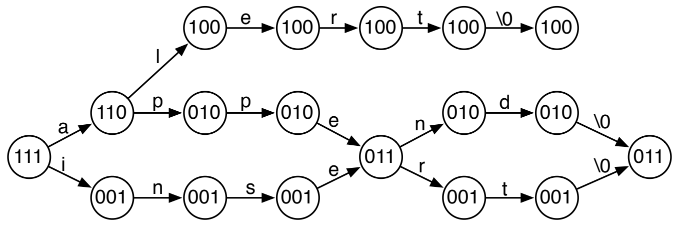


Fig. 9: Fast feature matching illustrated.

```
$matchList \leftarrow \langle 1, 1, \ldots, 1 \rangle$
$state \leftarrow 0$
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 $matchList$
```

Fig. 10: Fast matching algorithm.

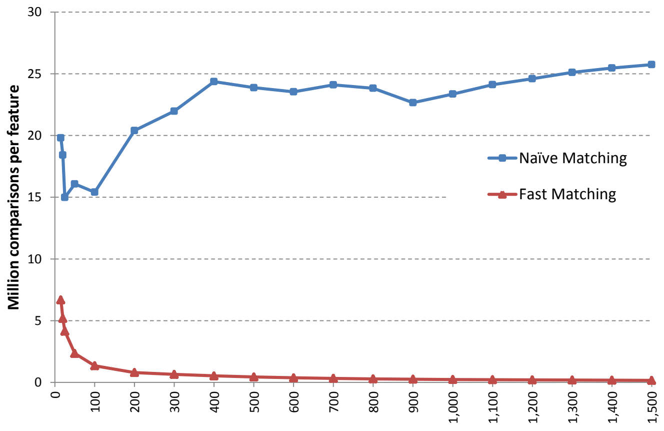


Fig. 11: Comparisons required per-feature with naïve vs. fast pattern matching. The number of features is shown on the x axis.

IV. Methods

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.

A. Gathering Malicious Samples

To gather the results for Section V, 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

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

Fig. 12: Examples of hand-picked features used in our experiments.

explained first in Section II 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. The frequency of each exploit type observed is shown in Figure 2.

B. 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.

C. 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 III) 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.

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 12.

For comparison purposes, samples of the automatically extracted features, including a measure of their discriminating power, are shown in Figure 13. 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 set.

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 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.

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

Fig. 13: Sample of automatically selected features and their discriminating power as a ratio of likelihood to appear in a malicious or benign context.

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

Fig. 14: 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%

Fig. 15: False positives and false negatives for flat and hierarchical features using hand-picked and automatically selected features.

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.

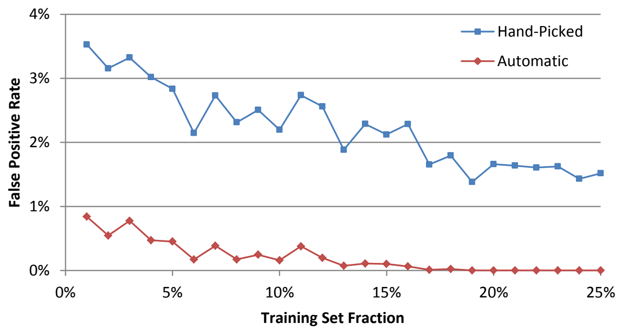
For n-level features, we record the entire stack of AST contexts such as

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

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 V.

