ROZZLE: De-Cloaking Internet Malware

Clemens Kolbitsch Benjamin Livshits and Benjamin Zorn Christian Seifert
Vienna University of Technology Microsoft Research Microsoft

Microsoft Research Technical Report
MSR-TR-2011-94
October 25, 2011
**Rozzle: De-Cloaking Internet Malware**

Clemens Kolbitsch  
Vienna University of Technology  
Benjamin Livshits and Benjamin Zorn  
Microsoft Research  
Christian Seifert  
Microsoft Corporation

Abstract—In recent years, attacks that exploit vulnerabilities in browsers and their associated plugins have increased significantly. These attacks are often written in JavaScript and literally millions of URLs contain such malicious content.

While static and runtime methods for malware detection have been proposed in the literature, both on the client side, for just-in-time in-browser detection, as well as offline, crawler-based malware discovery, these approaches encounter the same fundamental limitation. Web-based malware tends to be environment-specific, targeting a particular browser, often attacking specific versions of installed plugins. This targeting occurs because the malware exploits vulnerabilities in specific plugins and fail otherwise. As a result, a fundamental limitation for detecting a piece of malware is that it is triggered infrequently, only showing itself when the right environment is present. In fact, we observe that using current fingerprinting techniques, just about any piece of existing malware may be made virtually undetectable with the current generation of malware scanners.

This paper proposes **Rozzle**, a JavaScript multi-execution virtual machine, as a way to explore multiple execution paths within a single execution so that environment-specific malware will reveal itself. Using large-scale experiments, we show that **Rozzle** increases the detection rate for offline runtime detection by almost seven times. In addition, **Rozzle** triples the effectiveness of online runtime detection. We show that **Rozzle** incurs virtually no runtime overhead and allows us to replace multiple VMs running different browser configurations with a single **Rozzle**-enabled browser, reducing the hardware requirements, network bandwidth, and power consumption.

I. INTRODUCTION

In recent years, we have seen mass-scale exploitation of memory-based vulnerabilities migrate towards drive-by attacks delivered through the browser. With millions of infected URLs on the internet, JavaScript malware now constitutes a major threat. A recent 2011 report from Sophos Labs indicates that the number of malware pieces analyzed by Sophos Labs every year in 2010 — about 95,000 samples — nearly doubled from 2009 [56].

While static and runtime methods for malware detection have been proposed in the research literature (e.g., see [17, 18, 48]), both on the client side, for just-in-time in-browser detection, as well as offline, crawler-based malware discovery, these approaches encounter the same fundamental limitation. Web-based malware tends to be environment-specific, targeting a particular browser, often with specific versions of installed plugins. This targeting happens because the exploits will often work on specific plugins and fail otherwise. As a result, a fundamental limitation for detecting a piece of malware is that it is only triggered occasionally, given the right environment; an excerpted example of such malware is shown in Figure 1.

While this behavior has been observed previously in the context of x86 malware [36, 39, 67], the traditional approach to improving path coverage involves symbolic execution, a powerful multi-path exploration technique that is unfortunately often associated with non-trivial performance penalties [11, 12, 23, 50]. As such, off-the-shelf symbolic execution is not a feasible strategy. In a brute-force attempt to increase detection rates, offline detectors often deploy and utilize a variety of browser configurations side-by-side. While potentially effective, it is often unclear how many environment configurations are necessary to reveal all possible malware that might be lurking within a particular web site. Conversely, many sites will be explored using different configurations despite the fact that their behavior is not environment-specific. As a result, this approach has significant negative implications on the overall hardware requirements, as well as power and network bandwidth consumption.

This paper proposes **Rozzle**, a JavaScript multi-execution virtual machine, as a way to explore multiple execution paths within a single execution so that environment-specific malware will reveal itself. Using large-scale experiments, we show that **Rozzle** increases the detection rate for offline runtime detection by almost seven times. In addition, **Rozzle** triples the effectiveness of online runtime detection. We show that **Rozzle** incurs virtually no runtime overhead and allows us to replace multiple VMs running different browser configurations with a single **Rozzle**-enabled browser, reducing the hardware requirements, network bandwidth, and power consumption.

```
1 var E5Jrh = null;
2 try {
3 E5Jrh = new ActiveXObject("AcroPDF.PDF")
4 } catch(e) { }
5 if(!E5Jrh)
6 try {
7 E5Jrh = new ActiveXObject("PDF.PdfCtrl")
8 } catch(e) { }
9 if(E5Jrh) {
10 lv = E5Jrh.GetVersions().split(".")[4].
11 split("=")[1].replace(/\./g,""),
12 if(lv < 900 && lv != 813)
13 document.write('<object classid="ShockwaveFlash.ShockwaveFlash.9")
14 .GetVariable("f" + "version").split(".")
15 } catch(e) { }
16 try {
17 var E5Jrh = 0;
18 E5Jrh = (new ActiveXObject("ShockwaveFlash.ShockwaveFlash.9")
19 .GetVariable("f" + "version").split(".")
20 } catch(e) { }
22 document.write('<object classid="clsid:d27cdb6e-ae...
23 width=100 height=100 align=middle><param name="movie"...
```

Fig. 1: Typical JavaScript exploit found in the wild that demonstrates environment matching.
ple execution paths within a single execution so that environment-specific malware will reveal itself. Rozzle implements a single-pass multi-execution approach that is able to detect considerably more malware without any noticeable overhead on most sites. The goal of our work is to increase the effectiveness of a dynamic crawler searching for malware so as to imitate multiple browser and environment configurations without dramatically reducing the throughput.

A. Contributions
This paper makes the following contributions:

- **Insight.** We observe that typical JavaScript malware tends to be fragile; in other words, it is designed to execute in a particular environment, as opposed to benign JavaScript, which will run in an environment-independent fashion. We experimentally demonstrate that as a metric, fragility highly correlates with maliciousness in Section II.

- **Low-overhead multi-execution.** We describe Rozzle, a system that amplifies other static and dynamic malware detectors. Rozzle implements lightweight multi-execution for JavaScript, a low-overhead specialized execution technique that explores multiple malware execution paths in order to make malware reveal itself to both static and runtime analysis.

- **Detection effectiveness.** Using a collection of 65,855 JavaScript malware samples, 2.5% of which trigger a runtime malware detector, we show that Rozzle increases the effectiveness of the runtime detector by almost a factor of seven, enabling detection of 17.5% of the samples. We also show that Rozzle increases the detection capability of static and dynamic malware detection tools used in a dynamic web crawler, increasing runtime detections in that case over three-fold. When used for static online detection, Rozzle finds an additional 5.6% of malicious URLs.

- **Runtime overhead.** Using a collection of 500 representative benign web sites, we show that the median CPU overhead is 0% and the 80th percentile is 1.1%. The median memory overhead is 0.6% and the 80th percentile is 1.4%. The average overhead is slightly higher, because of a few outliers: the CPU overhead averages 10% and the memory overhead of using Rozzle is 3% on average.

- **Malware roulette.** We outline attack strategies that are not detectable with the current generation of static and runtime malware detection tools and use these attacks as a motivation and a quality bar for Rozzle’s design.

B. Paper Organization
The rest of the paper is organized as follows. Section II gives some background information on JavaScript exploits and their detection. Section III gives an intuitive overview of Rozzle. Section IV describes the implementation of our analysis. Section V describes our experimental evaluation. Section VI proposes ways to design more powerful attacks that would not be uncovered with the current generation of malware detection tools. Section VII discusses the limitations of Rozzle. Section VIII discusses related work, and, finally, Section IX concludes.

