ANALYSIS OF EVASION TECHNIQUES IN WEB-BASED MALWARE

by

Gen Lu

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCES
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

2013
As members of the Dissertation Committee, we certify that we have read the dissertation prepared by Gen Lu entitled Analysis of Evasion Techniques in Web-Based Malware and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy.

Date: 6 January 2014

Saumya Debray

Date: 6 January 2014

David Lowenthal

Date: 6 January 2014

John Hartman

Date: 6 January 2014

Christopher Gniady

Final approval and acceptance of this dissertation is contingent upon the candidate’s submission of the final copies of the dissertation to the Graduate College. I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Date: 6 January 2014

Dissertation Director: Saumya Debray
STATEMENT BY AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at the University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. This work is licensed under the Creative Commons Attribution-No Derivative Works 3.0 United States License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/3.0/us/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

SIGNED: Gen Lu
ACKNOWLEDGEMENTS

First and foremost, I would like to express the deepest appreciation to my advisor, Professor Saumya Debray, for his enthusiasm, patience and encouragement. Without his guidance and persistent help this dissertation would not have been possible.

Next I would like to thank the rest of my committee members, Professor David Lowenthal, Professor John Hartman and Professor Chris Gniady. I am grateful for their assistance and suggestions throughout my study. It has been an honor and a privilege to work with each of them.

I would also like to thank all my friends here at the University of Arizona who have made my life at grad school enjoyable. A special shout-out to Mingsong Bi, I have been truly fortunate to have a friend like him and I cannot imagine a better person to procrastinate with. Besides, many thanks to my friends Yi Lu, Kevin Coogan and Yun Liu.

I would not survive the pressure from research without some down time. As a wildcat for life, I would like to thank the coaches and players of Arizona basketball, football and baseball teams for all the fun they have brought to me. Bear down!

Finally and most importantly, none of this would have happened without the unconditional love and support from my family. I especially thank my grandma, grandpa, mom and dad. My loving grandparents had sacrificed a lot for me, I miss them so much. My parents have been the best someone can ever ask for in the whole world, I am who I am today all because of them. I thank my lovely cat, Maozhi, who has been there for me during some tough times. A special thank you to my wife and best friend in life, Xiaofei Qu, for making all of my hard work meaningful. She has always believed in me and stood by me every time I have been down. I love you, always and forever.
DEDICATION

For Xiaofei, with love and gratitude.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>8</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>9</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>10</td>
</tr>
<tr>
<td>CHAPTER 1 INTRODUCTION</td>
<td>11</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>12</td>
</tr>
<tr>
<td>1.2 Basic Approaches</td>
<td>14</td>
</tr>
<tr>
<td>1.2.1 Deobfuscating Web-Based Malware</td>
<td>14</td>
</tr>
<tr>
<td>1.2.2 Weaknesses in Current Malware Detectors</td>
<td>16</td>
</tr>
<tr>
<td>1.2.3 Defending Environment-Dependent Malware</td>
<td>18</td>
</tr>
<tr>
<td>1.2.4 Conclusions</td>
<td>20</td>
</tr>
<tr>
<td>CHAPTER 2 BACKGROUND AND RELATED WORK</td>
<td>21</td>
</tr>
<tr>
<td>2.1 Obfuscation and Anti-Analysis techniques</td>
<td>21</td>
</tr>
<tr>
<td>2.1.1 JavaScript Basics</td>
<td>21</td>
</tr>
<tr>
<td>2.1.2 JavaScript Obfuscation Techniques</td>
<td>23</td>
</tr>
<tr>
<td>2.1.3 Environment-Dependent Malware</td>
<td>25</td>
</tr>
<tr>
<td>2.2 Analysis and Detection of Web-Based Malware</td>
<td>27</td>
</tr>
<tr>
<td>CHAPTER 3 DEOBFUSCATION OF WEB-BASED MALWARE</td>
<td>30</td>
</tr>
<tr>
<td>3.1 Overall Approach</td>
<td>31</td>
</tr>
<tr>
<td>3.1.1 Semantics-Based Approach</td>
<td>31</td>
</tr>
<tr>
<td>3.1.2 System Overview</td>
<td>32</td>
</tr>
<tr>
<td>3.2 Semantics-Based Deobfuscation</td>
<td>34</td>
</tr>
<tr>
<td>3.2.1 Trace Collection</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2 Control Flow Graph Construction</td>
<td>36</td>
</tr>
<tr>
<td>3.2.3 Semantics-Based Deobfuscation</td>
<td>40</td>
</tr>
<tr>
<td>3.2.4 Decompilation</td>
<td>45</td>
</tr>
<tr>
<td>3.2.5 Code Transformation</td>
<td>46</td>
</tr>
<tr>
<td>3.3 Experimental Evaluation</td>
<td>47</td>
</tr>
<tr>
<td>3.4 Attack Models</td>
<td>57</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS – Continued

<table>
<thead>
<tr>
<th>CHAPTER 4</th>
<th>WEAKNESSES IN EXISTING DEFENSES</th>
<th>58</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Limitations of Existing Approaches</td>
<td>59</td>
</tr>
<tr>
<td>4.2</td>
<td>Thwarting Analysis</td>
<td>61</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Emulation-Based Obfuscation</td>
<td>62</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Anti-Analysis Defense</td>
<td>65</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Implicit Conditionals</td>
<td>66</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Implementation</td>
<td>70</td>
</tr>
<tr>
<td>4.3</td>
<td>Experimental Evaluation</td>
<td>70</td>
</tr>
<tr>
<td>4.4</td>
<td>Countermeasures</td>
<td>73</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 5</th>
<th>DETECTING ENVIRONMENT-DEPENDENT MALWARE</th>
<th>79</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Environmental Predicates</td>
<td>80</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Definition</td>
<td>81</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Construction of Environmental Predicate</td>
<td>83</td>
</tr>
<tr>
<td>5.1.3</td>
<td>Analysis of Environmental Predicate</td>
<td>89</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Implementation</td>
<td>94</td>
</tr>
<tr>
<td>5.2</td>
<td>Experimental Evaluation</td>
<td>96</td>
</tr>
<tr>
<td>5.3</td>
<td>Attack Models</td>
<td>99</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 6</th>
<th>CONCLUSIONS</th>
<th>102</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>APPENDIX A</th>
<th>Code Transformation Algorithm</th>
<th>105</th>
</tr>
</thead>
</table>

| REFERENCES |                            | 108 |
## LIST OF FIGURES

1.1 Structure of web-based malware ............................................. 12
2.1 Example of obfuscated JavaScript code .................................................. 23
2.2 Excerpt of an obfuscated malicious script ............................................. 25
3.1 Semantics-based deobfuscation: system overview ...................................... 33
3.2 Augmenting dynamic control flow graph .............................................. 37
3.3 Control flow graph structure for function calls/returns .......................... 38
3.4 The test programs $P_1$ and $P_2$ ......................................................... 47
3.5 Fragments of obfuscated versions of the program $P_1$ .......................... 48
3.6 Deobfuscator outputs for programs $P_1$ and $P_2$ ................................. 49
3.7 Unfolded code contexts from obfuscated version of programs $P_3$ ....... 51
3.8 Deobfuscator outputs for programs $P_3$ ............................................... 52
3.9 Execution flow of malware sample $P_4$ ................................................. 52
3.10 The first code context of $P_4$ .............................................................. 53
3.11 The second code context of $P_4$ .......................................................... 54
3.12 The third code context of $P_4$ ............................................................. 55
3.13 Deobfuscator output for malware sample $P_4$ .................................. 56
4.1 Example of an emulation-obfuscated program ....................................... 64
4.2 Comparison of code execution overhead in various environments ..... 66
4.3 Structure of proposed evasion techniques .............................................. 69
4.4 Comparison of running time between code-unfolding and emulation .... 73
4.5 Basic code embedding transformations .................................................. 75
5.1 Workflow of environmental predicate generation .................................. 83
5.2 An example of environmental predicate ................................................. 87
5.3 System architecture ................................................................. 95
# LIST OF TABLES

3.1 Running time of the deobfuscator for test programs .......................... 56
4.1 Description of malware samples .................................................. 71
4.2 Detection Results ................................................................. 72
5.1 Rules for checkpoint insertion algorithm ....................................... 88
5.2 Summary of testcases and experiment results ................................. 97
ABSTRACT

Web-based mechanisms, often mediated by malicious JavaScript code, play an important role in malware delivery today, making defenses against web-based malware crucial for system security. To make it even more challenging, malware authors often take advantage of various evasion techniques to evade detection. As a result, a constant arms race of evasion and detection techniques between malware authors and security analysts has led to advancement in code obfuscation and anti-analysis techniques. This dissertation focuses on the defenses against web-based malware protected by advanced evasion techniques from both defensive and offensive perspectives.

From a defensive perspective, we examine existing evasion techniques and propose deobfuscation and detection approaches to defeating some popular techniques used by web-based malware today. In the case of code-unfolding based obfuscation, we use a semantics-based approach to simplify away obfuscations by identifying code that is relevant to the behavior of the original program. In the case of environment-dependent malware, we propose environmental predicate, which detects behavior discrepancy of JavaScript program between targeted browser and detector sandbox, therefore protecting users from possible detection false negatives caused by environmental triggers. From an offensive perspective, we analyze existing detection techniques to examining their assumptions and study how these assumptions can be broken. We also propose a combination of obfuscation and anti-analysis techniques, targeting these limitations, which can hide existing web-based malware from state-of-the-art detectors.
CHAPTER 1

INTRODUCTION

The growing importance of the Internet in recent years has been accompanied by a corresponding increase in web-based malware delivery. This is typically done via “drive-by downloads” that exploit vulnerabilities in browsers and other client-side software [38, 39, 37]. Such attacks are often carried out via scripts written in JavaScript, a language commonly used in client-side web applications. Defending such attacks usually requires the ability to detect malicious JavaScript code. However, this is not easy: attackers generally use a variety of evasion techniques, such as code obfuscation and environmental triggers, to create code that is highly obfuscated and inscrutable, which renders traditional signature-based detectors ineffective. The work presented in this dissertation focuses on defenses against web-based malware protected by advanced evasion techniques from both defensive and offensive perspectives. More specifically, we propose approaches for handling two of the most widely used malware evasion techniques — code obfuscation and environmental triggers. To anticipate future attacks, we also systematically study the limitations of existing web-based malware detectors, and propose new evasion techniques targeting these limitations.

The dissertation is organized as follows. Chapter 1 gives a high level overview of the motivation, background, and our basic approaches and conclusions, and is meant to stand alone as a brief summary of the entire work. Chapter 2 gives a detailed look at relevant background information, including a discussion of evasion techniques widely used in web-based malware, and a review of any work by other authors that we build on. Chapter 3 details the semantic-based deobfuscation technique. Chapter 4 presents the study on the limitations of existing defenses and describes a combination of evasion techniques capable of bypassing current detectors. Chapter 5 presents a client-side defense against environment-dependent web-based malware.
Finally, Chapter 6 summarizes the findings that result from this work.

1.1 Motivation

The term malware, by definition, refers to software used to disrupt computer operation, gather sensitive information, or gain access to private computer systems without the owner’s consent. We have seen a tremendous increase in web-based malware delivery in recent years. The process, also known as drive-by-download, is usually done by JavaScript code capable of exploiting vulnerabilities in web browsers as well as plugins for technologies such as Java, PDF and Flash. The attack is conducted automatically when users open the malicious web page in their browsers.

Web-based malware usually consist of 3 components in a layered structure, as shown in Figure 1.1.

**Payload** This is what most people think of when heard the word “malware”. Payload is the centerpiece of web-based malware and is responsible for conducting actual malicious activities. For drive-by-download, it is usually some arbitrary code provided by attacker to be executed on victim’s machine.

**Infiltration** Most software have security vulnerabilities (e.g. bugs and insecure
designs) which is the key for malware delivery. The infiltration layer (also known as exploitation) is designed to exploit vulnerabilities in web browsers and their plugins and deliver the payload. Note that the possible ways for implementing infiltration is limited by the variety (or lack thereof) of vulnerabilities. Buffer overflow is by far the most popular approach which is usually work along with code injection techniques such as heap-spray [40] and return-oriented-programming [44].

