A Simple Client-Side Defense Against Environment-Dependent Web-Based Malware

Gen Lu, Karan Chadha, Saumya Debray
Department of Computer Science
The University of Arizona
Tucson, AZ 85721,USA
{genlu, kchadha, debray}@cs.arizona.edu

Abstract

Web-based malware tend to be environment-dependent, which poses a significant challenge on defending web-based attacks, because the malicious code—which may be exposed and activated only under specific environmental conditions such as the version of the browser—may not be triggered during analysis. This paper proposes a simple approach for defending environment-dependent malware. Instead of increasing analysis coverage in detector, the goal of this technique is to ensure that the client will take the same execution path as the one examined by the detector. This technique is designed to work alongside a detector, it can handle cases existing multi-path exploration techniques are incapable of, and provides an efficient way to identify discrepancies in a JavaScript program's execution behavior in a user's environment compared to its behavior in a sandboxed detector, thereby detecting false negatives that may have been caused by environment dependencies. Experiment shows that this technique can effectively detect environment-dependent behavior discrepancy of various forms, including those seen in real malware.

1 Introduction

In recent years, web-based attacks against browsers—also known as *drive-by downloads*, and typically executed using JavaScript—have become an important mechanism for malware delivery. Detecting web-based malware requires the ability to identify malicious JavaScript code, which is a challenging task. Existing techniques for the detection of such malware, discussed in more detail in Section 7, suffer from various shortcomings that make it possible for malware to evade them [14]; in particular, malware triggered by environmental conditions can be difficult to detect.

This paper proposes a different technique, called *environmental predicates*, for defending environment-dependent malware. In contrast to the attempts on increasing code coverage in detectors, this technique extends and augments existing sandbox based detector with a lightweight outside-the-sandbox mechanism that can allow a browser to check, safely and efficiently, whether the JavaScript code would follow the same execution path as the one examined by the detector, therefore protecting users from possible detection false negatives caused by environmental discrepancy between real browser and the sandbox (such difference may exist due to the implementation of the sandbox, even when both browser and detector run on the same machine). This technique can handle various mechanisms, involving indirect control transfers and environment-dependent loops, that can be problematic for existing multi-path exploration techniques. Finally, the cost of analyzing a web page can be amortized across multiple views of the same page. Experiment results show the proposed technique can effectively detect environment-dependent behavior discrepancy of various forms, including those seen in real malware.

2 Background

Environment-dependent malware use characteristics of their execution environment to determine whether, or how, the malicious code is deployed. Existing web-based malware often check whether the hosting system is vulnerable before exposing any malicious content, thereby achieving greater stealth. Kolbitsch *et al.* list a collection of environmental-specific values widely exploited by existing malware [11]. The simplest and most direct approach for this is to test the environmental inputs against expected value(s). One drawback with straightforward code like this is that it is not difficult for automated tools to identify and analyze, e.g., using multi-path explorations techniques to force execution

into the branch of the conditional that causes the malicious code to be exposed. However, traditional symbolic execution based approaches often incur significant overhead, which renders them infeasible for the purpose of malware detection, especially for online detectors [18, 15, 3, 6].

More recently, Kolbitsch *et al.* propose Rozzle, a lightweight JavaScript multi-path execution framework, which explores multiple execution paths within one execution instead of exhaustively executing all possible paths [11]. While experiment result shows Rozzle can effectively improve the detection rate of existing detectors with little runtime overhead, the improved performance is at the cost of possible runtime errors introduced by execution of logically infeasible paths. Moreover, it also has the same limitations encountered by many other symbolic-execution based approaches. Handling symbolic loops probably presents the most complex case for symbolic execution algorithms. For example, Rozzle treats all environment-dependent data as symbolic, and the execution of symbolic loop is terminated after one symbolic iteration. Malicious code can take advantage of this using environment-dependent loops and thereby escape detection [14].

