Rozzle: De-Cloaking Internet Malware

Clemens Kolbitsch
Lastline, Inc.

Benjamin Livshits and Benjamin Zorn
Microsoft Research

Christian Seifert
Microsoft Corporation

***Abstract*—JavaScript-based malware attacks have increased in recent years and currently represent a significant threat to the use of desktop computers, smartphones, and tablets. 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 fails 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. We observe that, using *fingerprinting* techniques that capture and exploit unique properties of browser configurations, almost all existing malware can be made virtually impossible for malware scanners to detect.**

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.**

***Index Terms*—malware; cloaking; JavaScript**

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 [31].

While static and runtime methods for malware detection have been proposed in the research literature [11, 12, 26], 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 [22, 23, 37], the traditional approach to improving path coverage involves *symbolic execution*, a powerful multi-path exploration technique that is often associated with non-trivial performance penalties [7, 8, 16, 27]. As such, off-the-shelf symbolic execution is not a feasible strategy. In a brute-force attempt to increase detection rates, offline detectors often deploy and utilize a variety of browser configurations side-by-side. While potentially effective, it is often unclear how many environment configurations are necessary to reveal all possible malware that might be lurking within a particular web site. Conversely, many sites will be explored using different configurations despite the fact that their behavior is *not* environment-specific. As a result, this approach requires significantly more hardware, power , and network bandwidth.

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. Rozzle implements a single-pass multi-execution approach that

```
try {
  E5Jrh = new ActiveXObject(" AcroPDF.PDF");
} catch ( e ) { }
if (! E5Jrh )
  try {
    E5Jrh = new ActiveXObject("PDF.PdfCtrl");
  } catch (e) { }
if ( E5Jrh ) {
  lv = E5Jrh.GetVersions().split(" ,")[4].
    split(" = ")[1].replace(/\./ g, " ");
  if ( lv < 900 && lv != 813)
    document.write('<embed src ="../validate.php?s=PTq..."
      width=100 height=100 type="application/pdf"></embed>')
}
try {
  var E5Jrh = 0;
  E5Jrh = (new ActiveXObject(
        " ShockwaveFlash.ShockwaveFlash.9 " ))
        .GetVariable(" $ " + " version " ).split (" ,")
} catch (e) { }
if ( E5Jrh && E5Jrh [2] < 124)
  document.write('<object classid ="clsid:d27cdb6e-ae..."
    width=100 height=100 align=middle><param name="movie"... ');
```

Fig. 1: Typical JavaScript exploit found in the wild that demonstrates environment matching.

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. In Section II, we experimentally demonstrate that the *fragility* metric correlates highly with maliciousness.
- **Low-overhead multi-execution.** We describe Rozzle, a system that *amplifies* other static and dynamic malware detectors. Rozzle implements lightweight multi-execution for JavaScript, a low-overhead specialized execution technique that explores multiple malware execution paths in order to make malware reveal itself to both static and runtime analysis.
- **Detection effectiveness.** Using 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.* 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 over three-fold and statically finding 5.6% more malicious URLs. Furthermore, in our experience, Rozzle does *not* introduce any new false positives.
- **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.
- **New attack directions.** 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 outlines approaches to creating more powerful attacks that use cloaking to hide from the current generation of malware detection tools. Section VII discusses the limitations of Rozzle and illustrates real instances of cloaking, discussed in more detail in a companion technical report [20]. Section VIII discusses related work, and, finally, Section IX concludes.