Evasion Also known as cloaking or protection. The purpose of the evasion is to hide malware from detectors. Although it does not affect functionality, almost all modern web-based malware employ some kind of evasion techniques to increase the success rate of exploitation. There are various techniques can be applied, for example, code obfuscation and environmental triggers, usually against different defensive techniques. Typically, a detector has to penetrate the evasion layer to identify malware accurately.

The utilization of evasion techniques places malware authors in an advantageous position over the security analysts. Web-based malware is capable of infecting hundreds of thousands of machines in a short time. Therefore, most damage has been done even before the traditional anti-malware systems can react. It is not unusual that the malware hosting server is already offline when security analysts obtain the malicious URL. To make matters worse from the defensive perspective, evasion techniques such as code obfuscation and environmental triggers can help known malware evade detection.

To defend web-based malware, we clearly need to better understand the evasion techniques utilized by malware authors and develop tools that can effectively penetrate the evasion layer. The work presented in this dissertation attempts to handle two specific evasion techniques known to be used by existing web-based malware and anticipate possible techniques in future attacks.
1.2 Basic Approaches

We start by looking at the code-unfolding based obfuscation which is currently utilized by almost every piece of web-based malware. Our semantics based deobfuscation approach focuses on the observational behavior of target program and shows promising results against such obfuscations. Next, as an attempt to anticipate possible evasion techniques in future attack, we systematically study existing detection techniques for web-based malware and examine the assumptions they make. Experiment results based on a combination of evasion techniques we proposed show that the state-of-the-art defenses can be evaded by breaking their assumptions. Finally, we enhance current dynamic analysis based detectors by using environmental predicates to defend against environment-dependent malware. This approach is effective for detecting environment-dependent behavior discrepancy seen in real malware and is suitable for client side defense.

1.2.1 Deobfuscating Web-Based Malware

Code obfuscation makes identifying malicious JavaScript code a challenging task. First of all, the mere presence of obfuscated JavaScript does not, in itself, signal the presence of malicious content, since benign web pages also use code obfuscation to protect intellectual property. Moreover, attackers often use server-side scripting to deliver randomly obfuscated code where each instance is syntactically different from the next (server-side polymorphism). For these reasons, traditional static signature-based approaches have limited success when dealing with obfuscated web-based malware [20, 14, 22].

A better solution would be to focus on program behavior. It is also important and useful for making it possible for human analysts to easily understand the inner logic of obfuscated JavaScript code so as to deal quickly and effectively with any exploit no one ever seen before. Unfortunately, current behavioral analysis techniques for obfuscated JavaScript typically require considerable manual intervention, e.g., to modify the code in specific ways or to monitor its execution within a debugger
There has been some recent work on automated behavioral analysis of obfuscated JavaScript that in many cases has a “deobfuscator” component \([9, 12, 14, 22]\), as well as standalone JavaScript deobfuscation tools \([17, 35, 1, 50]\). These deobfuscators all rely on some simple and intuitive assumptions about the obfuscation. Although these assumptions seem plausible based on existing malware, it is not difficult to construct obfuscations that violate them and thereby defeat the corresponding deobfuscators.

We propose a semantics-based approach to analyze obfuscated JavaScript code (see Lu et al. \([27]\)). The idea is to identify code that contributes to the *observational behavior* of the program, i.e. the way it interacts with the external environment, and eliminate those are dedicated solely to obfuscation, while not assuming any information about the specific obfuscation structure.

More specifically, we first collect bytecode execution trace from the target program, then construct a dynamic control flow graph (CFG) from the trace to determine the structure of the executed code and conduct analysis to compute various information such as loop and function structures and control dependency.

Then we identify code that directly or indirectly affects the values of the arguments to built-in browser routines that appear as *native functions*, which we use as a proxy for observational behavior; these instructions are semantically relevant. This process is done by an algorithm we call deobfuscation slicing that does not follow dependencies through any code unfolding statements, therefore can automatically simplify the trace.

Next, we decompile the dynamic control flow graph to an abstract syntax tree (AST) and use semantics-preserving transformations to reconstruct deobfuscated JavaScript code from simplified trace. The code so obtained is observationally equivalent to the original program for the execution path considered, but has obfuscations simplified away, thereby exposing the core logic of the computation performed by the original code. The resulting code can then be examined manually or fed to other analysis tools for further processing. Our approach differs from existing approaches in that it makes no assumptions about the structure of the obfuscation and uses
semantics-based techniques to reveal the behavior of the code.

Experiments using a prototype implementation indicate that this semantics-based approach is able to penetrate multiple layers of obfuscations and extract the core logic of the underlying computation. While our current prototype is implemented in the context of the web browser. However, none of our ideas are specific to browsers, and in principle they could be adapted in a straightforward way to any environment that adopting dynamic scripting languages, for example, a PDF document reader.

1.2.2 Weaknesses in Current Malware Detectors

Experience with malware over the last two decades indicates that threats don’t go away: rather, they adapt to new platforms, targets, and opportunities, and evolve to work around improvements in defenses; it seems not unreasonable to believe that web-based malware delivery, mediated by malicious JavaScript, will not be different. Moreover, past experience suggests that the evolution of evasion techniques is very often driven by the desire to evade current defenses: attackers analyze existing detection techniques, identify implicit underlying assumptions, and devise techniques that violate these assumptions and thereby cause the defenses to fail.

Therefore, it is important to understand the weaknesses of current detection techniques, and anticipate possible evasion techniques malware might use to exploit those weaknesses. We explore this issue by first analysing existing detection techniques for web-based malware to examine their assumptions and study how these assumptions can be broken (see Lu et al. [28]).

A careful examination shows current approaches to detection of JavaScript malware boil down to determining whether a script “looks weird,” i.e., has characteristics peculiar to obfuscated or dynamically generated code, and/or “acts weird,” e.g., invokes known vulnerable functions, constructs shellcode-like strings or heap sprays during execution. The question, then, is: can script code be malicious and yet neither “look weird” statically nor (from an analyzer’s perspective, at least) “act weird” dynamically? The first objective can be realized by writing the script as an
interpreter—which looks just like ordinary JavaScript code—and embedding the malicious logic in the interpreted bytecode (this is similar to the idea of emulation-based obfuscation adopted in some native malware [47, 51]). The second objective can be realized by using anti-analysis technique, which profiles the performance to determine whether the script is being analyzed, and not reveal any malicious behaviors if it is (this technique is not new either, as it has been widely used in native malware for the same purpose). In addition, we propose a technique called implicit conditionals to forestall multi-path exploration techniques that might otherwise force execution into the malicious portion of program during analysis and bypass anti-analysis defense. As far as current detection tools are concerned, the resulting malicious code neither “looks weird” nor “acts weird”—i.e., it escapes detection.

More specifically, we first obfuscate the target program using emulation-based obfuscation: the original logic is transformed into bytecode of some arbitrarily chosen instruction-set and an interpreter for this specific instruction-set is attached. Second, the increment of the interpreter’s program counter is calculated based on the environment in which the code is running, thus effectively hide such environmental triggers in the bytecode interpretation instead of (typically) a branching statement in source code. As a result, the code protected by our approach shows no common characteristics from existing obfuscation techniques, and only behaves maliciously in the environment anticipated by attacker without revealing itself to any of the existing multi-path exploration techniques.

To prove emulation-based obfuscation and implicit conditionals can become a real threat against existing detectors, we applied our proposed technique on 7 real web-based malware and then evaluated its effectiveness by submitting these obfuscated malware samples to state-of-the-art detectors, including Wepawet [12], Zozzle [14] and VirusTotal [56]. In our experiments, our technique successfully protected all the malware samples and none of them has been identified as malicious/suspicious by targeting detectors.
1.2.3 Defending Environment-Dependent Malware

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. To penetrate code obfuscations, most existing web-based malware detectors rely on dynamic analysis which is usually conducted in a secure sandbox outside of the actual web browser. Due to the implementation of the sandbox, environmental discrepancy between real browser and the sandbox may exist, even when both browser and sandbox run on the same machine, therefore such environment-dependent malware can lead to limited code coverage during analysis. This is a fundamental limitation for existing detectors.

To address this issue, researchers proposed multi-path exploration techniques to explore multiple execution paths which 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 [43, 30, 8]. More recently, Kolbitsch et al. [24] proposed Rozzle, a lightweight JavaScript multi-execution framework, which explores multiple execution paths within one execution instead of exhaustively executing all possible paths. However, as shown in Chapter 4 of this dissertation, it still suffers from limitations that can be exploited by malicious code to escape detection.

We propose a different technique, called *environmental predicates*, for defending environment-dependent malware (see Lu et al. [26]). 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. The idea is that if a sandbox
based detector determines that an execution path $p$ is benign, and we can determine that the in-browser execution will not deviate from $p$, then we can conclude that the in-browser execution will also be benign. More specifically, the execution path discrepancy is tested by using environmental predicate that is derived from the original program using information collected from the sandboxed execution.

The first step of the construction of environmental predicate is to run the target program in the detector’s sandbox and collect its execution trace. If a document is found to be malicious in the course of the sandboxed analysis, then the analysis is done. Otherwise we construct a dynamic control flow graph (CFG) from the trace to determine the structure of the executed code and conduct analysis to compute various information such as loop and function structure and control dependency.

The next step is to use a program analysis technique known as dynamic slicing to identify code that is relevant to environment-dependent branches in the program’s execution trace. Our algorithm uses a combination of forward and backward slicing, both adapted for handling the dynamic bytecode trace generated by our sandbox. In addition, all the environment-dependent branch decisions have to be logged in execution order and stored in decision vector.

Next, we decompile the dynamic control flow graph to an abstract syntax tree (AST), label all the nodes constructed from a set of relevant code resulted from last step, and use semantics-preserving transformations to eliminate goto statements. A checkpoint is inserted at each of the branch targets for all environment-dependent branches, to determine if the execution would follow the exact same path seen in sandboxed analysis.

Finally, we traverse the AST to generate environmental predicate based on labeled (relevant) syntax tree nodes. The environmental predicate 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.

Initial tests indicate good success with environmental predicate based approach. Experiment results show our prototype system is able to handle various branching mechanisms, including indirect control transfers and environment-dependent loops
that can be problematic for existing multi-path exploration techniques, and is effective for detecting environment-dependent behavior discrepancy seen in real malware.

1.2.4 Conclusions

This dissertation focuses on the defenses against web-based malware from both defensive and offensive perspectives. From a defensive perspective, it examines existing techniques and proposes deobfuscation and detection approaches to defeating some popular evasion techniques used by web-based malware today. In the case of code-unfolding based obfuscation, it uses a semantics-based approach to identify code that relevant to the behavior of the original program and simplifies away obfuscations. In the case of environment-dependent malware, it identifies behavior discrepancy of JavaScript program between targeted browser and detector sandbox by using environmental predicates, therefore protecting users from possible detection false negatives caused by environmental triggers. From a offensive perspective, it analyzes existing detection techniques for web-based malware to examine their assumptions and studies how these assumptions can be broken, and a combination of obfuscation and anti-analysis techniques, targeting various existing detectors, has been proposed.
CHAPTER 2

BACKGROUND AND RELATED WORK

This dissertation examines the defenses against web-based malware from the perspectives of both defenders and attackers. This chapter begins with an overview of evasion techniques, then provides a brief discussion on existing approaches for analysis and detection of web-based malware. Chapter 3 will detail our approach to handling code unfolding based obfuscation. Chapter 4 will detail our study on the limitations of existing defenses and approach to thwarting those techniques. Chapter 5 will detail our approach to detecting behavior discrepancy caused by environmental triggers utilized in web-based malware.

2.1 Obfuscation and Anti-Analysis techniques

This section first provides an overview of the JavaScript language and the host environment of web browser, then describes some widely used real-world code obfuscation and anti-analysis techniques.

2.1.1 JavaScript Basics

The term JavaScript, commonly used to refer to a scripting language used for client-side programming of dynamic websites, consists of the core programming language together with the host environment, namely, the Document Object Model (DOM) provided by web browser.