3 Environmental Predicates

Environmental predicates extend and augment existing sandbox based detection [5, 17] with a lightweight outside-the-sandbox mechanism that allows a browser to efficiently check whether the JavaScript code would follow the same execution path in the browser as in the sandbox and found to be benign (a web page found to be malicious is quarantined immediately): if the check is satisfied, the web page can be viewed safely. This check is carried out each time the web page is viewed; this detects any changes to environmental conditions that would cause a different execution path to be taken. The web page can then be submitted for reanalysis or possibly quarantined as suspicious.

Given a JavaScript program P that takes a (non-malicious) execution path π during sandboxed analysis, an environmental predicate f with respect to a set S of environmental inputs for the pair (P, π) consists of the following: (1) the code from P that contributes to all of the control-flow branches in P that depends, directly or indirectly, on sources in S during its sandboxed execution; (2) a vector, called the *decision vector*, that records the decisions made by environment-dependent branches when traversing the execution path π; and (3) code snippets, called *checkpoints*, that are inserted at environment-dependent branched in f to ensure that the control flow behavior of the program does not deviate from that specified by the decision vector. The predicate f evaluates to true if and only if the sequence of branch decisions made during its execution, as verified by the checkpoints, exactly matches those recorded in the decision vector. Figure 1 shows an example of an environmental predicate.

The construction of environmental predicates is based on a JavaScript analysis framework developed by Lu *et al.* [13] and consists of the following steps:

1. Use an instrumented interpreter within a sandbox to obtain an execution trace for the JavaScript code under analysis, such that any access to the environment-dependent data will be labeled in the trace.

2. [*Control Flow Analysis*] Construct a control flow graph from the collected trace to determine the structure of the executed code. Since this control flow graph is constructed from an execution trace, it may be incomplete because some code may not have been executed. We refer to this as a *dynamic control flow graph*.

3. [*Code Trimming*] Identify instructions relevant to environment-dependent branches, i.e., changes of execution path that are affected by environment-dependent data.

4. [*Decompilation*] Decompile the dynamic control flow graph to an abstract syntax tree (AST), label all the nodes constructed from resulting set of relevant instructions, and use semantics-preserving transformations to eliminate goto statements.

5. [*Environmental Predicate Generation*] Finally, generate environmental predicate based on labeled (relevant) syntax tree nodes.

In this paper, we focus on code trimming and environmental predicate generation. Reader interested in other components of the system is referred to [13].

A naive way to construct an environmental predicate would be to include all the code recovered from execution trace, regardless of whether it is relevant to any environment-dependent branch. Also, every branch decision made during an execution has to be logged in the decision vector, and checkpoints are required at all branches. However, the environmental predicate constructed this way can be large and inefficient. It is shown that, unlike malicious code, most benign programs do not branch on environment-dependent data [11]. Having such irrelevant code would significantly increase the execution time and size of environmental predicates.

To address this problem, we use a code trimming algorithm to eliminate irrelevant code and reduce the size of environmental predicates. The basic idea is to use

```
try{
  x = new ActiveXObject(
       "ShockwaveFlash.ShockwaveFlash");


}catch(e){




}
```

(a) Original program

```
function environmentalPredicate () {
  vecIndex=0;
  DecisionVector=[1];
  try{
    try{
      x = new ActiveXObject(
            "ShockwaveFlash.ShockwaveFlash"
      if(decisionVector[vecIndex++]!=0)
        throw "wrong path";
    }catch(e){
      if(e=="wrong path")
        throw "wrong path";
      if(decisionVector[vecIndex++]!=1)
        throw "wrong path";
    }
  }catch(e){return false;}
  return true;
}
```

(b) Environmental predicate

Figure 1: An example of environmental predicate, generated from the execution of the original program in which new statement threw an exception.

a program analysis technique known as dynamic slicing [1] to identify code that is relevant to environment-dependent branches in the program's execution trace. In addition, all the environment-dependent branch decisions have to be logged in execution order and stored in a vector. Our algorithm uses a combination of forward and backward slicing, both adapted for handling the dynamic bytecode trace generated by our sandbox. They take the dynamic trace T, an instruction instance $instr$ in T and the dynamic control flow graph G as input, and return a forward/backward slice S of T with respect to $instr$.