II. Background
In the last several years, we have seen web-based malware experience a tremendous rise in popularity. Much of this is due to the fact that JavaScript, a type-safe language, can be used as a means of mounting drive-by attacks against web browsers. A prominent example of such attacks is heap spraying [53, 59], where many copies of the same shellcode are copied all over the browser heap before a jump to the heap is triggered through a vulnerability in the browser. This exploitation technique showcases the expressive power for a scripting language, since copying of the shellcode is typically accomplished with a single for loop. Our experience with Nozzle [48], a runtime heap spraying detector and Zozzle [18], a static JavaScript malware detector, suggests that there are millions of malicious sites containing heap spraying as well as other kinds of JavaScript-based malware, such as scareware [22]. Previous reports point out to the prevalence of JavaScript malware cloaking [4, 14, 65]. Our experience indicates that various forms of cloaking, environment matching, or fingerprinting are virtually omnipresent in today’s JavaScript malware. In fact, as we discovered, the degree to which a particular piece of code depends on the environment in which it runs — code fragility — is an excellent indicator of maliciousness; most benign code is environment-independent, whereas most malicious code does at least some cloaking, environment matching, or fingerprinting. The rest of this section is organized as follows. Section II-A presents a simple running example to motivate our discussion. Section II-B discusses several commonly used environment-specific malware practices. Section II-C presents an experiment where we show that code fragility correlates highly with code maliciousness.

A. JavaScript Malware: An Example of Real-Life Malware
To give the reader a better understanding of individual problems our system has to address, we start with an example of a piece of browser fingerprinting code found in the wild. A cleaned up version of this example, shown in Figure 3, employs precise fingerprinting to deliver only selected exploits that are most likely to successfully attack the client browser. Lines 1–40 compile the portion of the fingerprint that records the presence of the Adobe Acrobat, Quicktime, and Java plugins. Lines 42–42 record the presence of the Windows Media Player. Lines 54–55 construct the fingerprint string variable and lines 58–77 augment it with the browser language. Finally, line 81
function killErrors() { return true; }

window.onerror = killErrors;

function jc() {
    jc_list = [...]; // list of image locations
    ischeck = 1;
    x = new Image();
    x.src = "";
    x.onerror = function() { ischeck = 0; }
    x.src = jc_list[i];
    if (ischeck == 1) return 1;
    delete x;
}

return 0;
}

if (!jc()) {
    var oop="sk";
    // inject malware if not crawler
    document.writeln("<iframe src=5.htm width=100 height=1></iframe>);
}

Fig. 2: Client-side cloaking designed to avoid dynamic crawlers.

issues a request to a malware hosting site to fetch the malware that corresponds to the computed fingerprint.

B. Current Practices: Matching, Cloaking, Fingerprinting

We distinguish between three loosely defined categories of techniques commonly used in today’s malware: environment matching, fingerprinting, and cloaking, described in turn below.

Environment matching: Figure 1 shows a typical example of environment matching, found in most of the malware we find in the wild. In this case, the script determines the capabilities of the browser and selectively alters the content of the page, such as showing a movie.

Fingerprinting: Browser fingerprinting is a technique in which a variety of environment variables are evaluated to assess the capabilities of the browser. In contrast to environment matching, browser fingerprinting is more comprehensive and detailed in its assessment. Privacy advocates show that browser fingerprinting can be used to track users across sessions without the help of cookies as browsers carry unique information that results in unique fingerprints [20, 41]. Malware writers also use fingerprinting, as illustrated in Figure 3, to deliver malware customized for a particular browser configuration or, perhaps, even in the case of targeted attacks, for a particular user.

Cloaking: Offline malware scanning is used routinely to compile black lists of malicious URLs [47, 48]. In this scenario, cloaking can be successfully used by malware writers to avoid being detected when the malware-detecting crawler visits a particular site. We distinguish between server-side cloaking, which often operates by treating certain categories of HTTP headers or IP addresses, such as those coming from security vendors, differently, thereby avoiding detection, and client-side cloaking, which implements cloaking using JavaScript. Figure 2 shows an example of client-side cloaking we found in the wild. In this case, the crawler may not load images to save on processing times and network bandwidth. This fact is used by the code in this example to detect the crawler in function jc. If some of the images in list jc_list cannot be loaded, the error handler is called, which sets ischeck to 0. If function jc returns 0, an iframe pointing to malware is created.

C. Code Fragility Experiment

An observation that we make on the basis of our experience with malware is that environment-dependent code is often malicious. In this section, we give the reader an understanding of the prevalence of environment sensitive JavaScript code.

Using a simple ad hoc static analysis tool designed to process JavaScript abstract syntax trees (ASTs), we experimentally analyze the frequency with which both benign and malicious sites get access to environment-specific data that could be used to identify the browser,
a specific browser-version, installed plugins, or even the underlying operating system, or CPU architecture. We conclude that, indeed, code fragility is an excellent measure of maliciousness.

**Fragility detection tool:** To help with evaluating our hypothesis, we have constructed a relatively simple static analysis tool for determining what conditionals (ifs) in JavaScript code are environment-dependent. The tool works by statically tainting values [60] that are dependent on the navigator object and its fields and values that come from ActiveXObject calls. Taint is conservatively propagated through unary and binary string operations such as trim and string concatenation, as well as assignments.

**Experimental results:** We start with a set of 38.9 million JavaScript code snippets, representing all JavaScript presented for execution, from 2.8 million unique URLs. The set contains 2,373 JavaScript files that were flagged by Zozzle [18] and 194 files flagged by the fragility detection tool classifier mentioned above. These 194 are a strict subset of the 2,373. A summary of this data is presented in Figure 4.

The figure shows the number and fraction of files that have particular characteristics. The three main columns, “All”, “Malicious” and “Fragility Detector”, show the fractions of the total with respect to the different subsets mentioned above (e.g., flagged by Zozzle as malicious and flagged by our fragility detector). The “Reference” row shows those files where the navigator object or plugins are explicitly referenced. The “Branch” row shows those files where conditional expressions are based on the values of the navigator object or the presence or versions of plugins. The remaining rows break out the branches into the number of uses of specific fields of navigator and other environment-related variables.

We highlight our observations based on this data below:

- Only 7.7% of all JavaScript files reference environment-specific values. This fraction provides an estimate of the fraction of files would require multi-execution to expose potential malicious behavior.
- In 1.2% of all files, there is a branch on a symbolic value. Because branches require explicit action during multi-execution, these files will incur an additional cost in Rozzle.
- We observe that 98.8% of malicious files (as flagged by the Zozzle classifier) reference the JavaScript environment, 89.5% get a reference to something we would treat as symbolic (the difference is ActiveX-XML-RPC objects, and document and window objects). The same 89.5% of malicious files will branch on these conditions.

The analysis above provides strong justification for our intuitive understanding: exploits are environment-dependent.

### III. Overview

Section III-A covers existing techniques and outlines their shortcomings. Section III-B describes the basics of Rozzle. Finally, Section III-C provides a detailed example of multi-execution.

#### A. Challenges and Existing Techniques

While static analysis is a powerful technique that allows one to explore all program paths, a particular issue that plagues static analysis in the context of malicious...
JavaScript is that we are unable to observe all code. The script shown in Figure 2, for instance, selectively loads exploit content only when images are successfully loaded effectively hiding the exploit content from static analysis. Runtime evaluation has been advocated in this context [18, 27], but runtime execution suffers from the issue of low path coverage. A specific example is JavaScript malware that is triggered only when the user hovers over a particular UI element. This malware would generally not be exposed in the context of offline detection otherwise. A number of approaches to improve runtime path coverage exist, as detailed in the rest of this subsection.

Large-scale distributed setup: Machine clusters running different environment configurations are traditionally used for offline malware scanning, detection, and analysis. There are a number fundamental problems with this approach, however.

- **Scalability and inefficient use of resources.** While it is feasible to deploy a number of machines using different operating systems as well as browser manufacturers or versions, there are a number of other factors that need to be considered. For example, the availability of certain add-ons (such as Adobe Flash or the Java runtime) can have a great impact on how a browser renders or interacts with a remote server. Clearly, the combinatorial growth of possible plugin version/browser/browser version combinations in practice pretty much dictates the use of most popular environment configurations. In practice, this approach linearly expands the requirements on scanning hardware, network bandwidth, and power.