V. EVALUATION

In this section, we evaluate the effectiveness of ZOZZLE using the benign and malicious JavaScript samples described in Section IV. The purpose of this section is to answer the following questions:



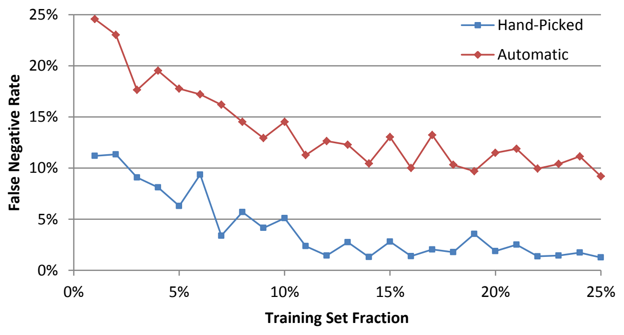
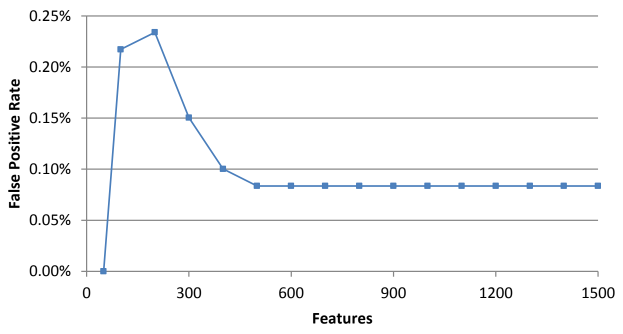


Fig. 17: False positive and false negative rates as a function of training set size.



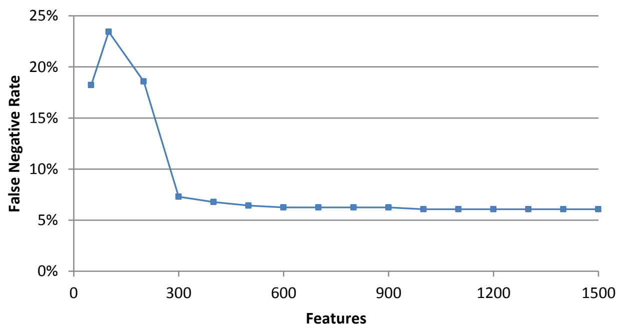


Fig. 18: False positive and false negative rates as a function of feature set size.

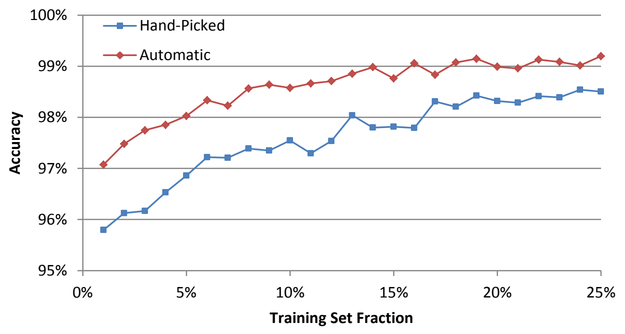


Fig. 16: Classification accuracy as a function of training set size for hand-picked and automatically selected features.

- How effective is ZOZZLE at correctly classifying both malware and benign JavaScript?
- What benefit do we get by including context information from the abstract syntax tree in creating classifiers?
- In terms of the amount of malware detected, how does ZOZZLE compare with other approaches, such as NOZZLE?
- What is the performance overhead of including ZOZZLE in a browser?

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.

A. ZOZZLE *Effectiveness: False Positives and False Negatives*

Figure 14 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 III, and then sub-divided by the amount of context used in the classifier (flat, 1-level, and n-level).

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.

Figure 15 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

[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.