Input: A dynamic trace T; a dynamic control flow graph G;
Output: A slice S with respect to all environment-dependent branches in T;

$S := \emptyset$;
$U := \emptyset$;
for i := 1 **to** *length of T* **do**
  $instr$:= i-th instruction instance in T;
  if $instr$ *is the source of environment-dependent data* **then**
    $U := U \cup$ Forward-Slicing(T, $instr$, G);
  end
end
for i := *length of T* **to** 1 **do**
  $instr$:= i-th instruction instance in T;
  if $instr$ *is a decision-making branch instruction* $\wedge$ $instr \in U \wedge instr \notin S$ **then**
    $S := S \cup$ Backward-Slicing(T, $instr$, G);
  end
end

Algorithm 1: Environment-dependent decisions identification algorithm.

The pseudocode of our algorithm for identifying environment-dependent branches is shown in Algorithm 1, which consists of two basic steps, both of which rely on dynamic slicing:

1. *identify all instructions dependent on environment-dependent data (line 3-8):* The trace collecting sandbox is instrumented such that any access to the environment-dependent data will be labeled in the execution trace. As a first step, for each labeled instruction instance in the trace, forward-slicing algorithm is applied on it to identify instructions dependent on environment, which include branches if there is any executed. After this step set the variable U contains all the instructions in trace T that are environment-dependent.

2. *identify all instructions relevant to environment-dependent branches (line 9-14):* The algorithm then traverses the trace backwards, applying the backward-slicing algorithm on each branch in set U. The backward slice in resulting set S contains all instruction semantically relevant to the environment-dependent branches during execution, i.e., instructions that affect the decisions made by environment-dependent branches. This slice can be executed as a stand-alone program after decompilation.

To construct an environmental predicate for a given execution, we insert checkpoint nodes into transformed AST from decompilation step, at the locations following each environment-dependent branch. New identifiers introduced are chosen in a way such that they don't interfere with original program state. Table 1 presents the rules used by our checkpoint insertion algorithm.

As we can see, inserting checkpoints for if-else, switch, loop and function invocation is very straightforward, as shown in the corresponding code snippet in the table.

The idea behind inserting checkpoint for try-catch statement is not as apparent, however. First, we need to insert a checkpoint after each statement in the try

body, since we assume every one of them might throw an exception during execution (this is a rather conservative assumption with room for improvement, which we leave for future work). As a result, every statement in try-clause is treated as a environment-dependent branch during code trimming step. Then we insert two checkpoints at the beginning of the catch-clause. The first one is to examine whether the exception is thrown by one of the checkpoints in try , if so, it has to be relayed to outermost catch. Otherwise, the exception is thrown by one of the statements in try from the original program, then the second checkpoint in catch will examine whether the decision of jumping into catch match the decision made in sandboxed execution (as logged in decision vector). Figure 1 shows an example of actual environmental predicates for try-catch branch mechanism.

It is also possible that a statement not enclosed in try-catch throw an unhandled exception during sandboxed execution and cause the program to halt. Environmental predicate needs to capture the branch of control flow in this scenario, it might cause a false negative otherwise. For example, a call to ActiveXObject function in our Firefox based sandbox would throw an (unhandled) exception and abort the execution, but the same statement would succeed in Internet Explorer with corresponding plugin installed and proceed to exploit. To handle this case, we simply treat the statement that threw the unhandled exception as an environment-dependent branch and insert throw "wrong path" right after it in corresponding environmental predicate.

In addition, identifiers used in extra code introduced by the construction of environmental predicate are chosen in a way such that they don't interfere with original program state.

4 System Architecture

While our main contribution in this paper is the environmental predicate, it is designed as an enhancement technology instead of a stand-alone defense, therefore a malware detector is still required for the prototype system to work properly. Our prototype system is designed as a client-side detector, which has three major components: a controller, a runtime malware detector and an environmental predicate generator, as shown in Figure 2 (the controller is omitted from Figure 2 for a clearer presentation of workflow). The environmental data sources selected are shown in the column "Environment-Dependent Data Referenced" of Table 2. The controller serves as the coordinator and interacting with other components in the system, and is also responsible for deciding whether a web page is safe to be executed, based on the information provided by detection result and environmental predicate. A results cache is employed to reduce overhead on previously processed pages, each cache record represents analysis result of a single web page, including detection result and environmental predicate.