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

function get_version(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_ver(navigator.plugins[i].description,4);
  else if (window.navigator.javaEnabled && java_plugin == 0 &&
         plugin_name.indexOf("Java") != -1)
    java_plugin = get_ver(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 generated fingerprint
fetch_exploit(fingerprint);
```

Fig. 2: Sophisticated environment fingerprinting.

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* [29, 33], 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 of a scripting language, since copying of the shellcode is typically accomplished with a single `for` loop.

Previous work on runtime heap spraying detection [26] and static malware detection [12] suggests that there are millions of malicious sites containing heap spraying as well as other kinds of JavaScript-based malware, such as scareware. Previous reports point out the prevalence of JavaScript malware cloaking [3, 10, 36]. 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 contains at least some form of cloaking, environment matching, or fingerprinting.

A. JavaScript Malware: An Example of Real-Life Malware

To give the reader a better understanding of specific problems that our work addresses, we present an example of browser fingerprinting code found in the wild. A cleaned up version of this example, shown in Figure 2, employs precise fingerprinting to deliver only selected exploits that are most likely to successfully attack the client browser. Lines 1–33 compile the portion of the fingerprint that records the presence of the Adobe Acrobat, Quicktime, and Java plugins. Lines 35–36 record the presence of the Windows Media Player. Lines 45–46 construct the fingerprint string variable and lines 48–63 augment it with the browser language. Finally, line 66 issues a request to a malware hosting site to fetch the malware that corresponds to the computed fingerprint.

B. Current Practices: Matching, Cloaking, Fingerprinting

Based on our experience with specific malware samples such as the example above, we distinguish between three categories of techniques commonly used in today's malware: environment matching, fingerprinting, and cloaking.

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 [14, 24]. Malware writers also use fingerprinting, as illustrated in Figure 2, to deliver malware customized for a particular browser configuration or, perhaps, even in the case of targeted attacks, for a particular user.

Cloaking: Cloaking is a technique to show different content depending on conditions reflecting who is visiting the site. Offline malware scanning is used routinely to compile black lists of malicious URLs [25, 26]. In this scenario, cloaking can be successfully used by malware writers to avoid being detected when the malware-detecting crawler visits a particular site. Cloaking occurs with server-side and client-side variants. Server-side cloaking works by treating certain categories of HTTP headers or IP addresses, such as those coming from security vendors, differently. Client-side cloaking implements cloaking using JavaScript that detects crawlers by indentifying client-side characteristics that are unique to them. For example, a common crawler optimization is to avoid loading images to save bandwidth. We have observed malware that checks if images have been successfully loaded before executing its attack.

C. Code Fragility Experiment

Our experience with malware indicates that environment-dependent code is often malicious. In this section, we measure 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, browser version, installed plugins, the operating system, or even the CPU architecture. We conclude that, indeed, *code fragility* is an excellent measure of maliciousness.

Fragility detection tool: To evaluate our hypothesis, we constructed a simple static analysis tool for determining what conditionals (`if`s) in JavaScript code are environment-dependent. The tool works by statically *tainting* values [34] that are dependent on the `navigator` object and its fields as well as 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 [12], a static malicious JavaScript detector, and 194 files flagged by a simple static classifier-based detector trained only to detect code fragility. These 194 files are a strict subset of the 2,373 files detected by Zozzle. A summary of this data is presented in Figure 3.

The figure shows the number and fraction of files that have particular characteristics. The three main columns,

	All		Malicious		Static fragility	
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. 3: Measuring code fragility: Environment usage statistics across different categories of code.

"All", "Malicious" and "Static fragility", show the fractions of the total with respect to the different subsets. The "All" column represents all files, "Malicious" represents the subset flagged as malicious by ZOZZLE, and "Static fragility" represents the subset flagged by a special static detector, similar to ZOZZLE, that we trained explicitly just to detect JavaScript that has control dependences on its environment. 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 below:

- Only 7.7% of all JavaScript files reference environment-specific values. This number is an estimate of the fraction of files that 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 (XML-RPC ActiveX objects, as well as `document` and `window` objects account for the remaining 9.3% which we do not consider to leak sensitive information). The same 89.5% of malicious files will branch on these conditions.
- The static fragility detector triggered on fewer files than ZOZZLE, but the files detected were a complete subset of the malicious files detected by ZOZZLE, indicating that files with significant dependences on the environment are almost always malicious.

This analysis provides experimental support for our intuitive understanding: *exploits are environment-dependent.*

Avoidance technique	Works against		ROZZLE **improves**	
	Dyn.	**Static**	**Dyn.**	**Static**
Envir. testing	yes	no	yes	yes
Fingerprinting	yes	yes	yes	yes
Cloaking (client)	yes	yes	yes	yes
Cloaking (server)	yes	yes	no	no

Fig. 4: How avoidance strategies (Section II-B) work against dynamic and static malware detection techniques and how ROZZLE improves existing detection techniques.

III. OVERVIEW

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

A. Challenges and Existing Techniques

While static analysis is a powerful technique that allows one to explore all program paths, a particular issue that plagues static analysis in the context of malicious JavaScript is that we are unable to *observe all code.* Runtime evaluation has been advocated in this context [12, 17], 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. 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

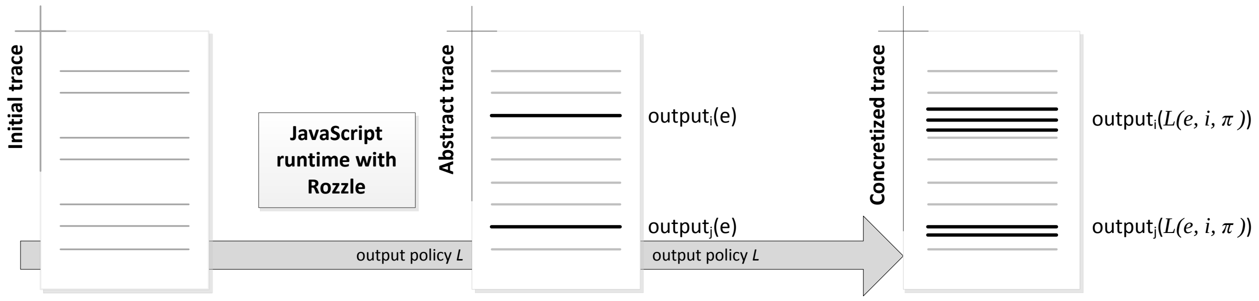


Fig. 5: Rozzle architecture as a series of trace rewriting steps.

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. The combinatorial growth of possible plugin version/browser/browser version combinations dictates the use of only the most popular environment configurations. In practice, this approach linearly expands the requirements on scanning hardware, network bandwidth, and power.

- **Overkill**. The ability to detect malicious pages that selectively target a particular type of browser requires re-scanning the same page. As shown 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.

Traditional symbolic execution: More recently, researchers have tried applying techniques of symbolic execution [5, 7, 8, 16] to the task of exposing malware [5, 23]. 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.