The core JavaScript language provides a set of data types (e.g. Boolean, String, Object), a collection of built-in objects and functions (e.g. RegExp, Math, Date), and a prototype-based inheritance mechanism, among other things. Like most scripting languages, JavaScript is highly dynamic in nature. It is dynamically typed, which means that a variable can take on values of different types at different points in a
program. Properties of (associative-array based) objects can be added/deleted on
the fly, and code can be generated from strings at runtime using the built-in \texttt{eval}
function.

The Document Object Model (DOM) is an API that abstracts HTML documents
as a structural representation of objects and provides a mechanism for manipulating
this abstraction, thereby enabling JavaScript code to modify and interact with the
content of web pages dynamically. For example, \texttt{write} method of the \texttt{document}
object can be used to dynamically write HTML expressions or JavaScript code to
a document. In contrast to the built-in objects defined in core JavaScript, objects
defined in the DOM specification are called “host objects”, and are provided as a
part of the host environment by the web browser.

At the implementation level, JavaScript typically uses an expression-stack-based
byte-code interpreter; modern implementations of these interpreters usually come
with just-in-time (JIT) compilers. For example, Mozilla’s popular FireFox web
browser uses an open source JavaScript interpreter, SpiderMonkey, written in
C/C++ [33]. This is a single dispatching function that steps through the byte-
code one instruction at a time.

As discussed above, client-side JavaScript programs have the ability to generate
code at runtime, using various mechanisms provided by both the interpreter and
browser. Further, dynamic code generation can be multi-layered, e.g., a string
that is \texttt{eval}-ed may itself contain calls to \texttt{eval}, and such embedded calls to \texttt{eval}
can be stacked several layers deep. We refer to such dynamic code generating as \textit{code
unfolding}, and for each piece of code generated by runtime unfolding, we call it a \textit{code
context}. Functions that are defined in JavaScript (using the keyword ‘\texttt{function}’) are
called \textit{non-native functions}; and functions provided by the interpreter or browser
(e.g. built-in functions and methods of host objects) are called \textit{native functions}.
Unlike non-native functions, native functions do not generate a bytecode trace when
executed.
2.1.2 JavaScript Obfuscation Techniques

The dynamic nature of JavaScript code makes possible a variety of obfuscation techniques. What’s particularly challenging is the combination of the ability to execute a string using code unfolding, as described above, and the fact that the string being “executed” may be obfuscated in a wide variety of ways. Howard discusses several such techniques in more detail [20]. For example, the characters in the string can be encoded in various ways, e.g., using %-encoding (a as %61, b as %62, ...), Unicode (a as \u0061, b as \u0062, ...), Base-64, etc. The string can be kept in encrypted, compressed, or permuted form. It can be constructed at runtime by concatenating other strings together. Besides, in addition to the traditional “dot notation” (obj.property) for object access, one can use a “bracket notation” (obj["property"]) instead. In the latter case, moreover, the use of a string as an array index makes applicable all of the string obfuscation techniques mentioned earlier.

The host environment of web browser also provides various options for obfuscation. One approach is to split code into several parts, either in the same file or even into multiple files stored among web servers. This technique is frequently seen with web-based malware. Another approach takes advantage of DOM interaction. For example, data can be stored in the HTML file, outside the <script> block, then retrieved using document.getElementById() at runtime. And, of course, docu-
\texttt{ment.write()} is a more powerful weapon than \texttt{eval()}, which can be used in combination of those obfuscation techniques mentioned above, to generate script, document elements storing data and pointers to external documents, all at runtime.

Figure 2.1 presents an example of JavaScript code obfuscated using some of the techniques discussed above. Line 3 of this code snippet uses bracket notation to reference object property, using the strings \texttt{getElementById} (obtained as the concatenation of the strings \texttt{get}, \texttt{Elem}, \texttt{ent}, \texttt{By}, and \texttt{Id}) and \texttt{innerHTML} (obtained as the concatenation of the strings \texttt{i}, \texttt{nn}, \texttt{erH}, and \texttt{TML}) as array indices instead of using the more straightforward dot-notation to obtain the corresponding property values. Furthermore, it uses the DOM method \texttt{document.getElementById()} to retrieve data, namely, the string \texttt{‘HelloWorld’.} For simplicity of exposition, this code uses a very straightforward obfuscation of the array index strings, namely, concatenation of a few smaller strings; the code could, however, just as easily have used arbitrarily more complex obfuscations to construct these strings. The script is equivalent to \texttt{var c = document.getElementById(‘x’).innerHTML}. The value finally assigned to variable \texttt{c} is a string \texttt{‘HelloWorld’} which is retrieved from the HTML \texttt{<div>} element with ID \texttt{‘x’}.

Those obfuscation techniques can be combined in arbitrary ways with multi-layered code unfolding, which makes it difficult to determine the intent of a JavaScript program from a static examination of the program text. Figure 2.2 shows an example of obfuscated JavaScript from a real malicious web page. To make it more challenging, the payload can be scattered in multiple code contexts at different levels, with each piece using various obfuscation techniques and hidden in the garbage code whose only purpose is to confuse deobfuscators. This trick can defeat existing JavaScript deobfuscators, which assume the unobfuscated, complete payload is revealed in one of the (typically the last) unfolded JavaScript code contexts.

There are also tools available for reducing the size of scripts [19, 61], usually by removing unnecessary whitespaces and comments, and renaming symbols. This technique is called code compression or minification, although it makes code difficult
aa=(document['createDocumentFragment']+'asd').
substr(2-1,4);if(aa="unct") {ss="";s=String;
e=window[\'e\'+\'val\'];t=\'b\';}ddd=new Date();
d2=new Date(ddd.valueOf()-2);h=(ddd-d2)*-1;
n="59.5b5 ... 49.5b26b17"; n=n[\"split\"](t);
for(i=0;i<n.length;i++) ss+=s[\'fromCharCode\'](-h*n[i]);zx=ss;if(aa="unct")e(zx);

Figure 2.2: Excerpt of an obfuscated malicious script

to read, the behavior is still apparent. Therefore, we don’t consider code minification as obfuscation.

2.1.3 Environment-Dependent Malware

In addition to the code unfolding based obfuscation discussed above, anti-analysis techniques are also utilized by malware authors to thwart detectors. In particular, one such anti-analysis technique is to make the malware environment-dependent, which uses characteristics of malware’s execution environment to determine whether, or how, the malicious code is deployed, such that the malicious behavior is exposed only when the malware is executed in the right environment. Malware utilizing this technique is called environment-dependent malware. This technique was first seen in native malware, which forms the basis for time bombs and logic bombs, where the malicious logic is activated on specific dates or times [13], or if the execution environment satisfies specific criteria. For example, some early malware code would only trigger their malicious actions on specific dates, e.g., the Jerusalem virus, which first came out in 1987, only deleted files on Friday the 13th. Modern malware often use timing checks to determine whether they are being executed in an emulator, and avoid exposing their malicious content if they are, thereby achieving greater stealth. The simplest and most direct approach for this is to simply test the environmental inputs against expected value(s). Environment-dependent code has been a widely used technique for web-based malware as well. Kolbitsch et al. show that 89% of web-based malware are environment-dependent [24]. For example, the following
code is a fragment of real malware, which retrieves and executes the malicious code if the system has Flash installed:

```javascript
try{
    x=new ActiveXObject
        ("ShockwaveFlash.ShockwaveFlash");
}catch(e){}
if(x){ retrieve and execute malicious code... }
```

One drawback with straightforward code like this is that it is relatively straightforward for automated tools to identify and analyze, e.g., using multi-path explorations techniques [30] to force execution into the branch of the conditional that causes the malicious code to be exposed.

Environmental inputs are also often involved in targeted attacks where the malicious functionality is activated only for specific users, groups, or countries [10, 18, 55]. Some malware use location information to customize the message used to dupe victims [49]. Song et al. discuss the use of environment-sensitive techniques for obfuscation purposes and to hinder automated malware analysis [48]. Howard discusses environment-dependent anti-analysis techniques used in JavaScript malware [20].

A variety of more sophisticated schemes can be used to make this more difficult. Sharif et al. [45] proposed conditional code obfuscation to conceal trigger-based behavior in native-code executables from multi-path exploration. Their technique uses a key generated from the input value to dynamically decrypt conditional code block, such that only the correct input can recover original code, otherwise the program will crash. While it still relies on environmental-dependent conditionals, the usage of cryptographic hash function ensures that the correct key cannot be recovered from its hard-coded hash value.

Anti-detection techniques against honeyclients have also been studied before. Kapravelos et al. [23] proposed a number of techniques to hide drive-by-download attacks. Those techniques, some of which also exploit the differences between defensive systems and regular browsers, are specifically designed for evading detection
of high-interaction honeyclients.

2.2 Analysis and Detection of Web-Based Malware

Web-based malware have received a lot of attention from security researchers in recent years, and a number of different approaches for analyzing and detecting such malware have been proposed. This section provides a brief introduction on those techniques.

Current analysis techniques for obfuscated JavaScript typically require considerable manual intervention, e.g., to monitor its execution within a debugger and to modify the code by hand [29, 34, 60]. Some automated JavaScript deobfuscation techniques have been proposed. The most widely used automatic deobfuscation technique, which is adopted and implemented by many JavaScript analyzers and malware detectors [12, 14, 22, 17, 35, 1, 50], is designed specifically for current code-unfolding based obfuscations. In general, those deobfuscators collect all the code contexts compiled by JavaScript engine (or similarly, all the strings passed to any code unfolding functions such as `eval()` and `document.write()`.) This simple technique is based on the observation from existing malware that after it is completely unfolded, the original malicious content would be revealed unobfuscated in one of the contexts. However, it is possible to construct code-unfolding based obfuscations that violate this assumption and thereby defeat the corresponding deobfuscators. Lu et al. [27] described a way for such obfuscation and proposed a semantics-based approach as an attempt to address code-unfolding obfuscation in general.

Several high-interaction honeyclients have been proposed in the literature [31, 37, 58, 36, 32]. They typically detect attacks by directing vulnerable browser to potential malicious web sites, and monitor changes to the operating system, which usually requires instrumented virtual machines. Therefore honeyclients-based approaches are not suitable for online detector deployed at Internet users’ systems.

As shellcode being an integral part of most JavaScript malware, shellcode detection has long been used by traditional anti-virus software. They work by statically...
identifying shellcode-like strings in a program. However, current obfuscation techniques render this approach ineffective. Inspired by this technique, many dynamic approaches have been proposed [16, 40, 54]. Instead of scanning static code text, those approaches execute target programs in an emulated environment and monitor allocated memory objects to identify the existence of possible shellcode strings at runtime.

Recent years have seen a great deal of interest in the use of machine-learning techniques to detect malicious JavaScript [12, 41, 14, 9]. These approaches focus primarily on “features” of static and/or dynamic aspects of the code, and their classifiers are trained on existing JavaScript code. Canali et al. [9] proposed a completely static approach which detects malicious web page base on syntactic features extracted from HTML, JavaScript and URL. This approach can process large number of web pages in high speed but suffers high false-positive rate. Curtsinger et al. [14] proposed Zozzle, a mostly static approaches for JavaScript malware detection. Zozzle identifies malicious JavaScript based on features automatically extracted from the abstract syntax tree, which is constructed from the complete code contexts collected by an instrumented web browser. Cova et al. [12] described a dynamic analysis system called JSAND, which works by monitoring the execution behavior of JavaScript in an emulated environment and recording various execution features, those recorded features are then passed to a trained classifier for analysis. Rieck et al. [41] presented another machine-learning based approach which combines both static and dynamic analyses. Like JSAND, this approach analyze recorded JavaScript execution from an emulated environment, but its analysis is not as broad, therefore it can achieve comparable detection rates with much better runtime performance.

Web-based malware often use environmental trigger to hide malicious content in invulnerable environment or hold attack until some specified conditions are met (e.g. time-bomb and logic-bomb). Environmental trigger usually leads to limited code coverage, thus is a fundamental limitation for both static and dynamic approaches discussed above.
While the powerful symbolic execution based techniques have been applied to explore multiple execution paths in malware [8, 30], the overhead makes it difficult to adopt them for large-scale or online detection.