At a high level, the process of checking an incoming web-page evolves in following steps, as shown in Figure 2:

1. Controller intercepts requested page before it is loaded by browser. If the record of this page (its hash value) is found in cache, go to step 4, else proceed to next step.
2. Page is forwarded to the secure sandbox for execution.
3. Detector and environmental predicate generator analyse the sandboxed execution and return detection result and an environmental predicate to controller.
4. Based on the analysis result, and the value returned by executing environmental predicate in the browser, the controller decides whether it is safe to load the web page in browser and act accordingly.

In addition to the design described above, in which all the analysis is done at the client side per request, the environmental predicate based technique can be utilized in a server-clients type of system as well. Consider the scenario in which a search engine runs analysis on every web page crawled. As discussed previously, there are malware cloaking techniques, when applied, capable of evading existing detectors, even with the help from state of the art multi-path exploration techniques. Therefore, all web pages detected as benign but with environment-dependent branches can potentially be malicious, and more labor intensive analysis is required. By using environmental predicate, the dubious page can still be returned for user queries even before the final analysis result is available. In this case, a environmental predicate is created by the server based on the benign execution path, which is passed to user who clicked on the dubious link in the search result first, and the user is redirected to the actual web page only if the environmental predicate shows the client makes identical environment-dependent decisions as in the benign path.

5 Experimental Evaluation

To evaluate the efficacy of our approach, we implemented a client-side prototype system as described in Section 4, and test it on real malware. Since the primary focus of this work is on detecting environment-dependent code, we simply assume the existence of a

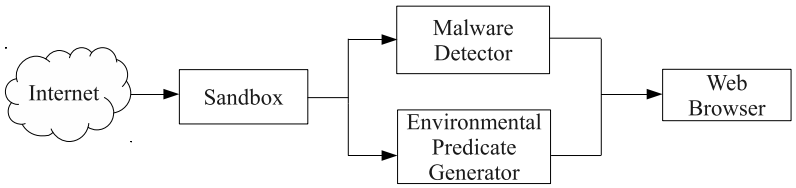

Figure 2: System Architecture

black-box detector that identifies malicious behavior and use an instrumented Mozilla Firefox web browser as the sandbox rather than a secure emulator for dynamic analysis. As a result, we selected real environment-dependent malware which are collected from the Internet (e.g. spam emails and security blogs) as testcases, such that they will not conduct attack when executed in our sandbox but might behave maliciously in the actual browser due to the environmental triggers employed. In addition, we arbitrarily chose several malware samples and manually inserted environment-dependent tests using branch mechanisms that have not been used in existing malware, such as function invocation using bracket notation and implicit conditionals (an emulation based technique proposed in [14]).

For our purposes, it suffices to consider only situations where the detector classifies a program as benign. Accordingly, we use a dummy detector that always reports "benign" in place for malware detection. The controller is implemented as browser extensions for both Firefox and Internet Explorer that interact with a local file based cache. We leave the implementation of environmental predicate based technique with a secure sandbox for future work.

Table 2 presents the summary of all 12 testcases and the experiment results of our prototype system for both Firefox and Internet Explorer as clients. The first several columns of Table 2 shows the summary of testcases. The column "Environment-Dependent Data Referenced" shows the environment-dependent data that used by the testcases to make decisions and change execution path at runtime. The column "Branch Mechanism" shows how testcases change the execution path based on environment-dependent data. As discussed in Section 3, we focus on five types of mechanisms, including if-else, loop, switch, function invocation and try-catch.