```
if (navigator.userAgent=="safari") {
  shellcode = unescape("1...");
} else {
  shellcode = unescape("A...");
}
```

Fig. 6: Simple example of the use of symbolic values.

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 4 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 6, Rozzle will execute both branches, one after another. Some readers might wonder if this creates a dependency on the order in which Rozzle will execute 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 [32] in optimizing compilers, except in the case of Rozzle, SSA construction happens at runtime.

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, transforming the initial trace on the left hand side of the figure, into the abstract trace in the middle. The abstract trace simultaneously captures the values of symbolic variables that are dependent on different program paths, but these abstract values must be made concrete when their values are used, for example, in network communications or with the DOM.

At such output statements, symbolic values are made concrete according to an output policy L, which is a function of the expression, e, the kind of output being done, i, and the environment, π.

C. Detailed Example of Multi-Execution

To build-up the reader's intuition, we now show a more involved example of how ROZZLE transforms execution. Figure 7 provides an illustrative example of multi-execution in action on a simplified code excerpt extracted from the fingerprinting routine in Figure 2. Figure 7(a) shows the original program. On line 9, we output the computed value `qt_plugin`. Figure 7(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 [4] to determine all potential outputs, to determine which inputs may lead to a particular output. Finally, Figure 7(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 7(b), with leaves contributing the potential values of the output, which are either the result of evaluating

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

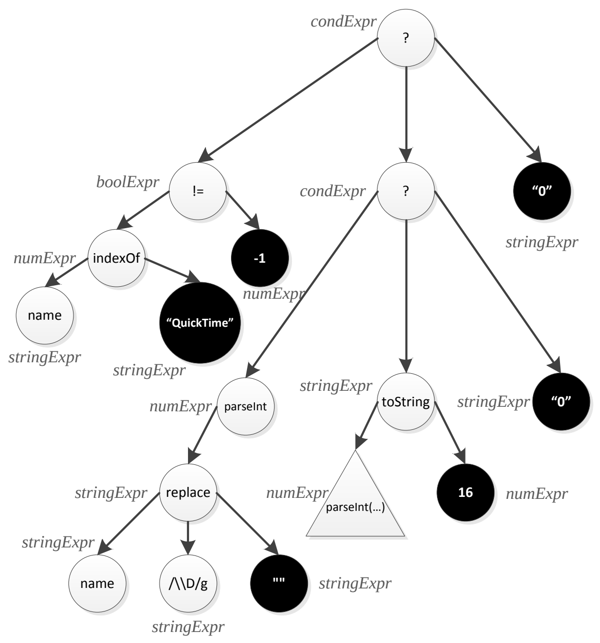
or `"0"`.

```
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 8. 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. 7: Example of multi-execution.

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 Internet Explorer 9. 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 built on top of IE 9.