The detection of environment-dependent malware is a challenging problem. Current work along these lines, generally follow two approaches. One line of work uses symbolic execution, in conjunction with dynamic instrumentation and/or monitoring techniques in a sandboxed environment, to explore alternative execution paths. Brumley et al. [8] 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. [13] use a combination of VM-based timer perturbation and symbolic execution to discover time bombs in malware. While the powerful symbolic execution based techniques have been applied to explore multiple execution paths in malware, the overhead makes it difficult to adopt them for large-scale or online detection. To address the performance issue, Kolbitsch et al. [24] propose a lightweight JavaScript multi-execution framework which focuses on environment-dependent conditionals to improve detection rates of existing malware detectors. The other approach involves running the code in multiple different environments and comparing their execution behaviors Lindorfer et al. [25] compare malware interactions with the operating system across multiple different sandboxes to identify environmental dependencies. Balzarotti et al. [5] discuss a conceptually related approach where system call logs are used to compare a program’s behavior in a sandbox with that on a reference machine. Both approaches are fairly heavyweight and incur nontrivial execution overhead.

A number of authors have described approaches to analysis of emulation-obfuscated native-code executables [46, 42, 11], which all rely on collecting and analyzing dynamic execution traces of the code under consideration. Because trace collection and analysis can incur significant overheads, these algorithms are currently best suited for off-line analyses. The development of efficient on-line algorithms that can penetrate layers of obfuscation and expose the internal logic of the malware code in a way that permits effective analysis is an area of ongoing research.
Section 2.1 presents the basic approach for code unfolding based obfuscation, which is a technique widely used by malware author. The obfuscation makes identification of malicious JavaScript challenging. The mere presence of obfuscated JavaScript does not, in itself, signal the presence of malicious content, since benign web pages also use code obfuscation to protect intellectual property. Moreover, attackers often use server-side scripting to deliver randomly obfuscated code where each instance is syntactically different from the next (server-side polymorphism). For these reasons, static signature-matching based heuristics have limited success when dealing with obfuscated JavaScript. Traditional anti-virus tools that process web pages typically rely on such syntactic heuristics and so tend to produce a high misidentification rate.

There has been some recent work on automated behavioral analyses of obfuscated JavaScript that in many cases has a deobfuscator component, as well as standalone JavaScript deobfuscation tools. These deobfuscators all rely on some simple and intuitive assumptions about the obfuscation and the structure of the obfuscated code. Although these assumptions seem reasonable for existing web-malware, it is not difficult to construct obfuscations that violate them and thereby defeat the corresponding deobfuscators.

Our approach, which is published in SERE 2012 (see Lu et al. [27]), uses a semantics-based technique to analyze obfuscated JavaScript code. In particular, We collect bytecode execution traces from the target program and use dynamic slicing and semantics-preserving code transformations to automatically simplify the trace, then reconstruct deobfuscated JavaScript code from simplified trace. The code so obtained is observationally equivalent to the original program for the execution path considered, but has obfuscations simplified away, thereby exposing the core
logic of the computation performed by the original code. The resulting code can then be examined manually or fed to other analysis tools for further processing. Our approach differs from existing approaches in that it makes no assumptions about the structure of the obfuscation and uses semantics-based techniques to reveal the behavior of the code.

The remainder of this chapter is organized as follows. Section 3.1 discusses the theoretical basis and the overall structure of our approach. Section 3.2 details our semantics-based deobfuscation technique. Section 3.3 discusses the results obtained by running the prototype implementation of our system against several test files. And Section 3.4 discusses the possible attack models.

3.1 Overall Approach

3.1.1 Semantics-Based Approach

Deobfuscation refers to the process of simplifying a program to remove obfuscation code and produce a simpler and functionally equivalent program. Static analysis of obfuscated JavaScript code usually cannot penetrate (possibly multiple levels of) dynamic code unfolding to examine the actual code that is materialized at runtime. Most existing JavaScript deobfuscators therefore resort to simple dynamic techniques [12, 14, 17, 35, 50]. These techniques assume the original logic would be revealed in one of the (typically the last) unfolded JavaScript code contexts completely, therefore they would have difficulty to automatically handle programs using more general obfuscations based on code unfolding. For example, the original logic can be scattered in multiple code contexts at different levels, with each piece obfuscated differently and hidden in the garbage code whose only purpose is to confuse deobfuscators.

The motivation of our approach is from the semantic intuition behind the de-obfuscation process. In general, we cannot expect deobfuscation to produce the original source code for the program, either because the source code is unavailable, or due to code transformations applied during obfuscation. All we can require,
then, is that the process of deobfuscation must be semantics-preserving: i.e., that the code resulting from deobfuscation be equivalent to the original program. For the analysis of potentially-malicious code, a reasonable notion of equivalence is that of observational equivalence, where two programs are considered equivalent if they behave—i.e., interact with their execution environment—in the same way. Since a program's runtime interactions with the external environment occur through system calls, this means that two programs are observationally equivalent if they execute identical sequences of system calls (together with the argument vectors to these calls).

This notion of equivalence suggests a simple approach to deobfuscation: identify code that directly or indirectly affects the values of the arguments to system calls; these instructions are “semantically relevant”. Any remaining instructions, which are by definition semantically irrelevant, may be discarded. For the JavaScript code considered in this dissertation, the actual system calls are typically made from built-in browser routines that appear as native functions. Our implementation therefore uses native functions as a proxy for system calls: this is sound, but potentially conservative since not all native functions lead to system calls. Then, to identify instructions that affect the values of native function arguments, we use dynamic slicing, applied at the bytecode level. One of the advantages of doing analysis at bytecode level is that the JavaScript compiler does part of the job for us: many obfuscation techniques used to confuse human analysts or automated script parsers can be revealed or removed after compilation. Examples of such tricks are discussed in [60, 62].

3.1.2 System Overview

Our approach to deobfuscating JavaScript code consists of the following steps, as shown in Figure 3.1:

1. [Trace Collection] Use an instrumented web browser to obtain an execution trace for the JavaScript code under analysis.
2. **[Control Flow Analysis]** Construct a dynamic control flow graph from collected trace to determine the structure of the executed code.

3. **[Deobfuscation Slicing]** Use our deobfuscation slicing algorithm to identify semantically relevant instructions, i.e., instructions that affect the externally-observable behavior of the program. As previously discussed, externally-observable behavior is carried out by native functions.\(^1\)

4. **[Decompilation]** Decompile the dynamic control flow graph to an abstract syntax tree (AST) and label all the nodes constructed from resulting set of relevant instructions.

5. **[Code Transformation]** Use semantics-preserving transformations to eliminate invalid source-level constructs such as `goto` statements. Finally, generate deobfuscated source code by traversing the AST and printing only labeled (relevant) syntax tree nodes.

Our current implementation separates trace collection from the remaining steps: the generated trace is written out to a file, which is then read by the trace analyzer. This is purely for convenience, since it is conceptually straightforward to build the analysis facilities directly into the web browser. Our current implementation writes out the abstract syntax tree obtained at the end of the above process in the form of

\(^{1}\)Ideally, we would like to compute slices for the arguments of the system calls made by the program. However, the actual system calls are typically made from built-in browser routines that appear as native functions. As a proxy for system calls, therefore, our implementation computes slices for the arguments passed to any native function.
JavaScript source code, but one can also imagine directly applying other malware analysis tools to the syntax tree itself.

3.2  Semantics-Based Deobfuscation

In this section, we describe the concept of semantics-based deobfuscation and the architecture of our prototype system in more detail. In particular, we discuss how we collect execution information of JavaScript programs at runtime, and how we use dynamic slicing technique to identify semantically relevant code from obfuscated script.

3.2.1  Trace Collection

We use an instrumented Mozilla Firefox web browser to collect the program’s execution trace. The JavaScript source code is first compiled to bytecode by Firefox’s JavaScript engine, SpiderMonkey, it then executes it using an interpreter. Since obfuscations commonly used by malware take advantage of the built-in functionality of JavaScript interpreter as well as document related operations provided by the browser, our instrumentation covers both the interpreter and Document Object Model (DOM) API.

Each bytecode instruction instance generated by our instrumented web browser includes instruction’s address, opcode mnemonic, length (in bytes), and operands, together with any additional information about the instruction that may be relevant. In particular, we print the following information, which is used for subsequent steps of the deobfuscation:

- expression stack: set of memory locations on the stack that are read and/or written by the instruction;

- constants: encoded as part of the bytecode instruction, can be an integer or a reference to an object (e.g., string and floating point number), for the latter case, the actual value of constant is retrieved and printed instead of the reference;
- *conditional and unconditional branches*: the offset (relative to the current instruction) of the branch target.

- *global variables, array elements, and object property accesses*: which property of which object is being defined or used;

- *local variables and arguments*: an index specifying which local variable or argument amongst the function’s locals is being accessed;

- *function calls*: the reference to the callee (function object) and the number of arguments being passed, together with a flag indicating whether the callee is a native function;

- *function references*: the reference to the function object in which this instruction belongs to. This information is used to identify functions which is not called explicitly;

- *document.write flags*: a flag indicating current instruction is a call to `document.write()` function;

- *document elements*: the reference to the document element that is created or accessed by functions such as `getElementById()`;

- *unfolded code*: string passed to code generating operations (i.e. `eval()`, `document.write()`, etc.)

As discussed in Section 2.1, the execution of a non-native function generates a bytecode trace while a call to a native function does not. However, the call to the non-native function can not be determined merely by the existence of bytecode trace. There are native functions take other functions as arguments, i.e. callbacks. These callback functions are invoked implicitly by the native function and generate bytecode trace, makes it similar to the execution of a non-native function. One such example is `string.replace()`, it takes a callback function as argument, the callback will be invoked after the match has been performed. The callback result (return value)
will be used as the replacement string. Therefore, a flag is used to distinguish calls to native and non-native functions, and for each instruction instance in the trace, we use the function reference to indicate in which function this instruction belongs, in order to associate the execution and definition of callback functions. The callback functions can also be invoked inside a non-native function, in that case, no special handling is required, therefore, in this dissertation, the term \textit{callback} is only used to refer to non-native callback functions invoked implicitly by native function.

The references to document elements and \texttt{document.write} flags are used to handle obfuscations involving DOM operations, which is opaque to JavaScript interpreter, but is crucial for the purpose of deobfuscating JavaScript in web pages: HTML document elements can be created and modified dynamically, and are often used for storing data by obfuscated JavaScript programs (e.g., see Figure 2.1). Unlike \texttt{eval()}, which is directly translated to an \textquote{eval} bytecode instruction, a call to \texttt{document.write()} is indistinguishable from other native function calls. The \texttt{document.write} flag is used by our deobfuscation slicing algorithm to establish the connection between HTML document and JavaScript code.

\subsection*{3.2.2 Control Flow Graph Construction}

In principle, obtaining the static control flow graph (CFG) for a JavaScript program is possible. JavaScript source code is compiled into bytecode before execution, and it is straightforward to decompile this bytecode to an abstract syntax tree. In practice, the control flow graph so obtained may not be very useful if the intent is to simplify obfuscations away. The reason for this is that dynamic constructs such as \texttt{eval()}, commonly used to obfuscate JavaScript code, are essentially opaque in the static control flow graph: their runtime behavior—which is what we are really interested in—cannot be easily determined from an inspection of the static control flow graph. For this reason, we opt instead for a dynamic control flow graph, which is obtained from an execution trace of the program. However, while the dynamic control flow graph gives us more information about the runtime behavior of constructs such as \texttt{eval()}, it does so at the cost of missing information, which might cause lower
accuracy and errors for our deobfuscation technique. As a result, we modify the algorithm for static CFG construction, found in standard compiler texts [4], to deal with dynamic execution traces and construct a dynamic CFG, plus the dominator analysis to identify loops and compute control dependencies, which is necessary for deobfuscation step, as shown in Section 3.2.3.