- **Overkill.** In order to detect malicious pages that selectively target a particular type of browser necessitates re-scanning the same page. As show in the previous section, only a very small fraction of sites found today make use of environment fingerprinting. Thus, deploying large clusters of computers re-scanning the same page and getting the same result is highly unprofitable and constitutes a waste of resources.

- **Server load.** Since multiple re-scans of the site are necessitated by this approach, load on analyzed web servers is increased. Note that we cannot cache server responses, as they might be user agent-specific. This may lead to the server refusing to accept connections from our offline scanner.

- **Incomplete attack surface.** Any pre-defined browser setup can only handle a known set of browsers and plugins. Thus, there is no guarantee that this setup will detect vulnerabilities in less popular plugins that could be used in targeted attacks against a small group of victims using known browser configurations.

**Full symbolic execution:** More recently, researchers have tried applying techniques of symbolic execution [11, 12, 23] to the task of exposing malware [9, 39]. This approach, while increasing the coverage, suffers from scalability challenges and is, in many ways, unnecessarily precise. Indeed, with a very precise runtime or static detector, malicious behavior is so uncommon that the issue of feasible paths is a relatively small concern.

### 4. Architecture and Overview

**Rozzle** is an enhancement or amplification technology, designed to improve the efficacy of both static and runtime malware detection. Figure 6 summarizes how existing malware detection techniques are affected by avoidance strategies in Section II-B and how Rozzle improves existing detection techniques. Rozzle is effective at improving both static and runtime detection. However, Rozzle is helpless at avoiding server-side cloaking.

**Multi-execution explained:** The key idea behind Rozzle is to execute both possibilities whenever it encounters control flow branching that is dependent on the environment. For example, in the case of the if statement shown in Figure 7, Rozzle will execute both branches, one after another. Some readers might wonder if this creates a dependency on the order in which Rozzle will executing the then and the else branch. A key insight is that in this case we need to perform weak updates. In other words, the second assignment to variable shellcode does not override, but adds to the first value. This is like using gated SSA form [58] in optimizing compilers, except in the case of Rozzle, SSA construction happens at runtime.
if (navigator.userAgent=="safari") {
    shellcode = unescape("1...");
} else {
    shellcode = unescape("A...");
}

Fig. 7: Simple example of the use of symbolic values.

**ROZZLE architecture:** Figure 5 provides an overview of the ROZZLE architecture. ROZZLE augments the semantics of a regular JavaScript interpreter by introducing additional statements in the execution that correspond to multiple symbolically executed paths. The second stage concretizes output statements for symbolic values according to a concretization policy \( L \).

**C. Detailed Example of Multi-Execution**

To build-up the reader’s intuition, we now show a more involved example of how ROZZLE handles real-life code. Figure 8 provides an illustrative example of multi-execution in action on a simplified code excerpt extracted from the fingerprinting routine in Figure 3. Figure 8(a) shows the original program. On line 10, we output the computed value `qt_plugin`. Figure 8(b) shows the evaluation function computed by ROZZLE to symbolically represent the computed result of `qt_plugin`. Note that the evaluation function is parameterized with the `navigator` object, whose `plugins` array is used in the function code. Conditionals in the evaluation function correspond to conditional statements in the original program. While this is outside the scope of this paper, note that evaluation functions may be analyzed entirely statically using one of the proposed approaches in the literature [6] to determine all potential outputs, to determine which inputs may lead to a particular output. Finally, Figure 8(c) shows the symbolic value the way it is represented by ROZZLE. Once again, the symbolic evaluation tree directly matches the structure of the evaluation function in Figure 8(b), with leaves contributing the potential values of the output, which are either the result of evaluating

\[
\text{parseInt(name.replace(/\D/g,""))}.toString(16)
\]

or "0".

**IV. TECHNIQUES**

This section focuses on the details of multi-execution, covering both the fundamental principles and the details of ROZZLE implementation on top of the Chakra JavaScript engine in IE 9. Because of the amount of technical detail, it might be skipped on first reading and returned to afterwards. This section is organized as follows. Section IV-A describes how we construct and manipulate symbolic values. Section IV-B elaborates challenges faced with a naïve implementation of multi-execution. Section IV-D discusses “concretizing” symbolic values on-demand. Section IV-E discusses the details of multi-execution in ROZZLE. Finally, Section IV-F talks about our implementation of ROZZLE built on top of IE 9.

**A. Symbolic Values**

At the core of the ROZZLE approach is the idea to treat some heap values as `symbolic`. This is a departure from traditional symbolic execution approaches: for example, in Sage [23], numerous program traces are considered, one after another, which correspond to different program paths. In ROZZLE, similar exploration is achieved through executing multiple branches in the course of a single modified execution, using symbolic heap values to reflect multiple program outcomes. For example, the merge of two versions of `shellcode` in Figure 7 gives rise to a
symbolExpr ::= numExpr \| stringExpr \| boolExpr
\| memberCall \| funcCall \| *
\| condExpr \| concretize(symExpr)

condExpr ::= boolCond \& symExpr \| symExpr

binaryExpr ::= symExpr binaryOp symExpr

memberCall ::= symExpr \. memberCall

funcCall ::= symExpr \( paramExpr \)

paramExpr ::= symExpr \| symExpr \, paramExpr

boolCond ::= boolExpr \& ~boolExpr
\| symExpr \& symExpr
\| symExpr \&\& symExpr

boolExpr ::= true \| false
\| symExpr boolOp symExpr
\| isSymbolic(symExpr)

numericFunc(symExpr) ::= numericFunc(symExpr) \| 1 \| 2 \| ...
\| numExpr binaryOp numExpr
\| parseInt(stringExpr)
\| indexOf(stringExpr, stringExpr)
\| abs(numExpr)
\| min(numExpr, numExpr)
\| ...

stringFunc(symExpr) ::= stringFunc(symExpr) \| "" \| ...
\| encodeURI(stringExpr)
\| decodeURI(stringExpr)
\| subtr(stringExpr)
\| concat(stringExpr, stringExpr)
\| replace(stringExpr, stringExpr)
\| replace(/stringExpr/, stringExpr)
\| toString(numExpr)
\| toString(numExpr, numExpr)
\| ...

binaryOp ::= + \| - \| * \| / \| % \| << \| >>

boolOp ::= = \| \ne \| < \| <= \| > \| >=

Fig. 9: BNF for symbolic expressions used in Rozzle. The start symbol is symExpr.

that it is wrapping (e.g. a userAgent string) as well as the current concrete type. Initially, each symbolic wrapper has the runtime type of the wrapped object (e.g. string when wrapping a navigator.userAgent string value).

Marking values as symbolic: All environment-specific values start out as symbolic in Rozzle. For example, navigator.userAgent is treated symbolically, whereas the string "0" is not. This is quite similar to the notion of runtime tainting [60]. In Rozzle, taint originates with the fields of the navigator object. Additionally, we mark as symbolic the results of functions ActiveXObject, ScriptEngine, ScriptEngineMajorVersion, ScriptEngineMinorVersion, and ScriptEngineBuildVersion in the engine.

B. Challenges

While the basic idea of maintaining symbolic values on the heap is straightforward, any implementation must address some fundamental challenges, including the following:

- **Looping on a symbolic value:** When looping on a symbolic value, how many iterations do we need to perform?
- **Writing symbolic values to the DOM:** Symbolic values represent multiple concrete ones, so which of the concrete values do we write out to the DOM?
- **Output operations on symbolic values:** What if a symbolic value is used to compute the URL that the program is reading data from? How do we make these network requests concrete? Do we consider all of them?