A. Symbolic Values

Like traditional approaches to symbolic execution (e.g., [16, 18], our analysis treats specific values during execution as *symbolic*. Unlike traditional approaches to symbolic execution, however, that emphasize representing only sound and feasible executions, as the cost of performance overhead, our analysis allows unsound and infeasible executions with the benefit of increased performance. In ROZZLE, path 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 6 gives rise to a symbolic value that is created at runtime *after* the `if`/`else` construct:

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

Note the merge of the weak updates in the conditional due to 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-

Expressions		
symExpr	::=	*numExpr* | *stringExpr* | *boolExpr*
		| *memberCall* | *funcCall* | $\star$
		| *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. 8: BNF for symbolic expressions used in Rozzle. The start symbol is *symExpr*.

types) or `ActiveXObjects`. 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 the 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 runtime tainting. 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.

C. Symbolic Values: Manipulation and Representation

Figure 8 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 7(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 [6], etc. This way, symbolic values will share some of the subtrees, as illustrated with the triangle in Figure 7. In Rozzle, somewhat akin to [16], 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 containing symbolic path elements, requesting new content from the web, etc.) Figure 7(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 7.

Value concretization is required when the program interacts with various IO subsystems within the browser such as the DOM, sending data on the network or opening an new URL. The exact concretization approach is an implementation choice, governed by a *concretization policy* L, as shown in Figure 5. Accurate value concretization is not necessary to achieve significant experimental success with Rozzle. Specifically, many malicious exploits of the environment-matching variety are exposed to either a static or dynamic malware detector if the code that introduces them is executed *in any event*. However, sophisticated attacks using enviornment fingerprinting as shown in Figure 2 do require systematic concretization.

We represent profiles as collections of independent parts (e.g., user agents, ActiveX objects, etc.) and at each output apply the set of parts to the symbolic expression, selecting parts (e.g, "IE6") using brute-force search for values that are consistent with the constraints in the path expression. We take the first consistent output that results

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

Fig. 9: Statements in our program representation.

using this method and ignore the rest. We leave the generation of multiple concrete values and their prioritization for future work.

E. Details of Multi-Execution

Figure 10 shows pseudo code for our multi-execution engine. The inputs of the algorithms are program P, which is a collection of statements $\mathit{stmt}_1, \ldots, \mathit{stmt}_n$, 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 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 10 consists of an interpreter loop that handles the cases of an if conditional, a loop, etc. in turn.

Branching on symbolic values: When code branches on a symbolic value, we need to make a decision which branch to take. Because our goal is to detect possible malicious behavior down any path, despite potential unsoundness, Rozzle executes *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 11, 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 assign to either a variable or a heap object of the form $\mathit{object.field}$ 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. As mentioned before, in Rozzle, we execute branches that are dependent on symbolic variables sequentially, one after another. To support weak updates

```
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:         $\Gamma_t = []$
 6:         $\Gamma_t$.prototype $= \Gamma$
 7:         $MultiExecute(T_t, \pi, L, mod_t)$
 8:         $\Gamma_f = []$
 9:         $\Gamma_f$.prototype $= \Gamma$
10:         $MultiExecute(T_f, \pi, L, mod_f)$
11:         $\Gamma = Range(\Gamma_t) \cap Range(\Gamma_f)$
12:         for all $v$ in $\Gamma$ do
13:           $\Gamma[v_t] = \Gamma_t[v]$
14:           $\Gamma[v_f] = \Gamma_f[v]$
15:           $\Gamma[v] = \phi(e, v_t, v_f)$
16:         end for
17:         for all $v$ in $(Range(mod_t) \setminus \Gamma)$ do
18:           $\Gamma[v] = \phi(e, v, v_t)$
19:         end for
20:         for all $v$ in $(Range(mod_f) \setminus \Gamma)$ do
21:           $v = \phi(\neg e, v, v_f)$
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:         $\Gamma' = []$
35:         $\Gamma'$.prototype $= \Gamma$
36:         $MultiExecute(T, \pi, L, \Gamma')$
37:         for all $v$ in $Range(\Gamma')$ do
38:           $\Gamma[v] = \phi(e, \Gamma'[v], \Gamma[v])$
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:       $\Gamma[v] = e$
53:       end case
54:     case $v_1 = v_2$
55:       $\Gamma[v_1] = \Gamma[v_2]$
56:       end case
57:     case $output(e)$
58:       $L(e, i, \pi, \Gamma)()$
59:       end case
60:   end switch
61: end for
```

Fig. 10: 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.

on such code paths, we proceed as follows: Whenever a branch is encountered and ROZZLE finds that the condition is symbolic, the condition is pushed onto a stack used to keep track of path predicates. When executing the `else`-statement of a symbolic branch, the condition on the top of stack is inverted and used as condition for the new branch. The `else` block is handled by combining the element on stack with the new condition. When leaving the symbolic branch (i.e., after executing the last branch conditioned under the symbolic predicate), the symbolic condition is popped from the stack. Within a symbolic branch, weak updates are used for both variable assignments and heap object stores. For this, the current path condition (i.e., the conjunction of all elements of the path predicate stack) is used to build the tree of symbolic memory as described above. The pseudo code for handling conditionals is shown in lines 3–30 of Figure 10.