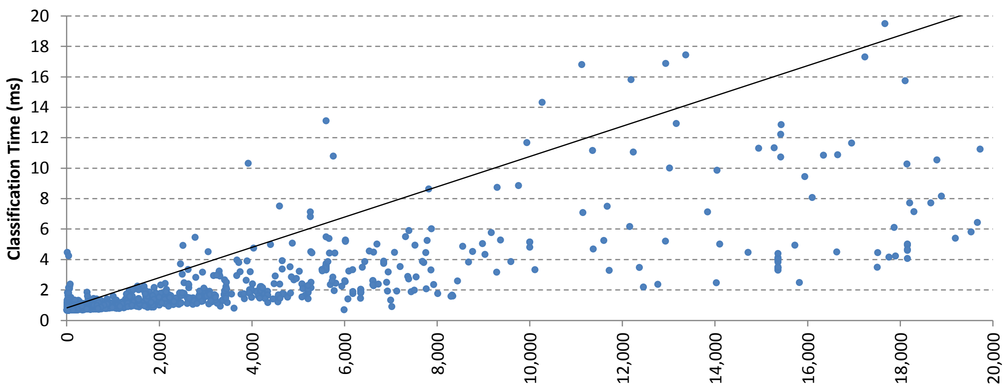


Fig. 20: Classification time as a function of JavaScript file size. File size in bytes is shown on the x axis and the classification time in ms is shown on the y axis.

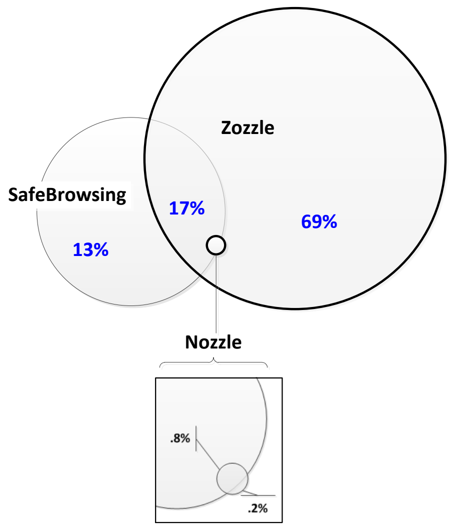


Fig. 19: A Venn diagram showing the quantitative relationship between ZOZZLE, NOZZLE, and Google SafeBrowsing. The callout at the bottom of the figure shows the percentage of NOZZLE findings that are covered by SafeBrowsing (80%).

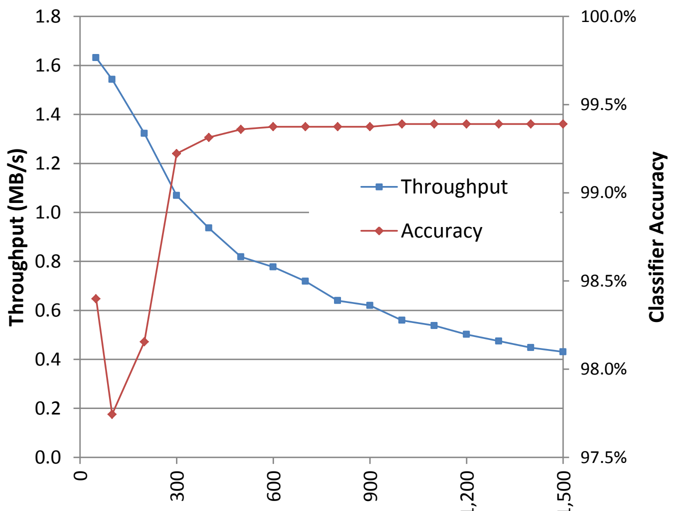


Fig. 21: Classifier throughput and accuracy as a function of the number of features, using 1-level classification with .25 of the training set size.

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. We note, however, that the differences between the automatic false positive rates are so small that the variation (0% versus 0.02%) may be the result of noise due to the randomness of training set selection.

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.

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 16 and false positives/negatives in Figure 17.

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 (< 10% of the overall data set) is sufficent to realize most of the benefit. The false positive 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 18 shows how the false positive and false negative rates vary as we change the size of the feature set to contain between 50 and 1300 features. The figures show that both the false positive and false negative rates are significantly higher and variable with small feature sets but after approximately 200 features, both rates steadily drop until we have approximately 500 features, at which point the rates stabilize. For both rates, using more than 600 features has little overall effect.

B. Comparison with Other Techniques

In this section, we compare the effectiveness of ZOZZLE to that of other malware detection tools, including NOZZLE and Google's Safe Browsing API [14]. SafeBrowsing is a frequently updated blacklist of malicious sites that is used by Google Chrome and Mozilla Firefox to warn users of potentially malicious URLs they might be visiting. The list is distributed in the form of hash values, with original URLs not actually being exposed.