- **Limiting the size of symbolic values within the heap:** When representing the symbolic heap naively, there is a very real possibility of running out of memory, because of the extra context provided by the symbolic values.

- **Introducing errors:** Rozzle may introduce new errors in correct code. One key reason is that Rozzle may expose differences in browser implementations, executing paths that would otherwise be infeasible. Other reasons exist as well, including running out of time, memory, or resources, making too many outside connections and being blocked by third-party servers, throwing exceptions, etc. We discuss errors that Rozzle may introduce in Section VII-A.

C. Symbolic Values: Manipulation and Representation

Figure 9 summarizes a grammar that captures symbolic values that may be created by Rozzle at runtime. We provide this in the form of a BNF grammar where symExpr is the start symbol; symbolic value trees that are created at runtime can be seen as parse trees for expressions in this grammar. Grammar elements such as condExpr, numericExpr, or stringExpr give rise to intermediate tree nodes, as shown in Figure 8(c). Elements memberCall and funcCall are slightly more complicated. Whenever there is
a call to a property of the object (i.e., a member function), we need to check if the current concrete type supports this
method. If so, the output is a symbolic object representing the result of calling the function on the given object. This
produces an AST of symbolic objects where each node in the tree contains a function and sub-ASTs for each call parameter.

**Depth limiting:** When creating new symbolic values, we are careful to limit the depth of resulting symbolic trees.
One of the common reasons this comes up is because of symbolic values updated in a non-symbolic loop, which
leads to the creation of unbounded nested trees. Our solution is to collapse the entire tree the moment its depth
exceeds a fixed threshold and represent it as a special symbolic value ⋆.

**Symbolic value compression:** A challenge that we have
to address when dealing with large JavaScript programs
that have a lot of tainted branches is to keep the size of
symbolic value trees small. Our approach to reducing the
memory footprint of Rozzle involves using a canonical rep-
resentation for data structures used to represent symbolic
values, in a manner similar to decision diagrams [10], etc.
This way, symbolic values will share some of the subtrees,
as illustrated with the triangle in Figure 8. In Rozzle, we
keep a pool of allocated symbolic values and, whenever
creating a new conditional value, consult the list to see if
the sub-components of the conditional are already found
in the pool. Comparisons against existing pool elements are
very fast and are currently done via a depth-first explicit
comparison; an alternative involves using hashing and a
lookup table for the same purpose.

**D. Resolution of Symbolic Values**

While the use of symbolic values in Rozzle allows us to
explore more code paths than can be observed through a
single concrete execution based on one particular en-
vironment, there exist cases where we need to obtain
contcrete values in the JavaScript engine from symbolic
values constructed by Rozzle. Typically this happens,
when an object is passed from the JavaScript engine to
another browser subsystem (e.g., during modifications
of the DOM, when browsing to a new URL, requesting
new content from the web, etc.). Figure 8(c) shows a
graphical representation of a variable in symbolic memory.
When Rozzle requires concrete values (e.g., when passing
a symbolic variable outside the JavaScript engine), it
traverses the tree in memory and generates code as seen
in Figure 8. Our system uses predefined profiles: Rozzle
generates code that iterates over the given sets, using the
generated formula to resolve concrete values. These values,
together with a label that identifies the underlying profile,
are stored as concrete value candidates.

**Handling the DOM and IO:** Concretization is required
when dealing with various IO subsystems within the
browser. Most common examples include writing symbolic
values to the DOM and using symbolic values as parameters
of a network request. Currently, we take the simple
approach of just extracting the first of potentially many
concrete values out of a symbolic one. This, of course, is
an implementation choice. When dealing with the DOM,
it is possible to cache symbolic values before they cross
the JavaScript engine/DOM divide and then, whenever a
value is returned back from the DOM, check it against the
cache to return the symbolic version back to the JavaScript
engine if necessary; this is the approach previously used
in the ConScript project [34]. In the case of network
requests being symbolic, we limit the requests to the first
concrete value as well. An alternative would be to create
multiple (cloned) browser instances or tabs that would
continue executing in parallel.

**Local focus:** Weak updates can lead to an unnecessary
loss of precision. To understand why, consider the following
loop:

```javascript
if(navigator.userAgent.indexOf("safari") > 0){
    for(i=0; i<5000; i++){
        // ignore the undefined because path
        // predicate matches the symbolic value predicate
        // i = is_safari ? 0 : undefined;
        memory[i] = nop + nop + shellcode;
    }
}
```

Naïvely, Rozzle would treat assignments to variable `i`
symbolically, because the loop increment is considered
to be an assignment that is control dependent on the
outcome of the `if`. However, for the special case of the
path predicate matching the predicate of the conditional
symbolic value, the other alternative, `undefined`, is projected
away, and Rozzle in this case will treat loop variable `i`
non-symbolically. This change to the default strategy is
actually quite important because treating this loop sym-
bolically means that we are not executing the loop 5,000
times and are therefore not going to be flagged at runtime
by Nozzle for attempting a heap spray attack. This form
of special-casing is akin to the notion of focus used to
obtain locally precise treatment in static analysis [21].

**E. Details of Multi-Execution**

Figure 11 shows pseudo code for our multi-execution
engine. The inputs of the algorithms are program `P`, which
is a collection of statements `stmt1, . . . , stmt_n`, browser
profile `π`, an output policy `L`, and a side-effect hash map
`mod`.

---

**Fig. 10:** Statements in our program representation.

Statements

\[
stmt := \begin{cases} 
\text{var} = \text{symExpr} & | \text{var.field} = \text{symExpr} \\
\text{if (symExpr)} & | \text{else stmt} \\
\text{while (symExpr)} & | \text{stmt} \\
\text{symExpr} = \text{symExpr}(\text{symExpr}, . . . , \text{symExpr}) \text{stmt} \\
\text{output (symExpr)}
\end{cases}
\]

---

**Fig. 11:** Pseudo code for our multi-execution
engine.
The profile contains specific details of the environment such as the userAgent string, the plugins array and the data accessible from it, the major and minor version of the JavaScript engine, etc. The output policy decides how to concretize a symbolic value at an output statement by potentially expanding it into multiple trace statements.

The output policy L defines how to concretize a particular value e at output statement i given profile π, and the mod hash map. Many policies exist, including concretizing e with respect to π, sequentially outputting all concrete values in e, etc.

Function isSymbolic is a runtime check that returns whether the value passed in should be treated symbolically. The algorithm in Figure 11 consists of an interpreter loop that handles the cases of an if conditional, a loop, etc. in turn.

Branching on symbolic values: In cases where the code branches on a symbolic value, we need to make a decision which branch to take. In traditional symbolic execution, the framework will consider both outcomes, one at a time, and check if either is feasible using a theorem prover to validate the path condition. In ROZZLE, we execute both cases. We do this, by maintaining a symbolic stack of conditions that must be fulfilled to reach the current point in the execution. Considering Figure 12, the if/else block would have an active symbolic value of fingerprint.indexOf("IE") >= 0 and any variable assignment within this block will need to respect this condition. Thus, when assigning to either a variable or a heap object of the form object.field outside this block, it will be made into a conditional symbolic value. Before executing the else branch, the active element on the symbolic-condition stack is inverted. Assignments to variables are merged when one sees that it is a reference to a variable that is already conditional on a symbolic. This means, after executing the above block, variable isIE would hold

isIE = (x.indexOf("IE") >= 0) ? true : false;

Note that this form of multi-execution when both branches are followed is only performed with the conditional is symbolic, which in practice happens quite rarely for benign programs, so we are not going to see a significant increase in the running time. As mentioned before, in ROZZLE, we execute branches that are dependent on symbolic variables sequentially, one after another. To support weak updates on such code paths, we proceed as follows: Whenever a branch is encountered and ROZZLE finds that the condition is symbolic, the condition is pushed onto a stack used to keep track of path predicates. When executing the else-statement of a symbolic branch, the condition on the top of stack is inverted and used as condition for the new branch. The else block is handled by combining the element on stack with the new condition. When leaving the symbolic branch (i.e., after executing the last branch conditioned under the symbolic predicate), the symbolic condition is popped from the stack. Within a symbolic branch, weak updates are used for both variable assignments and heap object stores. For this, the current path condition (i.e., the conjunction of all elements of the path predicate stack) is used to build the tree of symbolic memory as described above. The pseudo code for handling conditionals is shown in lines 3–28 of Figure 11.