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

Fig. 11: Symbolic execution: a simple `if`.

Looping on symbolic values: Handling symbolic loops 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 31–60 of Figure 10. 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. If 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 on line 39 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 more methods available to call than in other settings, leading to fewer errors introduced by ROZZLE. Another reason for choosing Chakra is its performance [21]. When ROZZLE needs to resolve symbolic variables into concrete values, we use the JavaScript engine: symbolic memory is represented as a new JavaScript type and operated on at runtime by the engine. Symbolic memory can be directly translated into code that, given a set of environments, produces possible concrete values, as illustrated in Figure 7. Below, we describe the modifications implemented to support multi-execution in Chakra:

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

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

Fig. 12: Symbolic execution: `try`/`catch`.

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 values may be passed into both the JavaScript language and the DOM API functions exposed by the JavaScript engine. Example of this include `concat` and `indexOf` functions on strings. To handle such interactions, we need to modify natively implemented functions 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 in C, we only need to insert a single macro into the prologue of each function.

Virtual branching and `try`/`catch` blocks: Exceptions are common mechanism used by attackers to test the capabilities of the environment in which their code executes. As a result, we need to handle this specific idiom in our runtime. Consider the following commonly found example shown in Figure 12. 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 invert the active condition (just like in the `else` 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.

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 (see the discussion of looping above) 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 [15].

V. Evaluation

In this section, we evaluate the cost and benefit of Rozzle by comparing an unmodified browser (base) against a browser with Rozzle.

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. We placed the files on a local disk and visited each file as a local URL using the high-interaction client honeypot twice, once using its default configuration using an Internet Explorer 9 profile (*base* run) and a second time using the Rozzle-extended version. As the client honeypot renders and executes the page content, it scans any JavaScript contexts found using Zozzle and also uses Nozzle to detect any suspicious behavior during the execution of the scripts.

Results: Figure 13 shows the detection rates using the 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

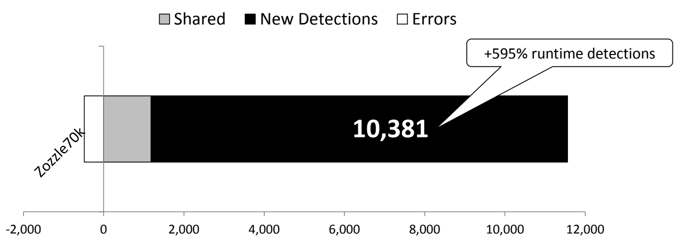
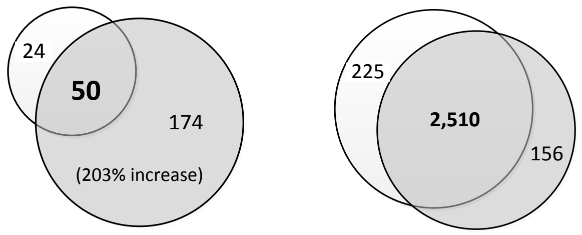


Fig. 13: Offline malicious-only detection: improvements.



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

Fig. 14: Online general URL detection: improvement rates.

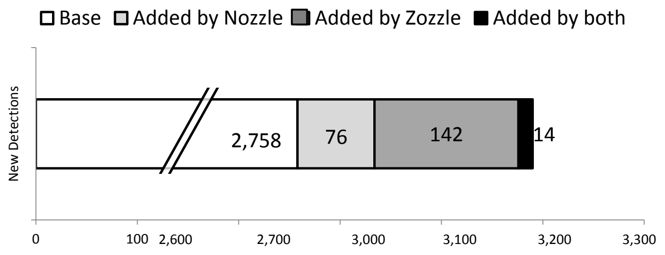


Fig. 15: Online general URL detection: new detections with Rozzle.

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% more detections). In 1178 cases (71%) the URL was flagged in both runs, leaving 484 URLs (29%) 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, as discussed Section VII-A. In contrast to the 484 introduced errors, we see that Rozzle is able to expose new malicious dynamic activity in 10,381 cases. We conclude that Rozzle is successful in increasing path coverage using multi-path execution and that environment-sensitive malware is a real problem that Rozzle is able to expose.