Our approach is to consider a collection of benign and malicious URLs and determine which of these is flagged as malicious by the different detection tools.

To compute the relative numbers of detected URLs for NOZZLE, ZOZZLE, and SafeBrowsing, we first take the total number of NOZZLE detections from our dynamic crawler infrastructure. It is infeasible to run ZOZZLE on every URL visited by the crawler on our single test machine (collecting deobfuscated JavaScript is the bottleneck here), so instead we use a random sample (4% of all crawled URLs) in order to estimate the number of ZOZZLE and SafeBrowsing detections for a single day. None of the randomly selected URLs were detected by NOZZLE, so we also ran ZOZZLE on all the URLs detected by NOZZLE. This gives us the exact number of URLs detected by both techniques. Dividing this count by the approximate number of ZOZZLE detections (ZOZZLE detections divided by sample rate) for the whole set of URLs gives us an estimate as to what fraction of ZOZZLE-detected URLs are also detected by NOZZLE. This same procedure was repeated for SafeBrowsing.

Our results are shown in Figure 19. This diagram illustrates the relative numbers of URLs detected by NOZZLE, ZOZZLE, and Google SafeBrowsing. We see that SafeBrowsing detects a large fraction of the URLs detected by NOZZLE, while ZOZZLE detects them all. SafeBrowsing detects less than half of the URLs found by ZOZZLE, but we have hand-verified a large number of these and have found that they are not false positives. ZOZZLE detects a large fraction of SafeBrowsing's URLs, but not all. Some URLs are flagged by SafeBrowsing as being part of a malware network, even if these URLs do not contain an actual exploit. Some URLs were flagged by the client-side SafeBrowsing tool, but Google's diagnostics page for the URL indicated that it was not malicious, which we believe can result from hash conflicts in the SafeBrowsing database sent to clients. Finally, there are legitimately malicious URLs detected by SafeBrowsing and not by ZOZZLE. Some of these are missed detections, while others are exploits that do not expose themselves to all browsers.

C. Classifier Performance

Figure 20 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 4000 features, and represents the worst case performance for classification. We see that for a majority of files, classification can be performed in under 4ms. 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 21 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 III, 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 prototype implementation, we believe that ZOZZLE's static detector has a lot of potential for fast on-the-fly malware identification.

VI. RELATED

Detection and prevention of drive-by-download attacks has been receiving a lot of attention over the last few years. Some of the methods have been incorporated into browser plug-ins, intrusion detection and prevention systems or client honeypots. This section focuses on the underlying techniques as opposed to the means by which they are deployed. We first review related work on detection of malicious web pages followed by a brief review on specific protection approaches.

A. Detection

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 [26, 33, 39, 45–47]. 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. Similar to ZOZZLE, they are capable to detect zero-days, but they are known to miss attacks. For a client honeypot that monitors the effect of a successful attack, the attack needs to succeed. As such, a client honeypot that is not vulnerable to a specific attack is unable to detect the attack (e.g. a web page that attacks Firefox browser cannot be detected by a high-interaction client honeypot that runs Internet Explorer.) ZOZZLE mines JavaScript features that are indiscriminative of the attack target and therefore does not have this limitation.

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 [13, 33, 38, 42, 43]. While these approaches, among others, consider static JavaScript features, ZOZZLE is the first to utilize hierarchical features extracted from ASTs. However, all produce some false positives and therefore are usually employed as a first stage to follow a more in-depth detection method, such as NOZZLE or high-interaction client honeypots.