Looping on symbolic values: Handling symbolic loops

Fig. 11: Algorithm for multi-execution which takes program P, profile π, and output policy L as inputs. The last parameter Γ is an optional in-out hash map that represents the side effects of calling MultiExecute; the default value for Γ is globalObject; that is typically the same as the window object in JavaScript.
var x = fingerprint;
var isIE;
if (x.indexOf("IE") >= 0) {
  isIE = true;
} else {
  isIE = false;
}

Fig. 12: Symbolic execution: a simple if.

allows us to check the type of a variable at runtime. In our scenario, this allows an attacker aware of Rozzle to detect and avoid execution inside our system. Although such code would be very indicative of malicious behavior if detected, we tackle this problem as follows: if the type of a variable of type symbolic is queried or compared to another type using the typeof keyword, Rozzle resolves the type that most closely resembles the given variable (e.g., for a symbolic variable holding the navigator.userAgent, the system returns a string).

Function calls: Symbolic value may be passed into both the JavaScript language and the DOM API functions exposed by the JavaScript engine. An example of this is are concat and indexOf functions on strings. We need to augment natively implemented function in Chakra to first check if any of the parameters is symbolic and to return a properly constructed new symbolic value if that is the case. While this might sound like a considerable amount of manual work, fortunately, as most parts of the API are written natively in C, we only need to insert a single macro into the prologue of each function.

Virtual branching and try/catch blocks: Exceptions provide a common way for the attacker to test the capabilities of the environment in which their code executes. As such, we need special handling of a particular way of testing the availability of certain features in JavaScript that happens often in malicious code. Consider the following commonly found example shown in Figure 13. We handle this by introducing “virtual if-blocks” after the allocation of symbolic values. We do this by pushing a special condition onto the condition stack after the allocation of an object that might not exist, treating the try block as the virtual then and the catch block as the virtual else. After the if-block, we execute the catch block and inverting the active condition (just like in the else block case). This would lead to a symbolic expression such as

\[
\text{hasPDF} = (\text{has\_activeX\_support\_pdf}) ? \text{true} : \text{false};
\]

where \(\text{has\_activeX\_support\_pdf}\) is a special variable that parameterizes the environment.

V. Evaluation

In this section, we evaluate the usefulness of Rozzle.

We compare two configurations: base, which is a standard browser without multi-path execution, and Rozzle, where multi-path execution is enabled. We compare these configurations in our experiments: in Section V-A, we analyze...
the impact of our multi-path execution on our runtime detector using a set of known malicious JavaScript files. In Section V-B, to understand the degree in which the system exposes new web-based malware on the Internet, we extend an existing high-interaction client honeypot with Rozzle and analyze live URLs. Finally, in Section V-C, we measure Rozzle’s impact on the high-interaction client honeypot in terms of memory as well as runtime overhead.

A. Improved Offline Detection Rates with Rozzle

To understand if Rozzle is able to extract new runtime behavior in real malicious scripts, we selected a set of 65,855 web-based malware samples found in the wild using the Zozzle static malware detector in combination with a high-interaction client honeypot on a large cluster of machines.

Setup: In this experiment, we take special care to minimize the degree to which external influences could affect the outcome, such as site availability or modifications of the exploits. For this, we extracted the JavaScript context flagged by Zozzle and hosted the file on a server on our network, thus the name offline experiments. For this experiment we placed the files on a local disk and we visited each file as a local URL using the high-interaction client honeypot twice, once using its default configuration using an Internet Explorer 9 profile (base run) and a second time using the Rozzle-extended version. As the client honeypot renders and executes the page content, it scans any JavaScript contexts found using Zozzle and also uses Nozzle to detect any suspicious behavior during the execution of the scripts.

Results: Figure 14 shows the detection rates using our dynamic detector, Nozzle, during the visits of our high-interaction client honeypot. We do not include static detection by Zozzle, as all scripts have previously been detected by the latter. The figure shows an overview of the detection results: Nozzle triggered in 1,662 cases total in the base run while the second run using Rozzle flagged 11,559 URLs (595.5% more detections). In 1178 cases (70.9%) the URL was flagged in both runs, leaving 484 URLs (29.1%) where Rozzle introduced an error into the script’s execution before it was flagged. These results provide valuable information to improve our prototype implementation in the future and we discuss possible reasons for these errors in Section VII-A. In contrast to the 484 errors introduced, we see that Rozzle is able to expose new malicious dynamic activity in 10,381 cases. This clearly demonstrates two things: first, Rozzle is successful in increasing path coverage using multi-path execution. Second, environment-sensitive malware is a real problem and Rozzle is able to expose it.

B. Improved Online Detection Rates with Rozzle

In a second experiment, we collect and visit a set of URLs hosted on the Internet. This allows us to test if Rozzle is able to extract previously unseen content that might be detected using either of our detectors. Early on in the evaluation, we found that this is not an easy task as malware hosting servers are quite unreliable. As described in [69], we encountered a large number of malicious URLs where attacks were served only once to a given IP address/subnet or only once within a given time frame (e.g., using a cookie-check). This creates the problem that depending on which configuration is used first (base versus Rozzle), that configuration may see more malware simply because the site fails to serve the malware a second time. To make our estimates of Rozzle’s effectiveness conservative, we visit each URL with the base configuration first. Thus, if the exploit is not served during the second run (using Rozzle), we might mistake this as an error introduced by Rozzle, but not as a new detection. Further, we manually verified a large fraction of the analysis results where we see a difference in the two runs, excluding those are caused by a clear difference in the content served by the server.

Setup: For this experiment, we obtained a large set of suspicious URLs from static analysis of web crawler content. Because our experimental resources were limited, we applied blacklist-based filtering to increase the likelihood of visiting URLs hosting malware: Each URL was checked against a list of hosts known to serve malicious content as well as using Google’s SafeBrowsing API. For 57,132 URLs (approximately 0.1% of the initial list), at least one of the checks succeeded and we visited the URL with our two experimental configurations. In this experiment, we wanted to determine the effect of Rozzle on both static and dynamic malware detectors, so we enabled both Zozzle and Nozzle detection in our browsers for both configurations.

Results: Nozzle and Zozzle improvement rates for these experiments are summarized in Figure 15(a). Our dynamic detector, Nozzle, flagged 74 malicious URLs with the base configuration and 224 using the Rozzle configuration. Similar to the results obtained during the offline evaluation, 24 (32.4%) of the base detections were not detected with the Rozzle configuration, but Rozzle enabled many more (174) new Nozzle detections. For the static detector, Zozzle, the results are somewhat different. In the base configuration, 2,735 URLs were flagged as malicious while using Rozzle only detected 2,660 malicious URLs. A total of 2,510 URLs were detected in both runs, with errors
on 225 URLs and 156 new ZOzzle detections in the second run. To better understand this result and test to what degree Rozzle was able to reveal new detections, we manually verified a subset of URLs constituting 1,540/1,557 ZOzzle and 31/120 Nozzle detections in the base and Rozzle second (ROZZLE) runs, respectively. For Nozzle, 2/9 missed detections were caused by the server not serving an exploit during the second run, and are not the fault of Rozzle.