B. Improved Online Detection Rates with Rozzle

In this experiment, we collect and visit 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 the detectors. Early on in the evaluation, we found that this is not an easy task as malware hosting servers are quite unreliable. As others have noted [38], we encountered many malicious URLs where attacks were served only once to a given IP address/ subnet or only once within a given time period (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 Rozzle run, 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 that are caused by a clear difference in the content served by the server.

Setup: 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. 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 14(a). The 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 225 errors and 156 new Zozzle detections in the second run.

To better understand the differences in detections caused by Rozzle, we manually verified a subset of URLs constituting 1,540/1,557 Zozzle and 31/120 Nozzle detections in the base and second (Rozzle) runs, respectively. Our goal was to understand 1) how often Rozzle's failure to detect was cause by Rozzle, as opposed to other factors (false negatives), and 2) whether the additional Rozzle detections were actually malicious sites and not false positives.

In investigating false negatives, 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. 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, as 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 the remaining 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.

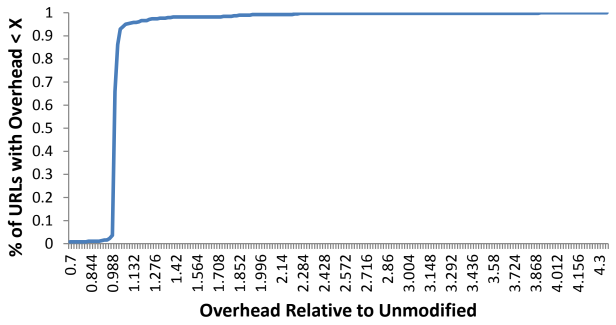
In investigating false positives, we scrutinized the additional malicious URLs detected using Rozzle, and determined that they were actually malicious. We also cross-checked to determine if there was overlap between the Rozzle-enabled Nozzle detections and detections found by the base configuration, to see if Rozzle had uncovered any previously undetected malware. We also did the converse check, looking for new Rozzle-enabled Zozzle detections the had previously not been detected by the base configuration.

We found that in 90 cases, Rozzle triggered a Nozzle dynamic malware detection whereas in the base run neither the static nor the dynamic detector detected the malicious script. Likewise, in 156 cases the Zozzle static detector flagged the URL as malicious although it had not been detected by either system without Rozzle. Of these new detections, 14 were detected by both Nozzle and Zozzle when Rozzle was used. Figure 15 summarizes these results graphically.

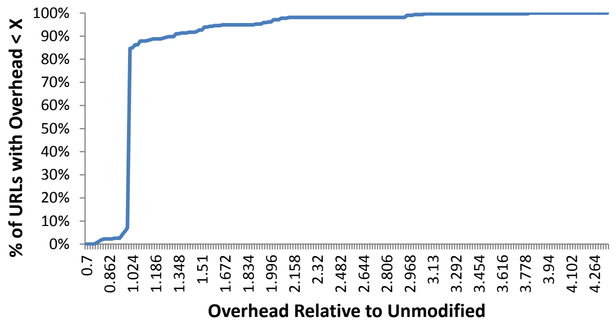
C. *Performance Overhead of* Rozzle

Setup: To measure the the CPU and memory overhead of Rozzle, we visit 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 an offline malware crawler would be expected to scan. To gather performance data, we added callbacks in the JavaScript framework before and after a context is executed. While we only measure the time within the JavaScript runtime, other factors contribute to the overall performance, and the relative overall impact on the end-to-end throughput is significantly less than shown here.

To measure Rozzle's memory impact, we added callbacks to the JavaScript allocator and garbage collector to be notified about allocation/de-allocation events. In measuring memory and CPU overhead, we analyzed each URL three times sequentially and we report the minimum of the three runs (the distribution of averages was similar to the distribution of minimums). Unfortunately, simply repeating analysis runs does not always provide reproducible numbers. We noticed that script execution times varied widely, often due to different contents 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



(a) Rozzle Memory overhead.



(b) Rozzle CPU overhead.

Fig. 16: Distributions of relative memory and CPU JavaScript engine overhead. Larger numbers imply more memory and slower execution time.

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 many fewer function invocations or called functions, we discarded the URL from analysis, assuming the site contained nondeterministic script inclusions or Rozzle caused execution to abort prematurely (and would thus skew results).