Recent work by Canali *et al.* [8] presents a lightweight static filter for malware detection called Prophiler. It combines HTML-, JavaScript-, and URL-based features into one classifier that is able to quickly discard benign pages. 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 enty point, whereas Prophiler extracts its features from the *obfuscated* code and combine them with HTML- and URL-based features into one classifier. Second, ZOZZLE applies an automated feature extraction mechanisms that selects *hierarchical* features from the AST, whereas Prophiler hand-picks a variety of statistical and lexical JavaScript features including features from a generated AST. Their feature “presence of decoding routines ” seems to utilize loop context information of long strings and is recognized as an effective feature correlated with malicious code by ZOZZLE as well. However, we should emphasize that ZOZZLE more generically extracts context-related information from the AST in an automated fashion. Third, the emphasis of ZOZZLE is on very low false positive rates, which sometimes imply higher false negative rates, whereas Prophiler has a more balanced approach to both. Fourth, the focus of ZOZZLE is on heap spraying detection, whereas Prophiler is more general. More narrow focus allows us to train ZOZZLE more precisely, using NOZZLE results, while perhaps not finding as many malicious sites as Prophiler is capable of detecting. We believe that there is room for combining interesting aspects of both tools.

Besides static features focusing on HTML and JavaScript, shellcode injection exploits also offer points for detection. They usually consist of several parts: the NOP sled, shellcode, the spray, and a vulnerability. Existing techniques such as Snort [36] use pattern matching to identify attacks in a database. This approach most likely fails to detect attacks that are not already in the 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 [29]. Like ZOZZLE, Polygraph utilizes a naive bayes classifier, but only applies it to the detection of shellcode.

Abstract Payload Execution (APE) by Toth and Kruegel [44], STRIDE by Akritidis *et al.* [3, 31], and NOZZLE by Ratanaworabhan, Livshits and Zorn [34] 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 unacceptable for always-on deployment in a browser (10-15% is fairly common). As mentioned earlier, ZOZZLE could be used to identify potential sprays and enable one of these detection tools when suspicious code is detected to dramatically reduce the overhead of spray detection when visiting benign sites.

Dynamic features have been the focus of several groups. Nazario, Buescher, and Song propose systems that detect attacks on scriptable ActiveX components [7, 28, 41]. They capture JavaScript interactions — either by instrumenting or simulating a browser — and use vulnerability specific signatures to detect attacks (e.g. method `foo` with a long first parameter on vulnerable ActiveX component `bar`.) This method is very 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 unseen attacks, or those that do not involve ActiveX components, which ZOZZLE is able to detect.

Cova *et al.* present a system `JSAND` that conducts classification based on static and dynamic features [10]. In this system, 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. Results show 0% false positives and 0.2% false negative rate. ZOZZLE's approach is similar, but has some distinct differences. ZOZZLE only considers AST features from the JavaScript that is compiled within the browser whereas `JSAND` extracts manually selected runtime features, such as number of specific shellcode strings. ZOZZLE's feature selection is entirely automated.

Karanth et al. also attempt to identify malicious JavaScript based on features present in the code [21]. 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 that, over a period of four months, they were able to detect a previously unknown zero-day vulnerability. Unlike our work, they use a set of 34 hand-picked features for training and classification. While we use an off-the-shelf nave Bayesian classifier, their ranking algorithm is more specialized for their task. Their approach generates a precision of 100% with a recall up to about 70%, with the precision falling off after that. Overall, our approach has a lower false positive and false negative rate than theirs.

As mentioned above, some approaches chose to emulate a browser (such as `JSAND` or Nazario's PhoneyC [28]) whereas others use instrumented versions of the browser, such as ZOZ-