First, dynamically collected execution trace has a reduced code coverage, which might introduce errors to control dependency computation. For example, consider the static CFG shown in Figure 3.2. Basic block $B_1$, $B_2$, $B_3$ and $B_4$ represent the logic of a two-way branch, in which $B_2$ and $B_3$ both control dependent on $B_1$. If a specific execution does not execute $B_3$, the resulting dynamic CFG constructed from the execution trace will contain only the blocks $B_1$, $B_2$ and $B_4$, as indicated by the solid edges. In this graph, however, $B_2$ is not control dependent on $B_1$. Such missing control flow information thus has the effect of producing incomplete control dependence information between basic blocks. This, in turn, can affect the correctness of the deobfuscation slicing algorithm discussed in next section. To address this problem, our CFG constructing algorithm also inspect static code and augment the dynamic CFG with dummy basic blocks and edges to ensure the correctness of control-dependency.
In addition, the dynamic control flow graph obtained is also missing some information necessary for transformation to a higher-level representation. In particular, the basic blocks in the program are not grouped into different functions. In order to identify the basic blocks belonging to each function, we first associate each function call block with the corresponding block to which control returns from the call.

Figure 3.3 shows the structure of control flow edges for function calls and returns in the control flow graph constructed after slicing. Consider a basic block $B_{\text{call}}$ ending in a call instruction. For each target for this call (in general there may be more than one) there is a control flow edge to the entry block $B_{\text{entry}}$ of the callee. Suppose that control transfer from this callee back to the caller at the end of this call occurs from a basic block $B_{\text{exit}}$ in the callee to a block $B_{\text{ret}}$ in the caller, as shown in Figure 3.3. In general, a call may target more than one callee; a function may be called from more than one call site; and different calls to a function may return from different basic blocks within that function. Thus, $B_{\text{call}}$ and $B_{\text{exit}}$ may have multiple outgoing edges, and $B_{\text{entry}}$ and $B_{\text{ret}}$ may have multiple incoming edges. Our objective is to link together the call block $B_{\text{call}}$ and the corresponding return
block $B_{ret}$ to which control returns at the end of the call. We proceed as follows:

1. Suppose that the call instruction at the end of $B_{call}$ is at address $A$ and is $n$ bytes long. This means that the instruction that comes immediately after it in the program’s bytecode, and to which control returns from the call, is at address $A + n$. We search the control flow graph to find the basic block $B_{ret}$ whose first instruction is at address $A + n$.

2. We remove all the outgoing edges from $B_{call}$ (however, we retain a list of the call targets for the call block) and the incoming edges into $B_{ret}$. We add an edge from $B_{call}$ to $B_{ret}$; this edge indicates that the call from $B_{call}$ returns to $B_{ret}$.

After all call blocks have been processed in this way, the control transfer edges out of each call block and the control transfer edges into the corresponding return block are replaced by a single edge from the call block to the return block. For each function, the basic blocks belonging to that function are then computed as the set of blocks that are reachable from the entry node for the function.

One particular challenge for JavaScript dynamic CFG construction is how to deal with code generated dynamically. In order to distinguish code used for generating other code at runtime, from code used for other computation, we treat dynamically unfolded code similar to the way we handle non-native functions: a separate CFG is constructed for each piece of dynamically unfolded code, as the function body; and the unfolding instruction (e.g. `eval()`) is treated as a call to the function. The adoption of function-like structure for unfolded code is a crucial part of our deobfuscation slicing algorithm, which is used to determine instructions irrelevant to the real behavior of program. This turns out to be conceptually simple and also reflects the way in which the `eval()` construct is handled in the underlying implementation.

Another modification we made on standard CFG construction algorithm is the processing of callback functions. For the function call instruction labeled as native call, if the instruction follows it in the trace is not adjacent (determined by instruction’s address), then all the instructions between the native call instruction and its
adjacent instruction are generated by (one or more execution of) callback function. In this sub-trace, the boundary of each execution is identified and it is then used for the CFG construction of corresponding callback function. Also, as a preliminary step for applying deobfuscation-slicing (discussed in Section 3.2.3), the sub-trace of callback execution is cut off from the trace and associated with the call instruction to the native function which invokes the callback.

3.2.3 Semantics-Based Deobfuscation

Given the execution trace and control flow graph, the next step is to identify the instructions that are semantically relevant to the program’s externally observable behavior. For this, we use a variation on a program analysis technique known as dynamic slicing.

In general, dynamic slicing is the problem of identifying, for a given execution of a program $P$, which instructions (or statements) in $P$ actually affect the value of a given variable at a given point in $P$ according to both data and control dependencies. The issue is somewhat more complicated in stack-based interpreters because the instructions that use the expression stack typically do not have their operands represented explicitly. Dynamic slicing in such situations has been investigated by Wang et al. [57] in the context of slicing Java bytecode traces. We adapt their algorithm in two ways, both having to do with the dynamic features of JavaScript used extensively for obfuscation. The first is that while Wang and Roychoudhury use a static control flow graph, we use the dynamic control flow graph discussed in Section 3.2.2. The reason for this is that in our case a static control flow graph does not adequately capture the execution behavior of exactly those dynamic code unfolding constructs, such as `eval()` and `document.write()`, that we need to handle when dealing with obfuscated JavaScript. The second is in the treatment of dynamic constructs during slicing, such as code-unfolding and the bracket notation. Consider a statement `eval(s)`: in the context of deobfuscation, we have to determine the behavior of the code obtained from the string $s$; the actual construction of the string $s$, however, is simply part of the obfuscation process and is not directly relevant.
for the purpose of understanding the functionality of the program. When slicing, therefore, we do not follow dependencies through any code unfolding statements.

A different approach is used to handle object property accesses using bracket notation. This access mechanism is useful in situations where different executions of the same piece of bracket-notation code access different object properties, e.g., when iterating over a list of properties in a loop. In such situations, the code used to construct the various strings used for the bracket-notation access is relevant to understanding the behavior of the program, and should be included in the slice. On the other hand, if every instance of the bracket-notation code accesses the same property, then this access mechanism provides no additional benefit compared to the more common dot-notation access; arguably, it makes the code a little harder to understand (especially if the string being used for the bracket-notation access is constructed dynamically), and so is obfuscatory. For this reason, given an instruction instance $I$ in the trace that uses a bracket-notation access, we check whether all the instances of $I$ in the trace access the same property name $s$: if so, the dependency from $I$ to the code that constructs $s$ is not followed; instead, the constructed name $s$ is used in decompilation stage. Otherwise, if different instances of $I$ use different property names $s$, the dependency from $I$ to the code that constructs $s$ is treated normally and the string construction code is included.

The key to any dynamic slicing based technique is to accurately capture the data and control flow of the target program, which is especially challenging for slicing JavaScript code in web pages: DOM enables the interaction between JavaScript code and the host HTML document, but the document related information is opaque to the JavaScript interpreter. For example, a program can store data in a HTML element and retrieve it later, similar to the usage of variables (e.g., see Figure 2.1). Furthermore, JavaScript program can also perform code unfolding at runtime using DOM methods (e.g. `document.write()`), either directly or from an external source. Therefore, to precisely slice JavaScript program, it is important to keep track of the connection between the bytecode used by interpreter and DOM operations. To this end, our slicing algorithm considers extra information provided by instrumented
DOM, as discussed in Section 3.2.1. More specifically, the reference to the document element is printed at the point where it is accessed by DOM methods, which recovers the data dependence hidden in the native functions.

In addition, we treat `document.write()` specially. `document.write()` method writes a string of text to the HTML document, which could be used both for basic document manipulation and code unfolding. Those two cases are handled differently, depending on whether new code is introduced and executed. If JavaScript code is dynamically generated and executed by calling this method, then our slicing algorithm handles the call in exactly the same way as `eval()`: cut the dependence between `document.write()` and the code generated. Otherwise, we consider it a regular native function call which is used for output purpose.

The handling of callback functions requires some extra processing. Assuming there is a call to a native function $f$ which takes a callback $c$ as an argument, in order for our slicing algorithm to follow the dependency accurately, we need to explicitly make connections between the return value of $f$ and the return value from each execution of $c$, as well as the arguments passed to $f$ and parameters of $c$. More specifically, if the call to $f$ is included in the slice because of data dependency on $f$’s return value, then return instructions from all the execution of $c$ should be included in the slice; if an instruction included in slice is dependent on any parameter of $c$, then the call instruction invokes $f$ should also be included in slice.

We refer to this algorithm as deobfuscation-slicing. The pseudocode is shown in Algorithm 1. Lines 1 – 5 are initialization. The algorithm traverses the execution trace backwards, processing each instruction in order from the last instruction to the first. Lines 8 – 9 extracts from the trace the set of memory locations on the stack that are read and/or written by the instruction, and similarly for properties and DOM elements. Lines 10 – 12 cut the dependency for property access using bracket notation, as discussed above. If we encounter a `return` instruction, this instruction must be in a callee function, and since the trace is being traversed backwards we
**Input:** A dynamic trace $T$; an instruction instance $instr \in T$; a dynamic control flow graph $G$;

**Output:** A program slice $S$;

$S := \emptyset$;
currFrame := lastFrame := NULL;
LiveSet := $\emptyset$;
stack := a new empty stack;
$I :=$ instruction instance at the last position in T;

while true do

inSlice := false;

Uses := locations of data used by $I$;
Defs := locations of data defined by $I$;

if $I$ is property access by bracket notation $\land$ all instances of the corresponding instruction access the same name then

| Uses := Uses - location of the string argument; |
end

inSlice := $I$ is instr;

if $I$ is a return instruction then

| push a new frame on stack; |
else if $I$ is an interpreted function call then

| lastFrame := pop(stack); |
else

| lastFrame = NULL; |
end

currFrame := top frame on stack;

if $I$ is an interpreted function call $\land$ $I$ is not eval $\land$ $I$ is not code-unfolding document.write then

| inSlice := inSlice $\lor$ lastFrame is not empty; |
else if $I$ is a control transfer instruction then

| for each instruction $J$ in currFrame s.t. $J$ is control-dependent on $I$ do |

| inSlice := true; |
| remove $J$ from currFrame; |
end

inSlice := inSlice $\lor$ (LiveSet $\cap$ Defs $\neq \emptyset$);
LiveSet := LiveSet $\cup$ Defs;

if inSlice then

| add $I$ into $S$; |
| add $I$ into currFrame; |
| LiveSet := LiveSet $\cup$ Uses; |
end

if $I$ is not the first instruction instance in $T$ then

| $I :=$ previous instruction instance in T; |
else

| break; |
end
end

**Algorithm 1:** Deobfuscation-Slicing algorithm
push a new frame on the stack (line 14); analogously, when we encounter a call to an interpreted function (native functions are not traced), we pop the stack because the call instruction is in the caller (line 16). The underlying implementation handles dynamic code generation via `eval()` and `document.write()` like a function call; line 22 of our algorithm ignores code unfolding, as discussed above. To keep the description of deobfuscation-slicing algorithm clear and concise, the detail of callback processing is omitted from pseudocode.

**Algorithm 2**: Semantics-based deobfuscation algorithm.

The deobfuscation-slicing algorithm only solves half of the puzzle, we still have to determine which instructions affect program’s behavior, i.e. on which instructions to apply deobfuscation-slicing. Our approach of semantics-based deobfuscation consists of two basic steps, both steps rely on the deobfuscation-slicing algorithm. The pseudocode is shown in Algorithm 2:

1. **identify all instructions relevant to code unfolding (line 3-8).** First, the algorithm traverses the execution trace in order from the last instruction instance to the first, for each instance of dynamic code unfolding instructions in trace
(e.g. `eval()` and call to `document.write()`), the deobfuscation-slicing algorithm is applied on it to identify instruction instances relevant to code unfolding, which include native function calls contribute to dynamic code generation. After this step, set $U$ contains all the instructions in trace $T$ that are relevant to code unfolding.

2. **identify all instructions relevant to observable behavior (line 9-14).** The algorithm traverses the trace backwards, applying the deobfuscation-slicing algorithm on each call to the native function which is irrelevant to code unfolding (those not in set $U$ as identified in the first step). The resulting set $R$ contains instructions semantically relevant to the observable behavior of the script.