Results: Figures 16(a) and Figure 16(b) shows Rozzle's impact on memory and CPU, respectively. The figures show the cumulative distributions of observed overheads on the Y-axis, with the measured relative overhead plotted on the X-axis. Thus, for example, we see that over 90% of the URLs we measured had no memory overhead relative to the unmodified runtime. X-axis numbers greater than 1 indicate slower execution and more memory. The figures both show that for the vast majority of web sites, Rozzle has no performance or memory impact, which matches our earlier measurements indicating that few web sites 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 memory overhead is 0.6%

```
if (navigator.userAgent.toLowerCase().indexOf(
  "\x6D"+"\x73\x69\x65"+"\x20\x36") > 0)
    document.write("<iframe src=x6.htm></iframe>");
if (navigator.userAgent.toLowerCase().indexOf(
  "\x6D"+"\x73"+"\x69"+"\x65"+"\x20"+"\x37") > 0)
    document.write("<iframe src=x7.htm></iframe>");
try {
  var a;
  var aa = new ActiveXObject(
    "Sh"+"ockw"+"av"+"e"+"Fl"+[...]);
} catch(a) { } finally {
  if (a!="[object Error]")
    document.write("<iframe src=svfl9.htm></iframe>");
}
try {
  var c;
  var f = new ActiveXObject(
    "O"+"\x57\x43"+"\x31\x30\x2E\x53"+[...]);
} catch(c) { } finally {
  if (c!="[object Error]") {
    aacc = "<iframe src=of.htm></iframe>";
    setTimeout("document.write(aacc)", 3500);
  }
}
```

Fig. 17: Real-life malware roulette.

and the 80th percentile is 1.4%. The average overhead is slightly higher: the average CPU overhead is 10% and the average memory overhead is 3%. Since the JavaScript runtime is only a fraction of the total CPU and memory overhead in a modern browser, we consider these overheads to be acceptable for offline malware scanning and also even potentially usable for in-browser scanning as well.

VI. Attack scenario

In the course of our research, we have observed that the majority of exploits detected in the wild are naïve about detection avoidance. We have, however, seen examples of real malware that are difficult to detect without the techniques presented here. One such example is shown in Figure 17. A key observation here is that malware is injected *after* fingerprinting the browser.

In this section and in our companion technical report [20], we outline an attack strategy designed to avoid detection. The crux of this approach revolves around only revealing a small piece of JavaScript at a time, in order to avoid static detection and using server-side page redirects to avoid runtime detection.

The key idea is to re-write the original malware into a series of redirects to hide from traditional client-side honeypots: In each state, a small script extracts the necessary information (e.g., the browser version) and submits it to the remote server. In turn, the attacker redirects the browser depending on the submitted data: If the extracted information does not fulfill any path constraints leading to a possible exploit state, the browser is redirected to a benign site to avoid raising suspicion. Otherwise, the attacker redirects to a server under her control to proceed in the roulette. Eventually, a server delivers the actual exploit to the requesting browser. 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, the crawler is able to detect environment-sensitive parts of the URL and, using individual profiles, enumerate different sub-pages to visit. Thus, 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 can thwart client-side cloaking attempts but it is not effective 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.

A common technique to avoid crawler-based malware detection is to trigger the malware execution only when a user interaction occurs, or when a timer fires. This approach can make Rozzle less effective and represents a general problem for dynamic crawlers. A simple, brute-force approach to mitigate this form of hiding is to set all timer intervals to zero in the crawler and execute all handlers preemptively.

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

Identifying that Rozzle is enabled: It is difficult to hide the fact that Rozzle is used within the browser, because of functional differences in execution as well as timing differences, etc. As a result, client code that detects the presence of ROZZLE can avoid delivering the payload in that case. Another 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, time, or other resources.

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 17 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, 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`. Our companion technical report [20] provides a number of other such examples found in practice.

VIII. RELATED WORK

A. *Symbolic Execution*

Symbolic execution was introduced by King [18] and is used in research areas such as program testing and bug detection. Some notable efforts include [5, 8, 16, 23]. Path exploration approaches include enumerating program traces [16] or cloning the program environment (by `fork`-ing a new process [8]). In contrast, ROZZLE maintains the symbolic state of the program entirely in the runtime engine's memory.

Most importantly, potentially costly and unpredictable path feasibility testing is no longer employed by ROZZLE because is willing to consider potentially infeasible paths. We branch on environment-sensitive values instead of program inputs. Additionally, consider that JavaScript code frequently communicates with third-party servers that will detect and prevent repeated requests from the same machine. Symbolic path exploration in ROZZLE is performed in a single run; as a result, ROZZLE will minimize this communication instead of going through it repeatedly, as part of path exploration. These design choices, in combination generally make ROZZLE very lightweight.