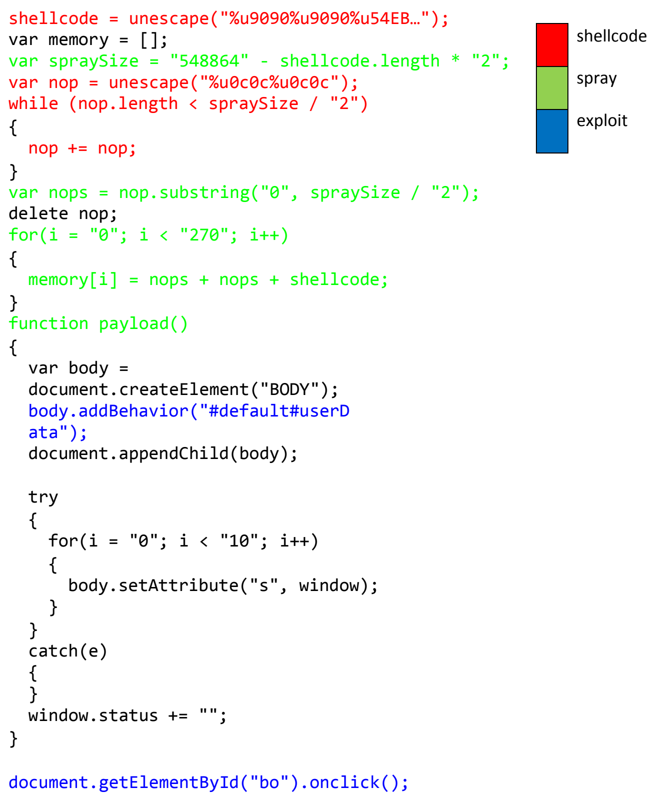
ZLE and Song's system [41]. Both approaches have advantages and disadvantages. Emulation, for instance, can be used to impersonate different browser personalities, but could also be detected by malicious web pages to evade the system as illustrated by Ruef as well as Hoffmann [16, 37]. Usually the implementation approach is not a design limitation of the method. Conceptually one method can be implemented on an emulated or instrumented platform. However, the implementation choice does need to be taken into account when considering the false negative rate as this design choice may influence this detection aspect.

B. Protection

Besides detection, protection-specific methods are presented. For instance, while ZOZZLE focuses on detecting malicious JavaScript, other techniques take different approaches. Recall that heap spraying requires an additional memory corruption exploit, and one method of preventing a heap-spraying attack is to ignore the spray altogether and prevent or detect the initial corruption error. Techniques such as control flow integrity [1], write integrity testing [2], data flow integrity [9], and program shepherding [22] take this approach. Detecting all such possible exploits is difficult and, while these techniques are promising, their overhead has currently prevented their widespread use.

Numerous research efforts are under way to directly protect the client application. BrowserShield, for instance, defuses malicious JavaScript at run-time by rewriting web pages and any embedded scripts into safe equivalents [35]. XSS-GUARD [6] proposes techniques to learn allowed scripts from unintended scripts. The allowed scripts are then white-listed to filter out unintended scripts at runtime. ConScript aims to use aspects to restrict how JavaScript can interact with its environment and the surrounding page [23]. Noncespaces, XSS Auditor, or "script accenting," make changes to the web browser that make it difficult for an adversary's script to run to avoid cross-site scripting attacks [5, 15, 27]. ZOZZLE can operate or be combined with one of these tools to provide protection to the end user from malicious JavaScript.

Protection mechanisms that do not involve JavaScript have also been explored, such as the ClearView system [30]. This research protects commercial off-the-shelve (COTS) software by detecting attacks in a collaborative environment and automatically applying generated patches to prevent such attacks in the future. Application of this method on the Firefox browser serves as a proof-of-concept. Anagnostakis *et al.* use anomaly detection and shadow honeypots to protect, among others, client applications, such as a web browser [4]. Before the web browser is permitted to process the requested data, suspicious data is forwarded to the shadow honeypot, an instrumented browser that detects memory violation attacks. If no attack is detected, the data is forwarded to the end user; if an attack is detected, the data is dropped and the end user effectively protected. A similar approach that uses execution-based web content analysis in disposable virtual machines is presented by Moshchuk *et al.* [25]. ZOZZLE could operate in a similar fashion with very little runtime overhead.



```
shellcode = unescape("%u9090%u9090%u54EB…");
var memory = [];
var spraySize = "548864" - shellcode.length * "2";
var nop = unescape("%u0c0c%u0c0c");
while (nop.length < spraySize / "2")
{
  nop += nop;
}
var nops = nop.substring("0", spraySize / "2");
delete nop;
for(i = "0"; i < "270"; i++)
{
  memory[i] = nops + nops + shellcode;
}
function payload()
{
  var body =
  document.createElement("BODY");
  body.addBehavior("#default#userD
  ata");
  document.appendChild(body);

  try
  {
    for(i = "0"; i < "10"; i++)
    {
      body.setAttribute("s", window);
    }
  }
  catch(e)
  {
  }
  window.status += "";
}

document.getElementById("bo").onclick();
```