3.2.4 **Decompilation**

The slicing step described above identifies instructions in the dynamic trace that directly or indirectly affect arguments to native function calls, which includes functions that invoke system calls. Instead of recomputing a control flow graph considering only those relevant instructions, we adopt a simpler approach for decompilation: transform the original dynamic control flow graph to the higher-level representation such as an abstract syntax tree (AST), and label those AST nodes constructed from relevant instructions. This way, we avoid the complexity of handling potential problems caused by slicing, for example, basic blocks might be scattered and the branching target instruction might not in the slice.

A program in the bytecode representation of SpiderMonkey can not be directly converted into valid JavaScript source code, due to the existence of those low level branch instructions, e.g. `ifne`, `goto`, etc. Therefore, as the first step, we use `goto` statement to represent those branch operations in AST. Since the CFG has already been processed using loop analysis and function identification, we need to construct an abstract syntax tree for each function. The basic blocks of the CFG are traversed in depth first order on the corresponding dominance tree, `goto` node is created in two cases: at the end of basic block that doesn’t end with a branch instruction, or
whenever a branch instruction is encountered. In addition to storing information of target block in goto nodes, we also keep track of a list of preceding goto nodes in each target node. Once every basic block has been translated to an AST node, loop structures are constructed by creating infinite while loop node which, initially, contains only the nodes of corresponding natural loop obtained from Section 3.2.2. Once we have an AST with goto nodes, additional code transformation can be applied to generate valid JavaScript source code (Section 3.2.5). Basic block node and loop node in the AST will be referred as block node.

3.2.5 Code Transformation

Introducing goto statements during decompilation allows us to apply a straightforward algorithm to construct the AST, but JavaScript source code generated directly from this AST is invalid, because goto statement is not supported in the core language of JavaScript. To recover valid code, we need to transform the extended AST to eliminate goto statements, without changing the logic of the program.

Joelsson proposed a goto removal algorithm for decompilation of Java bytecode with irreducible CFGs [21]. This algorithm traverses the AST over and over and applies a set of transformations whenever possible. We adapt this algorithm to handle JavaScript and the instruction set used by the SpiderMonkey JavaScript engine [33]. The basic idea is to transform the program so that each goto is either replaced by some other construct (such as break and continue), or the goto and its target are brought closer together in a semantics-preserving transformation so it can be deleted. The transformation stops when none of the transformations can be applied to the AST. The resulting syntax tree is traversed one last time, for each node labeled by the decompiler described in Section 3.2.4, corresponding source code has been printed out. The detailed description of the code transformation algorithm and the rules applied can be found in Appendix A.
3.3 Experimental Evaluation

We evaluated the efficacy of our approach using a prototype implementation based on Mozilla’s open source Firefox web browser, which uses SpiderMonkey as its JavaScript engine. We tested this prototype on three synthetic programs as well as an actual JavaScript malware sample obtained from the Internet. First, we used two versions of the familiar Fibonacci program: this was chosen, first because it contains a variety of language constructs, including conditionals, recursive function calls, and arithmetic operations; and second because it is small and familiar, which makes it easy to assess the quality of deobfuscation. Our third synthetic test case is a very simple program that obfuscated so as to distribute its logic over multiple code contexts. This poses a problem for most existing JavaScript deobfuscators, which assume that the entirety of the logic is contained in one of the unfolded JavaScript code contexts. Finally, we tested our prototype using a sample of actual web-based malware obtained from the Internet by retrieving the contents of a URL extracted from a spam email.
Figure 3.4 shows two versions of Fibonacci number computation programs. The first one, $P_1$, is shown in Figure 3.4(a), this program is hand-obfuscated to incorporate multiple nested levels of dynamic code generation using `eval` for each level of recursion. The second program, $P_2$, as shown in Figure 3.4(b), is also hand-obfuscated, in which we added dependency between real workload and the value used by `eval` (local variable $x$ in function `fib`). Three versions of each of these programs are used—the program as-is as well as two obfuscated versions—one using an obfuscator we wrote ourselves that uses many of the obfuscation techniques described in Section 2.1, including DOM operation; and an online obfuscator [2]. Figures 3.5 shows the fragments of obfuscated programs corresponding to $P_1$; the obfuscated code for $P_2$ are very similar and not shown separately due to space constraints.

The output of our deobfuscator for these programs is shown in Figure 3.6. Figure 3.6(a) shows the deobfuscated code for all three versions of $P_1$ (the original code, shown in Figure 3.4(a), as well as the two obfuscated versions shown in Figure 3.5). Figure 3.6(b) shows the deobfuscated code for all three versions of $P_2$. For each of $P_1$ and $P_2$, the deobfuscator outputs are the same for all of the three versions. It can be seen that the recovered code is very close to the original, and expresses the
function f (arg0) {
    var local_var0, local_var1, local_var2;
    local_var0 = arg0;
    local_var1 = arg0;
    if(arg0<2)
        local_var2 = 1;
    else {
        local_var1 = local_var1-2;
        local_var1 = f(local_var1);
        local_var0 = f(local_var0-1);
        local_var2 =
            local_var0+local_var1;
    }
    return local_var2;
}

var x,y;
x = 3;
y = f(x);
alert(y);

function fib (arg0) {
    var local_var0, local_var1;
    local_var1=1;
    if(arg0<2)
        local_var0=local_var1;
    else
        local_var0=
            fib(arg0-1)+fib(arg0-2);
    return local_var0;
}

var y;
y=fib(3);
alert(y);

Figure 3.6: Deobfuscator outputs for programs $P_1$ and $P_2$
same functionality. The results obtained show that the technique we have described is effective in simplifying away obfuscation code and extracting the underlying logic of obfuscated JavaScript code. This holds even when the code is heavily obfuscated with multiple different kinds of obfuscations, including runtime decryption of strings and multiple levels of dynamic code generation and execution using `eval()` and `document.write()`, in particular, from simplified code of $P_2$ (Figure 3.6(b)), we could see that our approach handles those code intended to be "hidden" by `eval` correctly.

All obfuscations shown in Figure 3.4 and 3.5 are typical techniques widely used in the wild, they also satisfy the assumption made by current deobfuscators: the unobfuscated, complete logic is revealed in one of the unfolded JavaScript code contexts, i.e. if the deobfuscator simply examines every string passed to code unfolding operations such as `eval()` and `document.write()`, the unobfuscated logic can be directly identified in one of them. Our next test program, $P_3$, is purposely constructed to violate this assumption. For illustrative purpose, we make the original logic of $P_3$ very simple, which consists of only three statements:

\[
b=0; \quad ++b; \quad \text{alert}(b);
\]

This code is manually obfuscated by hiding each statement into obfuscated variants of $P_1$ and $P_2$, in four steps. First, we remove the calls to native function `alert()` in $P_1$ and $P_2$ in Figure 3.4. Next, the modified $P_1$ and $P_2$ are obfuscated using the online obfuscator [2]. Then we insert first two statements of $P_3$ into these two obfuscated programs, as if a part of the obfuscation process, and concatenate them together. Last, we apply one more level of obfuscation to the code from last step, and attach the third statement of $P_3$ to its end. Figure 3.7 shows the unfolded code contexts of obfuscated $P_3$ as described above, the original code of $P_3$ is highlighted and some smaller code contexts are omitted. Figure 3.7(a) is the topmost level obfuscation, statement `alert(b)` of $P_3$ resides in this context. Figure 3.7(b) is the context unfolded by the `eval()` in Figure 3.7(a). This context consists of obfuscated Fibonacci number programs, and the first two statements of $P_3$ are hidden in these two obfuscation processes consecutively. Figure 3.7(c) and (d) present the logic of
var cl=[167,184,163, ......,191,107,107]; var ii=0; var str=";";
for(ii=0;ii<cl.length;ii++)str+=String.fromCharCode(cl[ii]-66);
ii=0; eval(str); alert(b); 

(a) 

eval(function(p,a,c,k,e,d){e=function(c){return c.toString(36)}; b="0;
if(l".replace(/\^/,String)){while(c--){d[c.toString(a)]=k[c]]
c.toString(a)\k=[function(e){return d[e]}];e=function()
{return\w+};c=1;while(c--){if(k[c]){p=p.replace(new RegExp("\\b'\+e(c)
+\"b',g'),k[c])}}return p;'h f(6){0 8=6;0 4=6;0 7;0 9="5'v7=8'v4;":0 c="8=f(8-1);5(9)";0 a="4=f(4);5(c);":0 b="s(6=2){7=1}"g(4=4;2-5(a))";5(b);d 7;0
i=f(3)';19,19,.var||\tt2=eval[n[k[t1|s4|s2|s1|s3|return|if[else]function
y'.split(''),0,])]);eval(function(p,a,c,k,e,d){e=function(c){return c.toString(36)};if(l".replace(/\^/,String)){while(c--){d[c.toString(a)]=k[c]]
c.toString(a)\k=[function(e){return d[e]}];e=function() return\w+;};c=1;e b(i)0 5;0 3=1;0 6="b'";0 a=",";0 9="i-";0
d="3";f(i<2)7("5";"7"(c"+(3*2),8));g 7("5";"+6+9+3.8();+a++;"+6+9+(3*2).8() +a);h 5)0 =b(4);';20,20,.var||\tx||\ttf1=eval|toString|\tt|f2|fib|s2|function|if|else|
return\tt'z'.split(''),0,));

(b) 

function f(n){var t1=n;var t2=n;var k;var s4="eval(k=t1+t2;)";var s3="t1=f(t1-1);eval(s4);";var s2="t2=f(t2);eval(s3);";var s1="if(n<2){k=1}else(t2=t2-2;eval(s2))";eval(s1);return k}var y=f(3); 

(c) 

function fib(i){var k;var x=1;var f1="fib();var f2=";var s1="i-";var s2="x";
if(i<2) eval("k="+eval("s"+(x*2),toString()));else eval("k="+f1+s1+x.toString()+f2++;"+f1+s1+(x*2),toString()+f2);return k}var z=fib(4); 

(d) 

Figure 3.7: Unfolded code contexts from obfuscated version of programs $P_3$
function f0 (a0, a1, a2, a3, a4, a5) {
    b=0;
}
function f4 (a0, a1, a2, a3, a4, a5) {
    ++b;
}
f0(0,0,0,0,0,0)
f4(0,0,0,0,0,0)
alert(b);

Figure 3.8: Deobfuscator outputs for programs $P_3$

Fibonacci number computation, from $P_1$ and $P_2$, both unfolded by context of Figure 3.7(b). Some of the smaller code contexts generated are not shown here.

Although $P_3$ is extremely simple, identifying its original logic from unfolded contexts in Figure 3.7 is still challenging: the original code is scattered among different code contexts at different obfuscation levels, hidden in garbage code; and it is easy to misidentify $P_3$ as Fibonacci computation. Therefore, as we can see, deobfuscators adopt the simple “context-unfolding” technique is very ineffective against the obfuscation which does not satisfy its assumption. In comparison, Figure 3.8 presents the output of our deobfuscator, in which most of the obfuscation and garbage code are removed, recovered code is very close to the original $P_3$, and expresses the same functionality. The extra code (function f0 and f4) is introduced because of the control dependency, which is possible to be simplified away by further analysis, e.g. in this case, since none of the arguments is relevant, the invocation of the functions can be simply replaced by their body.

Finally, we evaluated our prototype system using a JavaScript malware sample, $P_4$, which we collected from the Internet as described earlier. Figure 3.9 shows the
high-level dynamic structure of $P_4$. Context 1 is the initial obfuscated JavaScript which resides in the web page opened by web browser; it is a small piece of code (see Figure 3.10) that invokes `document.write()` method to dynamically insert a hidden iFrame, and cause the load of an external web page. This newly loaded web page contains more obfuscated code, which consists context 2 (see Figure 3.11). Similarly, context 2 causes one more level of code unfolding using `eval()` and generates context 3 (see Figure 3.12). Context 3 is the intended logic, which tries to detect whether the user has the Adobe reader plugin in her browser and loads the actual payload — a PDF file created to exploit a vulnerability in the Adobe PDF reader. This action is also conducted using a dynamically created hidden iFrame.

Figure 3.13 shows the output of our deobfuscator. The recovered code is very close to the code in Figure 3.12, which is obtained by manual deobfuscation. We can see that the result captures the essence of the malicious behavior of $P_4$ and eliminates all the obfuscation in context 1 and context 2. The code in the `else` branch of the `if-else` statement and function `pdf2()` is missing because it is not executed therefore not existed in the trace.