In the other 7 cases, handling of symbolic variables caused errors during script execution. We discuss the problem of error handling in the context of Rozzle in Section VII-A. For Rozzle, 60 detections were missing in the ROZZLE-enabled run. In 44 cases, the exploit was not served during the second analysis run and, like above, the failure to detect is not the fault of Rozzle. In 5 cases, our system caused an error during JavaScript execution, stopping the exploit from being unpacked or being downloaded and resulting in a missing ZOzzle detection. In additional 11 cases, the script caused an error at runtime. Although it is not clear whether Rozzle has any impact on these code snippets, we conservatively assume that the errors are caused by the multi-path execution. As we discard missing detections in the repeated analysis runs, we also need to verify new detections when using ROZZLE to allow for a fair comparison. On this data set, we tried answering another interesting question: In cases where we see a new Nozzle detection, did we already cover this URL previously through a ZOzzle detection? Or, in other words, do we find previously undetected malware URLs that the system would have missed without ROZZLE? We found that in 90 cases, ROZZLE triggered a Nozzle detection whereas in the base run neither Nozzle nor ROZZLE detected the malicious script. Likewise, in 156 cases ROZZLE flagged the URL as malicious although it had not been detected by either system previously. Last, there are 76 new Nozzle detections that are not flagged by ROZZLE and 142 new ZOzzle detections that are not flagged by Nozzle. Figure 16 summarizes these results. One can see that ROZZLE was successful in finding new detections in 232 cases, and with both detectors contributing.

C. Performance Overhead of ROZZLE

Memory and CPU consumption are key factors in determining the number of client honeypots that can be run simultaneously on a single host. Thus, the overhead introduced by our system plays an important role.

Setup: To measure this impact, we conducted two experiments summarized in Figures 17(a) and 17(b). For the first experiment, we run a set of 500 randomly selected URLs from the list selected for the online evaluation described above. We chose this set of URLs as it is a representative sample of URLs that we expect to be using while running our system in the future. For the second experiment, we used SPIDERMONKEY to get a feeling for the performance impact of sites relying on heavy JavaScript computations. To measure the performance impact we proceed as follows: We added callbacks into the JavaScript framework to signal whenever a context is about to be executed and after execution has finished. This way we are able to directly see the slowdown caused by ROZZLE. Note that while we are only measuring the time within the JavaScript runtime, many other factors contribute to the overall performance, and the overall impact on the end-to-end throughput is significantly less than shown here.

For measuring the memory footprint of ROZZLE, we use a similar approach: we used hooks in the allocation of JavaScript objects as well as the garbage collector to be notified about allocation/de-allocation events. In both memory as well as runtime measurement, we analyzed each URL three times sequentially for each case and we report the minimum as well as average numbers. However, simply repeating analysis runs does not provide that reliable numbers: We noticed that script execution times varied heavily, often due to different content being loaded per visit (such as advertisements) or special cookie initialization code upon the first visit. To compensate for this, we visited each URL one additional time for script setup purposes and discarded this run from any calculation. Additionally, we counted the number of script contexts, number of function invocations, as well as unique functions called per URL. Whenever the ROZZLE-enabled run showed fewer script contexts or considerably fewer function invocations or called functions, we discarded the URL from analysis, assuming the site contained nondeterministic script inclu-
sions or ROZZLE caused execution to abort prematurely (and would thus skew results).

Results: Figure 17(a) summarizes results for the measured memory footprint, while Figure 17(b) shows the impact of ROZZLE on execution speed. The figures show distributions of observed overhead overheads on the y-axis, with the measured relative overhead plotted on the x-axis. X-axis numbers greater than 1 indicate slower execution and more memory. Note that the CPU overhead only measures the overhead of ROZZLE with respect to the IE9 Chakra JavaScript runtime. The performance of the rest of the browser remains unaffected. The figures both show that for the vast majority of web sites, the performance and memory impact of ROZZLE is 0, which matches our earlier measurements indicating that few websites make references to the environment or execute conditionally based on environment values. From these distributions, the median CPU overhead is 0% and the 80th percentile is 1.1%. The median memory overhead is 0.6% and the 80th percentile is 1.4%. We also see that in a small number of cases that there are performance outliers where the memory and CPU overhead is larger, up to a factor of two. As a result, the average overhead is slightly higher: the CPU overhead averages 10% and the memory overhead of using ROZZLE is 3% on average. Since the JavaScript runtime is only a fraction of the total CPU and memory overhead in a modern browser, we consider these performance results to be acceptable for offline malware scanning and also even potentially usable for in-browser scanning as well.

VI. ATTACK SCENARIO

In our research on ROZZLE, we have observed that while the majority of exploits we find in the wild are pretty naïve about detection avoidance, we do see examples of malware are increasingly difficult to detect without techniques presented here. One such example is shown in Figure 19; others are presented in Appendix A.

These examples inject the malware after fingerprinting the browser. While we currently see this technique used to deliver malware customized to the browser or plugin versions, it is easy to imagine similar fingerprinting used to avoid offline detection, as shown in Figure 2.

In this section, we present an approach to malware construction we dubbed malware roulette that is designed to avoid detection. Malware roulette can be thought of as a rewriting pass that may be applied to a given piece of malware to avoid detection.

The key principle is as follows: we take a JavaScript program represented as a set of functions with control flow graphs (CFGs) and convert each CFG into a network request with, request parameters encoding the state of the program at this point in the call graph.

Example 1 Specifically, considering a CFG with three basic blocks

\[ \begin{align*}
\text{BB1:} & \quad \text{if(navigator.userArent.indexOf("safari") > 0)}
\end{align*} \]

In this code, there are only two paths to consider: path 1 consisting of BB1 → BB2 → BB3 and path 2: BB1 → BB3. At the entry to BB2, we would issue a call to roulette.php?x = 3&y = 5&BB = 2, at the transition from BB2 to BB3, we would issue a call to roulette.php?y = 7&BB = 3, and at the end of BB3, we would issue a call to roulette.php?x = 2&BB = -1. In other words, the effect of each basic block is contained in the variables passed around to the subsequent call to roulette.php. The job of the roulette “driver”, roulette.php is to send the JavaScript code for the upcoming basic block to the client. □

These ideas can serve as a basis for an avoidance technique:

- In particular, we can make it so that the snippet that is shipped to the client in each case is quite small,
limited either by the size of a basic block, or even smaller, since we can break up lengthy basic blocks into smaller ones. For a static detector such as Zozzle, of course, the smaller the visible chunk is, the more challenging it is to get detected.

- Fingerprinting code will be made more difficult to detect as well. Currently, fingerprinting code is often grouped together such as in Figure 3, which makes it possible to train static classifiers to detect fingerprinting. Breaking this code up would hamper this kind of detection as well.

Example 2  To validate the feasibility of this technique, we have transformed one exploit found during our experiments with ROZZLE as shown in Figure 18.

The sample we use for this experiment performs three consecutive checks on the environment: First, the underlying browser is tested for Internet Explorer 6. Then, the code tries to instantiate ActiveX-plugins for handling PDF content from different vendors. Depending on the type of plugin that is available, the script uses different code for identifying the plugin version. Eventually, if this version matches a set of vulnerable candidates, a vendor-specific exploit is executed on the client browser.

The code paths to launching an exploit can be represented as a state machine (see Figure 18). States represent (sub-)queries on the environment (generation of a specific fingerprint), transformations are equivalent to checks (comparing a fingerprint against an expected value). Final states include launches of a specific exploit or 'no action', if the environment does not appear vulnerable (against any of the exploits available).

Using this state machine, we can re-write the example script into a series of redirects to hide from traditional client-side honeypots: In each state, a small script extracts the necessary information (e.g., the browser version) and submits it to the remote server. In turn, the attacker redirects the browser depending on the submitted data: If the extracted information does not fulfill any path constraints leading to a possible exploit state, the browser is redirect to a benign site to avoid raising suspicion. Otherwise, the attacker redirects to a server under her control to proceed in the roulette. Eventually, a server delivers the actual exploit to the requesting browser.