🟥 shellcode
🟩 spray
🟦 exploit

Fig. 22: Colored malware sample.

Some existing operating systems also support mechanisms, such as Data Execution Prevention (DEP) [24], which prevent a process from executing code on specific pages in its address space. Implemented in either software or hardware (via the no-execute or "NX" bit), execution protection can be applied to vulnerable parts of an address space, including the stack and heap. With DEP turned on, code injections in the heap cannot execute.

While DEP will prevent many attacks, we believe that ZOZZLE is complementary to DEP for the following reasons. First, security benefits from defense-in-depth. For example, attacks that first turn off DEP have been published, thereby circumventing its protection [40]. Second, compatibility issues can prevent DEP from being used. Despite the presence of NX hardware and DEP in modern operating systems, existing commercial browsers, such as Internet Explorer 7, still ship with DEP disabled by default [19]. Third, runtime systems that perform just-in-time (JIT) compilation may allocate JITed code in the heap, requiring the heap to be executable.

VII. FUTURE WORK

We believe that much remains to be done in the area of precise classification of JavaScript, with ZOZZLE paving the way for the following major areas of future work.

A. Automatic Script Analysis and Script "Coloring"

ZOZZLE can be extended to classify at a finer granularity than entire scripts. A trained classifier could be used to identify the malicious *components* of a script. This could be taken a step further with a specially-trained classifier to distinguish between the components of an attack: the *shellcode*, *heap spray*, *vulnerability*, and *obfuscation*. This second, more powerful approach requires hand-labeled data. Our preliminary results are promising, which can be seen in Figure 22. This capability demonstrates the enormous power of source-level analysis and classification. For binary malware reverse-engineering and analysis this level of precision is unprecedented. With further work the process of annotating collected exploits could be automated by extending Nozzle to trace heap sprays back to JavaScript source lines.

B. 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.

C. 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.

D. 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. To prevent a particular feature from appearing in a particularly informative context (such as `COMMENT` appearing inside a loop, a component the Aurora exploit [32]) 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.

VIII. CONCLUSIONS

This paper presents ZOZZLE, a mostly static malware detector for JavaScript code. ZOZZLE is a versatile technology that may be deployed in the context of a commercial browser, staged with a more costly runtime detector like NOZZLE, or used standalone, for offline URL scanning. 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 thousands of pre-categorized code samples. We conclude that the most accurate classifier did not produce any false positives, implying a false positive rate of below 0.01%. Despite this high accuracy, the classifier is very fast, with a throughput at over 1 MB of JavaScript code per second.

We see tools like ZOZZLE deployed both in the browser to provide "first response" for users affected by JavaScript malware and used for offline dynamic crawling, to contribute to the creation and maintanence of various blacklists.

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*, pages 340–353, 2005.

[2] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with WIT. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[3] P. Akritidis, E. P. Markatos, M. Polychronakis, and K. G. Anagnostakis. STRIDE: Polymorphic sled detection through instruction sequence analysis. In R. Sasaki, S. Qing, E. Okamoto, and H. Yoshiura, editors, *Proceedings of Security and Privacy in the Age of Ubiquitous Computing*, pages 375–392. Springer, 2005.

[4] K. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the USENIX Security Symposium*, 2005.

[5] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. *International World Wide Web Conference*, 2010.

[6] P. Bisht and V. N. Venkatakrishnan. XSS-GUARD: precise dynamic prevention of cross-site scripting attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

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

[8] 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.