Thus far we have focused our efforts on implementing functionality in our deobfuscation tool instead of performance. Nevertheless, its current performance seems acceptable. For all the test programs described earlier, the average overhead of
Figure 3.11: The second code context of $P_4$
Figure 3.12: The third code context of $P_4$

trace collection is $2.5\mu s$ per instruction executed. Table 3.1 presents the performance for the deobfuscation process: with traces ranging from 582 instructions to 25,330 instructions, our tool takes an average of about $76\mu s$ per trace instruction. In particular, the trace for our malware sample was 12,225 instructions long and required 1.126s to analyze, which works out to about $92\mu s$/instruction.
function pdf(){
    if(navigator.plugins){
        local_var0 = 0;
        while (local_var0 < navigator.plugins.length) {
            local_var1 = navigator.plugins[local_var0].name;
            if(local_var1.indexOf("Adobe Reader") != -1)
                document.write("<iframe src=’./f3256c.pdf’ width=’1’ height=’1’ frameborder=0></iframe>");
            local_var0++;
        }
    }
}
pdf();

Figure 3.13: Deobfuscator output for malware sample $P_4$

<table>
<thead>
<tr>
<th>Test program</th>
<th>Length of trace ($instructions$)</th>
<th>Total time ($\mu s$)</th>
<th>Avg. time ($\mu s/instr.$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$ obf1</td>
<td>6166</td>
<td>221949</td>
<td>35.9</td>
</tr>
<tr>
<td>$P_1$ obf2</td>
<td>582</td>
<td>40514</td>
<td>69.6</td>
</tr>
<tr>
<td>$P_2$ obf1</td>
<td>5755</td>
<td>155537</td>
<td>27.0</td>
</tr>
<tr>
<td>$P_2$ obf2</td>
<td>587</td>
<td>50209</td>
<td>85.5</td>
</tr>
<tr>
<td>$P_3$</td>
<td>25330</td>
<td>3793117</td>
<td>149.7</td>
</tr>
<tr>
<td>$P_4$</td>
<td>12225</td>
<td>1125874</td>
<td>92.1</td>
</tr>
</tbody>
</table>

Table 3.1: Running time of the deobfuscator for test programs
3.4 Attack Models

Intuitively, there are two ways by which an attacker might attempt to evade our approach to deobfuscation. The first is to hide relevant instructions, by adding fake dependency between them and strings to be unfolded. Our approach is immune to this technique, because an unfolded string $s$ depends on some code $v$ doesn’t automatically exclude $v$ from the resulting slice; if the real workload depends on $v$, then $v$ would be added to slice regardless of the connection with code unfolding operation. In other words, only code which is solely used for obfuscation would be eliminated.

The second evasion technique is to disguise the obfuscation code as relevant by adding extra irrelevant native function calls and creating dependencies between the obfuscation code and those irrelevant calls. Our semantics-based approach can not automatically simplify away this kind of disguised obfuscation because, in general, the additional native function calls potentially change the observable behavior of the program. One approach to mitigating such attacks is to select, either manually or automatically, a (possibly proper) subset of the native function calls in the program that are used as the basis for the slicing process described above. This leads to a key issue of our semantics-based deobfuscation: the model of observable behavior of JavaScript programs.
CHAPTER 4

WEAKNESSES IN EXISTING DEFENSES

Section 2.2 discussed the state of the art approaches for detecting web-based malware, which range from dynamic monitoring of memory objects for shellcode-like strings to machine-learning techniques, both static and dynamic, to identify code features associated with malware. These techniques can be complemented using multi-path exploration techniques to increase code coverage. While the experiments show some promising results, however, past experience suggests that the evolution of obfuscation and cloaking techniques is very often driven by the desire to evade current defenses: attackers analyze existing detection techniques, identify implicit underlying assumptions, and devise techniques that violate these assumptions and thereby cause the defenses to fail. This chapter presents work published in DIMVA 2013 (see Lu et al. [28]) that studies the limitations of existing analysis and detection systems for web-based malware and proposes a combination of obfuscation and anti-analysis techniques targeting those limitations. The main contributions of the work presented in this chapter are as follows:

1. We systematically analyze existing web-based malware detection approaches, and study their limitations.

2. By targeting at those limitations, we describe a possible combination of obfuscation techniques capable of bypassing existing detectors.

3. We evaluate our ideas using real JavaScript malware. Our experiments indicate that current detection technology fails to detect malicious code obfuscated using our approach.

4. We describe the strengths and limitations of the proposed obfuscation and discuss approaches that could potentially be used to counteract our techniques.
We also provide insights into the possibility and cost of adjusting existing detection tools to the new techniques.

The remainder of this chapter is organized as follows. Section 4.1 discusses the limitations of existing approaches. Section 4.2 describes proposed obfuscation and anti-analysis techniques in details. Section 4.3 presents the experimental evaluation. And Section 4.4 provides a detailed discussion on possible countermeasures to defeat the proposed obfuscation.

4.1 Limitations of Existing Approaches

The works discussed in Section 2.2 focus on detecting today’s malware; a natural question to ask is: what are the weaknesses of current detection techniques, what sorts of cloaking techniques might malware use to exploit those weaknesses, and what might tomorrow’s malware look like? This section explores this question by analysing existing detection techniques for JavaScript malware to examine their assumptions and study how these assumptions can be broken.

Static approaches. Simple signature-based techniques are widely used by commercial anti-malware software, in part because of their low runtime overhead. However, the obfuscation techniques described in Section 2.1, combined with server-side polymorphism, render such techniques of limited utility. For example, Curtsinger et al. report false negative rates for traditional anti-virus software that are as high as 83% [14].

To address this shortcoming, a number of researchers have proposed the use of machine learning techniques to identify code that “looks weird”. The general approach is to train a classifier on a corpus of current malware, then use it to identify new malware. One problem with this approach is that, because the training process uses current malware using current obfuscation techniques, that is what the classifier learns to recognize: new obfuscation techniques, where the code looks does not resemble current obfuscated malware, can be problematic to handle. Furthermore, due to the existence of code unfolding, even the static approach needs to execute
the code to reveal hidden code contexts, which makes it suffer from limited code coverage.

**Dynamic approaches.** These kinds of approaches typically rely on executing malware in a controlled environment and monitoring their behaviors. Different techniques usually focus on different aspects of execution, such as memory objects, suspicious function invocations and sequence of actions. In contrast to static approaches, dynamic-based detectors are not hindered by code obfuscations, but they have their own limitations.

For approaches rely on monitoring the code execution to identify malicious behaviors, it usually suffers from limited code coverage of a single execution. In addition, runtime monitoring typically introduces significant execution overhead. While the overhead might not be a serious problem for applying runtime monitoring on JavaScript programs written for simple document manipulation, the web has already evolved beyond just a collection of documents and became a platform for full-fledged application software (thanks to technologies such as HTML 5), such performance degradation is unacceptable for those complex JavaScript programs with intensive user-interaction (e.g. games) which are now common on the Internet.

For online detectors based on sandboxing, deciding when to transfer control back and let the browser execute target script is another issue. Current approaches usually run target code in the sandbox until finish or a predetermined timeout is reached, whichever comes first. Again, for complex JavaScript applications discussed above, this approach is no longer feasible. Since the target program might run indefinitely and/or require complicate user intervention to proceed, the execution might not have a clear finish point and the running time is not suitable to be the indication of progress anymore such that the timeout method might terminate execution prematurely, i.e. before malicious behavior presents.

**Multi-Path Exploration.** Trigger-based behaviors have been employed extensively in web-based malware, usually to hide exploits in environments not vulnerable to the attack. Previous study shows, among collected malware, 89.5% of them will branch on environment-dependent conditionals [24], such as `ActiveXObject`. While
dynamic analysis makes it possible to examine any code that may be created as the program executes, it has the drawback that it shows only a single execution path through the program. This means that if the malicious code happens to lie on a different path through the program, it may go undetected. To get around this problem, researchers have proposed various multi-path exploration techniques to increase code coverage [8, 30, 24]. Some recent proposals are lightweight enough to be practical for online analysis on a large scale [24]. These approaches typically focus on conditional branches in the code; the problem of identifying program inputs that would cause alternative execution paths to be explored becomes considerably more difficult if the environmental trigger is not implemented by conditional branch constructs provided by the programming language.

Summary. As the discussion above suggests, obfuscations aimed at evading existing detectors should satisfy three properties. First, the obfuscated code should look, at least syntactically, like ordinary unobfuscated JavaScript code. Second, the malware should avoid exposing its malicious behaviors if its execution is being monitored. Finally, to thwart multi-path exploration, it should avoid using conditional jumps to implement the control flow logic that activates the malicious code if no execution monitoring is detected. One way to accomplish these goals is using a code obfuscation technique called emulation-based obfuscation [47, 51] together with anti-analysis techniques we called anti_monitoring defense and implicit conditionals. We discuss each of these in more detail below.

4.2 Thwarting Analysis

This section considers how the limitations of existing detection techniques for web-based malware can be exploited to allow malicious code to evade detection. As the discussion in the previous section suggests, obfuscations aimed at evading existing detectors should satisfy three properties. First, the obfuscated code should look, at least syntactically, like ordinary unobfuscated JavaScript code. Second, the malware should avoid exposing its malicious behaviors if its execution is being monitored.
Finally, to thwart multi-path exploration, it should avoid using conditional jumps to implement the control flow logic that activates the malicious code if no execution monitoring is detected.

One way to accomplish these goals is using a code obfuscation technique called emulation-based obfuscation together with anti-analysis defenses and a technique we call *implicit conditionals*. These techniques are not specific to JavaScript, and emulation and anti-analysis techniques have been encountered in native-code malware. However, what makes them especially relevant to web-delivered malware is a combination of circumstances. First, the routine use of browsers, together with the proliferation of resource-limited devices such as smartphones, means that malware detection has to be cheap, lightweight, and online (i.e., has to occur as web pages or documents are opened for viewing). This requirement, combined with the increased code complexity resulting from technologies such as HTML5, limits the computational effort detectors can devote to code analysis. The remainder of this section explores how this observation can be exploited by constructing obfuscations that allow for a high degree of code diversity and require significant computational effort to penetrate, thereby rendering them likely to be able to escape detection.

### 4.2.1 Emulation-Based Obfuscation

Emulation-based obfuscation transforms the original JavaScript program $P$ into a pair $(B_P, I_P)$, where $B_P$ is a bytecode representation of $P$ and $I_P$ is an interpreter written in JavaScript whose sole purpose is to execute the program $B_P$. For example, one of many possible choices for $(B_P, I_P)$ is stack-based bytecode pairs with a dispatch routine: a loop repeatedly fetches an instruction opcode, decodes it, and dispatches execution to the code fragment that handles the operation specified. While we do not know of existing JavaScript malware using this approach to obfuscation, the idea itself is not new to security researchers and similar techniques have already been adopted by native malware writers.

From an attacker’s perspective, emulation-based obfuscation offers the advantage that the payload logic is not exposed: examining the executed code only reveals the
structure and logic of the bytecode interpreter; the underlying logic of the program being executed is encoded in the form of bytecode as data. Moreover, details of the bytecode encoding and corresponding interpreter can be perturbed randomly, which means successful reverse engineering of one obfuscated program may not give us much help for analyzing programs obfuscated by the same obfuscator.

In addition to significantly increasing the difficulty of analysis and reverse engineering, adopting new obfuscation techniques like emulation in web-based malware also provides a big advantage against current machine learning based detectors. Emulation is fundamentally different to obfuscation widely adopted by existing malicious JavaScript, such as code unfolding and string obfuscation. For example, existing malicious JavaScript often repeatedly applies string operations to encrypted texts/values to dynamically create new code, which usually is a very long string. It is then executed by code unfolding functions like \texttt{eval()} and \texttt{document.write()}. But none of the above is necessary for emulation-based obfuscation, in other words, by examining the code contexts of emulation-obfuscated program, we will not identify characteristics typical in existing obfuscated code. Therefore, emulation-obfuscated code looks syntactically similar to ordinary unobfuscated JavaScript code.