When visiting our system with a traditional client honeypot, the attacker can successfully hide her malicious intent in the majority of cases. This is because most code implementing the redirection logic is hidden on the server. Only if the honeypot implements the correct browser profile, it is able to reach one of the exploit states and see the actual exploit code (and detect that the site is malicious).

Executing the individual pieces of code in ROZZLE leads to a different outcome: As described above, fingerprint information collected within the individual states is treated as symbolic by our system. Thus, using it when building a request triggers resolving of concrete value candidates, and ROZZLE uses all available profiles to generate a set of request candidates. This way, the system is able to explore the state machine and eventually reach one of the exploit states.

We used our high-interaction honeypot described in Section V to test this assumption: As expected, without ROZZLE, the client was redirected to the benign site after a few checks (states). With the ROZZLE-enhanced version, however, we were able to navigate through the redirects and flag the servers as malicious.

VII. DISCUSSION

In this section we discuss limitations of ROZZLE, including ways that attackers can avoid it, as well as considering malware trends that are likely to impact other forms of static and dynamic detection.

A. Limitations

As with any detection tool, we need to consider ways that a determined attacker can avoid being detected by systems using ROZZLE. Avoidance approaches fall into three categories: hiding the decision making from the client, detecting that ROZZLE is being used and/or thwarting it, and avoiding the detection techniques that ROZZLE enhances. We consider each in turn.

Server-side cloaking: ROZZLE only protects against client-side cloaking attempts. It is not useful against server-side techniques such as IP black-listing, etc. In particular, a determined attacker can construct a fingerprint of the client-side environment and send it to the server, which in the response would direct the client in various ways depending on the configuration. Such behavior is itself quite suspicious (the server is unlikely to need to know all the details of the client configuration) and could perhaps be detected as potentially malicious.
Breaking existing code: It is not not entirely surprising that in certain circumstances Rozzle may lead to runtime errors because we execute infeasible paths. In particular, this often occurs when the program checks the user agent and then instantiates an object specific to that browser. While it is possible to provide mock-ups (or emulation) for this missing functionality, and we do that in a limited set of cases, our emulation is not exhaustive.

Similarly, aggressive execution of otherwise infeasible paths may lead to both an explosion of both the memory size because of the growth of symbolic values in the heap, as well as the time required to multi-execute the program, if sufficiently many nested conditionals are present. In practice the measurements in Section V show that these factors rarely impact performance, in part due to the symbolic value compression mentioned in Section IV-B.

Identifying that Rozzle is enabled: It is difficult to hide the fact that Rozzle is used within the browser, because of functional differences in execution as well as changed timing characteristics, etc. As a result, client code that detects the presence of Rozzle can avoid delivering the payload in that case. A specific case of this approach would be to construct a denial-of-service attack against Rozzle-enabled browsers: knowing the algorithm that Rozzle uses, an attacker could construct a program that caused Rozzle to run out of memory, CPU, etc.

B. Emergence of Better Malware Cloaking

While our results show that our current static malware detector is quite effective without Rozzle enhancement, we have also observed numerous cases of real malware that are resistant to static detection. Figure 19 shows a real-life example of malware that injects iframe-based payloads based on fingerprinting results computed on the client. While we hypothesized the existence of such malware and described the general approach its creation in Section VI, finding this malware in the wild validates our belief that such approaches need to be defended against. This example is one of the significant number of new runtime detections found with the help of Rozzle, as described in Section V. On lines 3 and 6, exploit code for IE 6 and 7 is included, respectively. Line 14 includes Flash-specific code. Finally, line lines 23 and 24 include code specific to ActiveX object GWC10.Spreadsheet. More examples of this kind can be found in Figures 20–25.

VIII. RELATED WORK

Given the prevalence of malicious sites on the Internet and the threat of malicious software in general, malware research has received much attention in recent years. In this section, we summarize related work and compare it to Rozzle.

A. Symbolic Execution

Symbolic execution was initially introduced by King [28] and has since been used in a number of different research areas such as program testing and malware analysis. Most work can be classified into techniques working either online (using symbolic execution at runtime) versus offline (executing a piece of code in a verbose manner and using symbolic execution to analyze the generated log-files).

Moser et al. [39] use online symbolic execution of native x86 code to detect new code paths in malware binaries. To this end, similar to our system, they use taint-tracing of environment-provided values (such as the current time) throughout the execution of the malicious binary and record execution of conditional paths that depend on these values. At a later point (e.g., when the sample terminates), they revert to a particular state in the execution and choose to execute a previously untaken conditional branch, updating all variables that are influenced by the inverted branch condition. Concurrently with Moser et al., Brumley et al. propose Minesweeper [8], a tool to detect trigger-based behavior in malware binaries. Here, the authors describe a similar approach to extract conditions from native code when certain code paths are executed. Rozzle works in a completely different environment (JavaScript vs. native x86 code) and, thus, requires far greater power to reason on symbolic constraints. For example, the system proposed by Moser et al. is limited to resolving constraints that have a linear dependency on tainted values to assure a consistent state update when reverting. Our system supports arbitrary data- and control dependencies, as well complex string manipulations, which are essential in the context we operate in. In addition, our system does not depend on reverting to a particular state in the execution trace to improve code coverage and instead tries to execute all possible paths within a single execution trace. This provides more efficient analysis and avoids some of the problems related with space explosion.

Other research has concentrated on offline symbolic execution. Here, a program is executed using (potentially
Fig. 21: Precise fingerprinting of the PDF plugin and its version. Conditional code inclusion for PDF and Java exploit is shown.

Fig. 22: Precise fingerprinting of the PDF plugin and its version. Conditional code inclusion for PDF and Java exploit. There is additional checks for Safari that don’t seem to be used at the moment.

multiple) inputs and the execution trace is recorded. Based on these traces, systems have been proposed for a wide variety of purposes: Godefroid et al. introduce SAGE [23] and use symbolic execution to generate hostile inputs used in fuzz testing that explore previously unhandled code paths. Cadar et al. propose a similar system (EXE [12]) that finds programming errors and automatically generates inputs that trigger the buggy code region. Costa et al. [16] analyze buggy programs to find control-flow conditions that lead to a successful attack. Brumley et al. use symbolic execution to extract formulas representing different protocol implementations [7]. These differences can indicate an incorrect protocol implementation or raise opportunity to fingerprint a remote application. In [1], Averinos et al. extend KLEE [11] and propose a system for automatically finding programming errors in binary programs as well as generating a working exploit. Similar to KLEE, other systems have proposed using symbolic execution for finding bugs in software [38, 45, 61, 66, 68].

ROZZLE differs from these systems as it is designed to be used in an online, real-time manner. Thus, it is limited in resolving the feasibility of conditions, which typically takes a considerable amount of time. Instead, we have invested much effort into supporting parallel, multi-profile execution using conditional memory. Only in situations where we require concrete values, we resolve conditional memory on demand.

B. JavaScript Analysis

With the growing popularity of client-side in-browser applications, JavaScript analysis has recently gained much attention from an industry as well as research perspective. Cova et al. present JSAND [17] to analyze and classify web content based on static and dynamic features. Their system provides a framework to emulate JavaScript code and determine characteristics that are typically found in malicious code. Such characteristics include code obfuscation, environment preparation, and exploitation techniques. In [48], Ratanaworabhan et al. present NOZZLE, a dynamic system that uses a global heap health metric to detect heap-spraying, a common technique used in modern browser exploits. In [18], the authors present a mostly static analysis engine called Zozzle. This system uses a naive bayes classifier to finding instances of known, malicious JavaScript [18].