[9] 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*, pages 147–160, 2006.

[10] 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.

[11] 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.

[12] M. Egele, P. Wurzinger, C. Kruegel, 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*, pages 88–106, 2009.

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

[14] Google, Inc. Google safe browsing API. http://code.google.com/apis/safebrowsing/.

[15] M. V. Gundy and H. Chen. Noncespaces: using randomization to enforce information flow tracking and thwart cross-site scripting attacks. *Proceedings of the Annual Network & Distributed System Security Symposium*, 2009.

[16] B. Hoffman. Circumventing automated JavaScript analysis. In *Proceedings of Black Hat USA*, Las Vegas, 2008.

[17] 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.

[18] 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.

[19] M. Howard. Update on Internet Explorer 7, DEP, and Adobe software. http://blogs.msdn.com/michael_howard/archive/2006/12/12/update-on-internet-explorer-7-dep-and-adobe-software.aspx, 2006.

[20] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *Proceedings of the USENIX Windows NT Symposium*, pages 135–143, 1999.

[21] 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.

[22] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the USENIX Security Symposium*, pages 191–206, 2002.

[23] L. Meyerovich and B. Livshits. ConScript: Specifying and enforcing fine-grained security policies for Javascript in the browser. In *IEEE Symposium on Security and Privacy*, May 2010.

[24] Microsoft Corporation. Data execution prevention. http://technet.microsoft.com/en-us/library/cc738483.aspx, 2003.

[25] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. SpyProxy: execution-based detection of malicious web content. In *Proceedings of the USENIX Security Symposium*, Boston, 2007. ACM.

[26] 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*, San Diego, 2006. The Internet Society.

[27] Y. Nadji, P. Saxena, and D. Song. Document structure integrity: A robust basis for cross-site scripting defense. In *Proceedings of the Network and Distributed System Security Symposium*, 2009.

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

[29] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 226–241, 2005.

[30] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. C. Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd Symposium on Operating Systems Principles (22nd SOSP'09), Operating Systems Review (OSR)*, pages 87–102, Big Sky, MT, Oct. 2009.

[31] 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*, pages 87–106, 2007.

[32] 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.

[33] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iFRAMEs point to us, 2008. Available from http://googleonlinesecurity.blogspot.com/2008/02/all-your-iframe-are-point-to-us.html; accessed on 15 Feburary 2008.

[34] 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.

[35] C. Reis, J. Dunagan, H. J. Wang, O. Dubrovsky, and S. Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, Seattle, 2006. Usenix.

[36] M. Roesch. Snort - lightweight intrusion detection for networks. In *Proceedings of the USENIX conference on System administration*, pages 229–238, 1999.

[37] M. Ruef. browserrecon project, 2008. Available from http://www.computec.ch/projekte/browserrecon/; accessed on 20 August 2008.

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

[39] C. Seifert, R. Steenson, T. Holz, B. Yuan, and M. A. Davis. Know your enemy: Malicious web servers, 2007. Available from http://www.honeynet.org/papers/mws/; The Honeynet Project; accessed on 20 August 2010.

[40] Skape and Skywing. Bypassing windows hardware-enforced DEP.

Uninformed Journal, 2(4), Sept. 2005.

[41] C. Song, J. Zhuge, X. Han, and Z. Ye. Preventing drive-by download via inter-module communication monitoring. In *ASIACSS*, Beijing, 2010.

[42] R. J. Spoor, P. Kijewski, and C. Overes. The HoneySpider network: Fighting client-side threats. In *First*, Vancouver, 2008.

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

[44] T. Toth and C. Krügel. Accurate buffer overflow detection via abstract payload execution. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection*, pages 274–291, 2002.

[45] K. Wang. HoneyClient, 2005. Available from http://www.honeyclient.org/trac; accessed on 2 Janurary 2007.

[46] 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*, San Diego, 2006. Internet Society.

[47] 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.