Austin *et al.* introduce *faceted values*, a form of multi-execution, to simultaneously simulate multiple security levels during JavaScript code execution [1]. While the technique has elements in common with ROZZLE, the application of multi-execution is different (they are interested in reasoning about information flow security) and the values being propagated are different. Wilhelm *et al.* propose forcing branches in the program for rootkit identification [35].

B. *JavaScript Analysis*

With the growing popularity of browser-based JavaScript applications, JavaScript analysis has recently gained much attention.Cova *et al.* describe JSAND [11] for analyzing and classifying 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. Ratanaworabhan *et al.* describe NOZZLE, a dynamic system that uses a global heap health metric to detect heap-spraying, a common technique used in modern browser exploits [26]. In [12], the authors present a mostly static analysis engine called ZOZZLE. This system uses a naive bayes classifier to finding instances of known, malicious JavaScript.

These systems' goals are orthogonal to those presented in this paper. Combining them with ROZZLE can improve detection results and 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.

A system closely related to ROZZLE is KUDZU [27]. 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 [28], 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.

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 [19, 30] to identify software components running on a remote target. This information greatly reduces the attack vectors and improves chances of a successful exploit. Another data source is the underlying CPU architecture. In [9], 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], whose aim is to detect programs showing CPU-dependent behavior intended to evade analysis inside malware sandboxes.

In [13, 14], 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.* [24] extend this idea and identify browser version, OS, as well as the underlying CPU model using timing information.

IX. CONCLUSIONS

JavaScript-based malware attacks have increased in recent years and currently represent a significant threat to the use of desktop computers, smartphones, and tablets. In this paper, we show that Web-based malware tends to be environment-specific, targeting a particular browser, often with specific versions of installed plugins. As a result, a fundamental limitation for *detecting* a piece of malware is that malware is only triggered occasionally, given the right environment. We observe that using current *fingerprinting* techniques, 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. ROZZLE offers significant practical improvements in hardware requirements, network bandwidth, and power consumption. Furthermore, in our experience, ROZZLE does *not* introduce any new false positives.

ACKNOWLEDGMENTS

We appreciate the cooperation and efforts of David Felstead and Paul Rebriy of the Bing malware team in the course of this research. We also appreciate the helpful feedback from the anonymous reviewers.

REFERENCES

[1] T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *POPL'12*, Jan. 2012.

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

[4] 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.

[5] 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.

[6] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Comp. Surveys*, 24(3), Sept. 1992.

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

[8] 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*, 2006.

[9] 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.

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

[11] 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*, April 2010.

[12] 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.

[13] P. Eckersley. How unique is your web browser? In *Privacy Enhancing Technologies*, 2010.

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

[15] M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI '02*.

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

[17] 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.

[18] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19, July 1976.

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

[20] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: Decloaking Internet malware. Technical report, Microsoft Research, Sept. 2011.

[21] Microsoft Corp. 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`, 2010.

[22] 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*, 2010.

[23] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proc. of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2007.

[24] 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.

[25] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The ghost in the browser: Analysis of web-based malware. 2007.

[26] 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.

[27] 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.

[28] 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*.

[29] 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.

[30] M. Smart, G. R. Malan, and F. Jahanian. Defeating TCP/IP stack fingerprinting. In *Proc. of the Usenix Security Symposium*, 2000.

[31] Sophos Labs. Security threat report 2011, 2011.

[32] P. Tu and D. Padua. Gated SSA-based demand-driven symbolic analysis for parallelizing compilers. In *Proceedings of the International Conference on Supercomputing*, 1995.

[33] R. van den Heetkamp. Heap spraying. `http://www.0x000000.com/index.php?i=412&bin=110011100`, Aug. 2007.

[34] L. Wall. Perl security. `http://search.cpan.org/dist/perl/pod/perlsec.pod`.

[35] J. Wilhelm and T. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Recent Advances in Intrusion Detection*. 2007.

[36] B. Wu and B. D. Davison. Detecting semantic cloaking on the web. In *Proceedings of the International Conference on World Wide Web*, 2006.

[37] C. Xuan, J. Copeland, and R. Beya. Toward revealing kernel malware behavior in virtual execution environments. In *RAID*, 2009.

[38] K. Zeeuwen, M. Ripeanu, and K. Beznosov. Improving malicious URL re-evaluation scheduling through an empirical study of malware download centers. 2011.