These systems’ goals are orthogonal to those presented in this paper. Combining them with ROZZLE can improve detection results, and, in fact, we extended two of these systems (NOZZLE and Zozzle) to evaluate our system (see Section V). Similarly, JSAND could benefit from our system, as its dynamic features are currently limited to a single-profile execution. WebPatrol [15] aids the security analysis of obfuscated, malicious webpages. It allows one to collect and replay an infection scenario, which reduces the number of requests required by repeated analysis of a given URL. However, it can only cache results it has seen during its initial analysis and, thus, the system cannot handle content for a particular, fingerprinted browser instance potentially required in our analysis scenario.

Other systems focus on vulnerability analysis and validation of code in a benign context. In [5], the authors describe NoTamper, a tool that analyzes validation routines written in JavaScript. Using a black-box approach, NoTamper checks server-side code to find missing or
C. Environment Fingerprinting

Malicious code frequently uses fingerprinting to gather information on a target host. This information is then used to accommodate to differences in the execution environment, to launch exploits specific to the host, or deter execution inside an analysis system.

Fingerprints can be extracted from a variety of sources. For instance, attackers use information from the network layer [29, 54] to identify software components running on a remote target. This information greatly reduces the attack vectors and improves changes of a successful exploit.

Another data source is the underlying CPU architecture. In [13], the authors present a system for building binaries that identify the CPU using semantic differences of individual opcodes. This way, programs are able to execute different behavior depending on the execution environment. Balzarotti et al. present a system [2] to detect “split personalities” in malware. Its aim is to detect programs showing CPU-dependent behavior intended to evade analysis inside sandboxes such as ANUBIS [3] or CWsandbox [63].

In [19, 20], the author describe PANOPTICLICK, a system for identifying the uniqueness of a particular browser configuration. The author argues that the fingerprint of most configurations are unique and might be used to track individual users browsing the web. Mowery et al. [41] extend this idea and identify browser version, operating system, as well as the underlying CPU model without the use of APIs offered by the JavaScript engine. More precisely, they generate fingerprints from JavaScript performance benchmarks or by exploiting the popular NoScript plugin [32]. While these techniques are related to our system, the domain is very different.

D. Malware Detection

Throughout the evaluation of ROZZLE (see Section V) we discovered malicious pages on the Internet. A number of other researchers have proposed different solutions to discovering “bad neighborhoods” on the web. Typically, these studies [37, 40, 46, 47, 52, 70] rely on a combination of high- and low-interaction client honeypots to visit a large number of sites, detecting suspicious behavior in environment state after being compromised. Nazario and Song et al. recognized the divergence of needed browser plugins and proposed methods around ActiveX emulation to grow the attack surface of client honeypots [44, 55]. Hu et al. [24] study redirection botnets using changing attributes of DNS information over time. In [57] Stokes et al. propose a bottom-up fashion for finding websites serving malicious binaries or drive-by exploits by following in-links.
try {
  var a;
  var aa = new ActiveXObject(
    "ShockwaveFlash.ShockwaveFlash" + _ + "v" + _ + "e" + _ "FL" + _ "a" + _ "s" + _ "h.3" + _ "bo" + _ "ckw" + _ "aveF" + _ "las" + _ "h");
  } catch (a) {
  }
  finally {
    if (a != "[object Error]") {
      document.write("<iframe width=" + 111 + " height=" + 111 + " src="svfl9.htm"></iframe>");
    }
  }

Fig. 25: Pulling in new contexts depending on some flash-plugin version.

on exploit sites found through AV software to connected landing sites.

Often, authors of malicious websites make use of automated poisoning of search engine results to increase the number of potential victims directed to their site [30]. To avoid being detected and in turn being deleted from the search index, they make use of cloaking to conceal the true, malicious intent of the webpage. Previous work [64] has studied the prevalence of different types of cloaking using the similarity of links and terms in documents. In [31], the authors extend this idea by adding tags found in a website to improve detection of cloaked web pages.

Rozzle extends this direction of research. While we also aim to detect suspicious sites on the web, our evaluation shows that our system is able to expose more malicious activity than existing techniques using vulnerable, virtualized environments. At the same time, our multi-profile, in-browser interpretation of JavaScript requires fewer resources and reduces load on the crawled web sites.

Similar to the discovery of previously unseen malicious pages on the web is the idea of finding new behavior in malicious binaries. To this end, systems using static, dynamic, or a combination of both approaches have been proposed [26, 33, 36, 42, 43, 49]. Our system is similar in the sense that we also try to discover new malicious code. However, the application domain (malicious JavaScript) as well as the form of protection (client-side fingerprinting) are somewhat different.

IX. Conclusions

In the last several years, we have seen mass-scale exploitation of memory-based vulnerabilities migrate towards drive-by attacks delivered through the browser. With millions of infected URLs on the internet, JavaScript malware now constitutes a major threat to everyday computer use.

While both static and runtime methods for malware detection been both proposed in the research literature, both on the client side, for just-in-time in-browser detection, as well as offline, honeymonkey-style malware discovery, these approaches encounter the same fundamental limitation. Web-based malware tends to be environment-specific, targeting a particular browser, often with specific versions of installed plugins. This is because the exploits will often only work on specific plugins and crash otherwise. As a result, a fundamental limitation for detecting a piece of malware is that malware is only triggered occasionally, given the right environment. In fact, we observe that using current fingerprinting techniques, just about any piece of existing malware may be made virtually undetectable with the current generation of malware scanners.

This paper proposes a JavaScript multi-execution technique to explore multiple execution paths in parallel as a way to make environment-specific malware reveal itself. We experimentally demonstrate that, when used for static online detection, Rozzle finds an additional 5.6% of malicious URLs, indicating that currently malware does not yet use sophisticated cloaking techniques to hide itself from our detector. Rozzle increases the detection rate for offline runtime detection by almost seven times. Finally, Rozzle triples the effectiveness of online runtime detection with minimal overhead.

Moreover, Rozzle offers linear savings to hardware requirements, network bandwidth, and power consumption.

Acknowledgments

The authors would like to thank Paul Rebriy, Kelly You, and David Felstead for their encouragement, insight, and support throughout this project.

Appendix

Figures 20–25 show representative examples of real-life malware cloaking found by comparing the findings of a dynamic malware detector when run with and without Rozzle.

References

function SetCookie(name, value) {
    var Days = 30;
    var exp = new Date();
    exp.setTime(exp.getTime() + Days * 24 * 60 * 60 * 1000);
    document.cookie = name + '=' + escape(value) + '; expires=' + exp.toGMTString();
}

function getCookie(name) {
    var arr = document.cookie.match(new RegExp("(^| )" + name + "=(\[^;]*)(;|$)"));
    if (arr != null) returnunescape(arr[2]);
    return null;
}

var keyStr = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

function decode64(input) {
    var output = "";
    var chr1, chr2, chr3 = "";
    var i = 0;
    while (i < input.length);
        enc1 = keyStr.indexOf(input.charAt(i++));
        enc2 = keyStr.indexOf(input.charAt(i++));
        enc3 = keyStr.indexOf(input.charAt(i++));
        chr1 = ((enc1 & 3) << 6) | enc4;
        chr2 = ((enc2 & 15) << 4) | (enc3 >> 2);
        chr3 = (enc1 << 2) | (enc2 >> 4);
        output += String.fromCharCode(chr2);
        if (chr3 != 64) {
            output += String.fromCharCode(chr3);
            while (i < input.length);
                return output;
        }
    return ";";
}

var s = "PHNJcm1wD4NCg0KdmFyIHNjID0gW1s2NhcGUoIiV1...";
if (navigator.userAgent.indexOf("MSIE 6") != -1) {
    if (getCookie(‘qtr’) == null) {
        document.write(decode64(s));
        SetCookie(‘qtr’, ’1’);
    }
}

Fig. 20: Check for IE version and decode + eval only if we are running IE 6. There is an additional check using a cookie to run the exploit only once every 30 days.