The experiment results are presented in two parts, one for Firefox and another one for Internet Explorer as clients (the environmental predicates evaluated in both clients are constructed by the same Firefox-based sandbox). For each client, the results are presented in two columns: "discrepancy?" and "detected?". The "discrepancy?" column indicates the existence of environment-dependent control flow discrepancy. In other words, it shows whether the difference in environment-dependent data would cause the testcases to behave differently when executed in sandbox and actual client browser. The "detected" column summarizes the results of using environmental predicate to detect such environment-dependent control flow discrepancy. It can be seen that, for testcases that would make environment-dependent decisions differently in sandbox and browser (with $\surd$ in "discrepancy?" columns), all discrepancies have been detected by executing their environmental predicate (with $\surd$ in "detected?" columns), including all branch mechanisms under evaluation, which gives the environmental predicate 100% detection rate on environment-dependent control flow discrepancy. With other testcases have identical environment-dependent data in two environments (with $\times$ in "discrepancy?" columns) all passed the test (with $\times$ in "detected?" columns), the environmental predicate based technique also achieves the 100% accuracy on 12 testcases. The overhead for performing analysis and constructing environmental predicates from execution trace for our 12 testcases is 8.77 millisecond in average, ranges from 1.73 millisecond to 38.3 millisecond.

6 Discussion

It is important to consider how an attacker who is aware of our approach and algorithm might try to evade it. One possible scenario would be to use timing-based techniques to detect the presence of runtime monitoring. This technique has been widely used for anti-analysis defense in native malware, but can also be applied against web-based malware detectors [9]. For example, JavaScript program execution under analysis of Wepawet [5] is significantly slower than in a browser.

We could revise our approach slightly to handle simple timing measurement defense. We observe that, the time value in Date object can be affected by all preceding code, even there is no data or control dependency between them. For example, when the time values are used for measuring time elapsed between given points in the program. Here is one such example:

```
var t0 = (new Date()).getTime();
... code to be timed ...
var t1 = (new Date()).getTime();
if ( t1 − t0 < T_threshold) { unpack and execute malicious code }
```

new Date() is not dependent on any code in the exam-

ple, but removing code between two occurrences of new Date() would make the time values obtained completely different thus changes the behavior of the program. To address this problem, we add all preceding code of the instruction which creates a Date object to obtain current time into slice, during the backward slicing in step 2 of our algorithm.

While this change is effective against simple timing defense shown above, in theory it may be possible to evade our approach by exploiting the small runtime overhead introduced by checkpoints. Specifically, attackers can intentionally create dependencies between the code to be timed and environment-dependent data such that, in the worst case, every statement in the environmental predicate has a checkpoint associated with it, thereby introducing a nontrivial overhead relative to original program (approximately a factor of 2 according to our test). Careful choice of execution time thresholds can cause the execution of the code with checkpoints to be slow enough to take the same (benign) path as in the sandbox, but the code without checkpoints running in the browser to be fast enough to trigger the malicious behavior. In this case, we would have a false negative. A similar issue arises with branches that are dependent on variable environment-dependent values, e.g., check if current time in milliseconds is divisible by 10. Then there is a (small) chance that sandboxed execution and evaluation of environmental predicate have same execution path, but the actual execution in the browser behaves differently. However, this kind of anti-analysis checks would significantly reduce the infection rate of the malware. One possible way to completely eliminate the false negatives discussed above is to insert checkpoints and decision vector directly into original web page. We leave an exploration of above issues as future research.

Another concern would be false positives, which can be caused by benign program branching on environment-dependent data. However, the study on code fragility in [11] shows only 1.2% of benign JavaScript files analyzed branch on environment-dependent data, in contrast to 89.5% among malware samples. This result suggests that the majority of benign programs are not environment-dependent. However, 1.2% of all benign files is still too large as the upper-bound of false positives.

To further reduce the possibility of false positives, a white-list based technique can be applied, such that only environment-dependent control-flow discrepancy caused by code from untrusted web sites are identified as suspicious by our approach. This is based on the observation that the landing pages of current attack are usually hosted on the servers under attackers' control, instead of compromised legitimate sites, to minimized the risk of being detected. In particular, the white-list based technique can potentially alleviate the problem of using web frameworks like jQuery (since they are usually referenced through content delivery networks for better performance), which perform environmental checks to hide implementation variations across browsers under a uniform API. However, this approach might increase false-negatives, if the malware perform environmental checks on white-listed web sites.

Attackers can also make malware triggered by user-driven events (e.g., mouse movement). Our analysis is done without user intervention, so the handling of user-driven scripts is determined by whatever strategy the detector uses. If the detector automatically triggers user-driven events during analysis in order to increase code coverage, any environment-dependent branches executed in user-driven script would be logged and then checked in environmental predicate as in non-user-driven code. However, if the detector does not automatically trigger user-driven events, exploits triggered by user-driven events might be missed. In either case, the detection rate is driven by the detector, not the environmental predicate.

7 Related Work

A number of malware detection approaches have been proposed in the research literature [8, 16, 5, 17, 7, 4]. To penetrate obfuscations, most of these approaches monitor program execution in a sandboxed environment, which often causes limited code coverage and non-trivial execution overheads. A separate but related issue is that it is not straightforward to construct a sandboxed environment that is completely transparent to the code executing within it.

To get around the limited code coverage problem of dynamic analysis, researchers have proposed various multi-path exploration techniques, mostly for native malware. Moser *et al.* [15] propose a system for native-code malware that executes all possible paths by using symbolic execution. Brumley *et al.* [3] use a combination of dynamic binary instrumentation and mixed symbolic and concrete execution, to identify behavior that is dependent on environmental triggers. Crandall *et al.* use a combination of VM-based timer perturbation and symbolic execution to discover time bombs in malware [6]. Detecting environment-dependent behavior in native malware has been discussed as well [12, 2], which involves comparing multiple executions in different environments.

Kolbitsch *et al.* propose Rozzle, a multi-execution framework for JavaScript, which efficiently explores multiple execution paths within one execution [11]. While Rozzle and our approach are both designed for detecting environment-dependent web-based malware, the principles behind are completely different. In ad-

dition, technique proposed in this paper is capable of handling more sophisticated malware cloaking mechanisms, such as implicit conditionals [14].

The work of Kiriansky *et al.* on program shepherding [10] has some conceptual similarities with this work. However, the specifics of their work are very different from ours: their primary focus is on dealing with native-code malware that carry out their attack by hijacking control flow, and they address this problem by using dynamic code rewriting to enforce security policies.

8 Conclusions

Environment-dependent malware, where the malicious behaviors are triggered only in specific environment, can be difficult to detect. Most current techniques for dealing with such malware depend on multi-path exploration techniques, which are generally limited to simple conditional branch based mechanisms and are expensive. This paper proposes a different technique for defending against environment-dependent malware which ensures the client will take the same execution path as the one examined by the detector. Evaluation shows that this technique can effectively detect environment-dependent behavior discrepancy of various forms, including those seen in real malware.

Acknowledgments

This work was supported in part by the National Science Foundation (NSF) via grant nos. CNS-1016058 and CNS-1115829, and the Air Force Office of Scientific Research (AFOSR) via grant no. FA9550-11-1-0191. The opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily reflect the views of AFOSR or NSF.

References

[1] H. Agrawal and J. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[2] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient detection of split personalities in malware. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[3] D. Brumley, C. Hartwig, Z. Liang, J. Newsome, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In *Botnet Detection*, volume 36 of *Advances in Information Security*, pages 65–88. 2008.

[4] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proc. 20th International Conference on World Wide Web*, pages 197–206, 2011.

[5] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *Proc. 19th International Conference on World Wide Web*, pages 281–290, 2010.

[6] J. Crandall, G. Wassermann, D. de Oliveira, Z. Su, S. Wu, and F. Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. In *Proc. 12th. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, Oct. 2006.

[7] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser JavaScript malware detection. In *USENIX Security Symposium*, 2011.

[8] M. Egele, P. Wurzinger, C. Kruegel, and E. Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proc. 6th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–106, 2009.

[9] F. Howard. malware with your mocha: Obfuscation and anti emulation tricks in malicious JavaScript. *Sophos Technical Papers*, 2010.

[10] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proc. 11th USENIX Security Symposium*, pages 191–206, 2002.

[11] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *IEEE Symposium on Security and Privacy*, May 2012.

[12] M. Lindorfer, C. Kolbitsch, and P. Comparetti. Detecting environment-sensitive malware. In *Proc. 14th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 338–357, Sept. 2011.

[13] G. Lu and S. Debray. Automatic simplification of obfuscated javascript code: A semantics-based approach. In *Proc. Sixth IEEE International Conference on Software Security and Reliability*, pages 31–40, June 2012.

[14] G. Lu and S. Debray. Weaknesses in defenses against web-borne malware (extended abstract). In *10th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, July 2013.

[15] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[16] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of the 18th conference on USENIX security symposium*, pages 169–186. USENIX Association, 2009.

[17] K. Rieck, T. Krueger, and A. Dewald. Cujo: efficient detection and prevention of drive-by-download attacks. In *Proc. 26th Annual Computer Security Applications Conference*, pages 31–39, 2010.

[18] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, pages 513–528, Oakland, CA, USA, 2010.

Original Program	After Checkpoint insertion

if (e){
S_1
} else {
S_2
}
	if (e){
	if($decisionVector[vecIndex++]$!= 0) throw *"wrong path"*;
	S_1'
	} else {
	if ($decisionVector[vecIndex++]$!= 1) throw *"wrong path"*;
	S_2'
	}

while (e){
S_1
}
	while (e){
	if ($decisionVector[vecIndex++]$!= 0) throw *"wrong path"*;
	S_1'
	}
	if ($decisionVector[vecIndex++]$!= 1) throw *"wrong path"*;

switch (e){
case c1:
S_1
case c2:
S_2
default:
S_3
}
	switch(e){
	case c1:
	if ($decisionVector[vecIndex++]$!= 1) throw *"wrong path"*;
	S_1'
	case c2:
	if ($decisionVector[vecIndex++]$!= 2) throw *"wrong path"*;
	S_2'
	default:
	if ($decisionVector[vecIndex++]$!= 0) throw *"wrong path"*;
	S_3'
	}

function FUNC-NAME (args...){
S_1
}
	function FUNC-NAME (args...){
	if ($decisionVector[vecIndex++]$!= "FUNC-NAME") throw *"wrong path"*;
	S_1'
	}

try{
S_1
} catch (e){
S_2
}
	try{
	S_1
	if ($decisionVector[vecIndex++]$!= 0) throw *"wrong path"*;
	} catch (e){
	if (e==*"wrong path"*) throw *"wrong path"*;
	if ($decisionVector[vecIndex++]$!= 1) throw *"wrong path"*;
	S_2'
	}

/* S_1 threw an unhandled exception during execution */ S_1	S_1 throw *"wrong path"*;

Table 1: Rules for Checkpoint Insertion Algorithm

	Environment-Specific Data Referenced	branch Mechanism	Results (Firefox)		Results (IE)	
			Discrepancy?	Detected?	Discrepancy?	Detected?
1	`navigator`	if-else	×	×	×	×
2	`navigator.userAgent`	if-else	×	×	√	√
3	`navigator.javaEnabled`	if-else	√	√	√	√
4	`navigator.language`	if-else	√	√	√	√
5	`navigator.systemLanguage`	if-else	×	×	√	√
6	`navigator.plugins`	loop & if-else	√	√	√	√
7	`navigator.mimeTypes`	loop & if-else	√	√	√	√
8	`ActiveXObject`	try-catch & if-else	×	×	√	√
9	`navigator.appVersion`	function	√	√	√	√
10	`navigator.appName`	switch	×	×	√	√
11	`Date`	implicit conditional	√	√	√	√
12	`ActiveXObject`	unhandled exception	×	×	√	√

√: environment-dependent control flow discrepancy exists or detected
×: no environment-dependent control flow discrepancy exists or detected

Table 2: Summary of Testcases and Experiment Results