The code snippet in Figure 4.1 shows an example of the emulation-based obfuscation; the bytecode sequence in this case is for the iterative Fibonacci number calculation program. The obfuscated script consists of three parts: the bytecode program (variable \texttt{bytecode}) written in an arbitrarily chosen stack-based instruction set, functions implement those instructions and the interpreter using a dispatch loop to execute the bytecode. For illustrative purpose, we don’t make any attempt to disguise the structure of the interpreter and use instruction mnemonics directly in the bytecode program, the detailed discussion on countermeasures based on identifying patterns of interpreter is presented in Section 4.4. As we can see from this simple example, the original program’s logic is hidden in the bytecode program; only the logic of the interpreter is exposed. More importantly, this emulation-obfuscated script doesn’t possess any obfuscation patterns from existing malicious JavaScript.

In addition to concealing the logic of the malicious code, emulation can also
//bytecode handlers
function op_int(){
    stack[sp++]=sc[pc++];
    return 0;
}
function op_bindname(){
    stack[sp++]=sc[pc++];
    return 0;
}
...130 lines omitted...
function op_setname(){
    var assign_temp = stack[sp-1];
    globals[stack[sp-2]] = assign_temp;
    stack[sp-2] = assign_temp;
    sp--;
    return 0;
}
function op_nameinc(){
    var nameinc_temp = sc[pc++];
    stack[sp++] = globals[nameinc_temp];
    globals[nameinc_temp]++;
    return 0;
}

/* Byte-code representation of the Fibonacci number program. */
var bytecode = [op_bindname,0,op_name,1,op_int,4,op_call,1,op_setname,op_pop,
    op_stop,op_reservestack,4,op_int,0,op_setvar,0,op_pop,op_int,1,op_setvar,1,
    op_pop,op_getarg,0,op_int,0,op_eq,op_ifeq,33,op_getvar,0,op_return,op_int,1,
    op_setvar,2,op_pop,op_getvar,2,op_getarg,0,op_lt,op_ifeq,68,op_getvar,0,
    op_getvar,1,op_add,op_setvar,3,op_pop,op_getvar,1,op_setvar,0,op_pop,op_getvar,
    3,op_setvar,1,op_pop,op_varinc,2,op_pop,op_goto,38,op_getvar,1,op_return];
var globals={0:0,1:11};

//initialization of the interpreter
stack=new Array(); pc=0,sp=0,bp=0;
stack[sp++]=-1;stack[sp++]=-1;bp=sp;
sc = bytecode;

//interpretation begins here
while(pc>=0 & pc<sc.length)
    if(sc[pc++]()==-1) break;

Figure 4.1: Example of an emulation-obfuscated program
be used to hide other components of the program, such as shellcode strings, that detectors often look for. This can be done by applying existing string obfuscation techniques to the shellcode strings, but instead of implementing the string decoding routine in JavaScript directly (which itself is suspicious and can be identified by existing detectors), transforming the decoding logic into bytecode as well. This makes it possible to conceal both the shellcode strings and the decoder from a static examination of the program.

4.2.2 Anti-Analysis Defense

Anti-analysis defenses, which are also encountered in native malware, involve detecting runtime monitoring/tracing system; if the program determines that its execution is being monitored, it can then alter its execution to avoid revealing any malicious behavior.

Ideally, a detection system should be indistinguishable from the intended target, at least from the attacker’s perspective. This very often does not hold true in practice, however, because dynamic analyses are typically performed within sandboxed environments, which are susceptible to detection. One reason is that complete behavior emulation of web browser, including DOM, ActiveX controls and various plug-ins, can be quite difficult. Also, sandboxed detectors incur significant execution overhead. Our experiments indicate, for example, that sandboxed execution monitoring systems for JavaScript are 1–2 orders of magnitude slower than modern browsers. This is illustrated in Figure 4.2, which shows the normalized results of experiments timing the execution of snippets of JavaScript code in different environments. It can be seen that sandboxed execution monitoring systems for JavaScript are about $46 \times$ to $100 \times$ slower than modern browsers such as Chrome 20, Firefox 14, Internet Explorer 9; and $3.5 \times$ to $7.4 \times$ slower than older JavaScript engines such as Internet Explorer 8 and SpiderMonkey.¹

¹For this experiment, Internet Explorer 6 and 7 were run on Windows XP with Service Pack 3 while Internet Explorer 8 and 9, Firefox 14 and Chrome 20 were run on Windows 7 64-bit version. Two different SpiderMonkey JavaScript engines were used, one unmodified while the other (labeled with “mod”) was instrumented to log each bytecode instruction executed in order to simulate the
This dramatic difference in overheads between monitored and non-monitored execution environments suggests that measurements of execution speed may be used to detect runtime execution monitoring. We note, however, that timing tests to detect monitoring are not infallible, and sometimes there may not be a clear line between fast monitors and slow clients. This means that anti-analysis defenses evade detection at the possible cost of reduced exploitation success rate. On the other hand, overhead variation due to different browsers is usually not a problem, since each exploit typically targets vulnerability in a web-browser of specific version and/or brand.

4.2.3 Implicit Conditionals

It may be possible to bypass the anti-analysis defenses described in the previous section by combining dynamic analyses with multi-path exploration techniques [8, 30, 24]. Existing multi-path exploration techniques focus on conditional branches in the code: whereas a normal program execution will take one branch of a conditional branch effects of runtime monitoring, and both versions were compiled and installed on Ubuntu 11.04 with kernel 2.6.38-12. Wepawet [59] is an online service based on JSAND [12].
branch, multi-path exploration involves exploring all possible branches. From an
analysis perspective, conditional branches have the advantage that straightforward
code inspection allows us to determine, for any given conditional branch, the branch
target where execution continues depending on whether the branch is taken or not.

By obfuscating a JavaScript program using emulation, conditional statements in
original program are transformed into bytecode sequences, which is equivalent to
converting conditionals to indirect jumps. This might make analysis more challeng-
ing, but it does not change the fact that the execution path is dependent on certain
conditions, i.e. conditional statement is required to implement bytecode handlers
after all, therefore existing multi-path exploration techniques can still be applied
with some modification.

We can make multi-path exploration more difficult by replacing conditional
branches with calculation of parameters used by the interpreter (discussed in Sec-
tion 4.2.1) in a way that makes the selection of execution paths transparent. We
refer to this approach as implicit conditionals. The intuitive idea here is that given
an explicit conditional $C \equiv \text{if } e \text{ then } C_0$, we replace $C$ by a code fragment $C'$
that has the following properties:

1. $C'$ does not contain an explicit test on $e$.

2. If $e$ holds, the effect of executing $C'$ is identical to that of executing $C_0$;
   otherwise, executing $C'$ has no or meaningless effect.

The main component of $C'$ is the emulated version of $C_0$, which is set up in such
a way that the parameters used by the interpreter (i.e. instruction-pointer, entry-
point, etc.) have the correct values if and only if $e$ holds. The calculation of
parameters can be done in various different ways using a function $f_e$ that satisfies
the following properties: $(i)$ $f_e$ computes some appropriate desired value if and only
if the condition $e$ holds; and $(ii)$ the computation of $f_e$ does not involve conditionals.
We list below some ways of using such conditional-free functions.
**Entry point generation.** The idea here is that the initial value of the interpreter’s instruction pointer, i.e., the offset in the byte-code array where the execution of the byte code program begins, is determined by a conditional-free function $f_e$ that takes as input an environment profile (i.e., a collection of values describing the program’s execution environment) and returns the correct value only if the condition $e$ holds. This can be done in many different ways; here we present an example based on the anti-analysis defense discussed in Section 4.2.2. In this case, the environment profile $p$ is the time required to execute some given fragment of code. Suppose that we have determined that the $p$ should be less than 100 (ms) in target browser, and the bytecode offset of the entry point for the malicious code is $entry_m = 20$, then $f_e$ might be implemented as:

$$f_e(p) = \left\lfloor \frac{p + 1}{100} \right\rfloor \times 20$$

In this case, $f_e(p) \equiv entry_m$ (i.e. 20) if and only if $p \in [0, 99]$, which ensures the attack runs normally; for $p \geq 100$, $f_e(p) \geq 40$, and the execution ends up with unpredictable behavior.

However, unpredictable behavior may not be guaranteed to be non-suspicious. For example, even if the value returned by $f_e$ is not the correct value $entry_m$, it may nevertheless expose some components of an attack, e.g., a heap spray or construction of a shellcode string, that can cause the attack to be recognized, or the program might crash, which itself can be considered suspicious. One way to deal with this using a more elaborate computation for the function $f_e$ such that, if the condition $e$ does not hold, returns a value that is out of bounds in the bytecode array. Or a better approach is to construct bytecode sequence deliberately, such that, while only the correct value leads to malicious behavior, all the other entry-point values calculated from possible inputs are corresponding to valid and harmless bytecode execution without crash.

Figure 4.3 shows an example of applying entry point generation for implicit conditional. Detailed discussion of Figure 4.3 will be presented in Section 4.2.4.
Instruction pointer increment generation. In this case, the amount by which the interpreter’s instruction pointer is incremented after each instruction is determined by a conditional-free function that returns the correct value only if \( e \) holds. Typically, the instructions of (non-branching) bytecode are laid out contiguously in memory and the instruction pointer is incremented by the size of a single instruction each time an instruction is executed. Such contiguous layout is not essential, however: for example, each real instruction can be separated by one or more “chaff instructions” such that proper execution requires that the instruction pointer be incremented by some multiple of the size of a single instruction. The value of this increment can then be set using an implicit conditional, similarly to the entry point generation described above. More generally, the amount by which the instruction pointer is incremented after each instruction need not be a constant: for example, it can be a sequence of pseudo-random numbers, each in some range \([\text{min},\text{max}]\): all we need is a predictable sequence of values such that bytecode instructions can be placed at the correct offsets. The function \( f_e \) can then be used to set the seed for the pseudo-random sequence to the right value if and only if \( e \) holds.

As shown above, by using conditional-free functions and emulation-based obfuscation, we can hide environment dependent conditionals and trigger-based behaviors in the logic of interpretation, thus thwart multi-path exploration. In addition, even though entry point and instruction pointer increment generation are introduced as two independent implementations of emulated conditionals, it is possible to combine both technique.
4.2.4 Implementation

Figure 4.3 shows the general structure of a program combining all the proposed techniques discussed in this chapter, namely, emulated-obfuscation, anti-analysis defense and implicit conditionals. As we can see from the high-level structure shown in Figure 4.3, anti-analysis defense, alone with other environmental fingerprinting code are located at the beginning of the program. Their result – environment profile $p$ is then passed to the implicit conditional. In this example, implicit conditional is implemented by entry point generation alone as discussed in Section 4.2.3, and the instruction pointer increment is 1. Furthermore, the conditional-free function $f_e(p)$ is designed to return 20 if and only if $p$ shows the intended condition holds and returns $20 \times i$ where $i \geq 2$ and $i \in \text{integer}$ otherwise. $f_e(p)$ is then used to set the entry point of the bytecode program. Finally, the bytecode is arranged such that bytecode instructions $bytecode[20], bytecode[21], \ldots, bytecode[38], bytecode[39]$ (corresponding to dark slots in the array), when executed in this order, will lead to malicious behavior, execution starts with other possible entry points (e.g. 40, 60, 80, \ldots) would cause the emulation to behave harmlessly (one simple way to implement this is to assign bytecode nop-slide in light-colored slots).

As the proof-of-concept implementation, we have applied all proposed anti-analysis techniques on existing malware and benign programs by manual transformation. For example, all the samples discussed in Section 4.3 are implemented by hand using an arbitrarily chosen, stack-based instruction set; and the anti-analysis defense and implicit conditionals are both implemented in their basic forms (i.e. single loop for anti-analysis defense, and simple instruction-pointer and entry-point generation as shown in Section 4.2.3).

4.3 Experimental Evaluation

To evaluate if the proposed techniques are effective against existing detectors, we selected 7 real malware samples, named $M_1$ to $M_7$, including 6 scripts in HTML pages and one in a PDF file (see Table 4.1, where OSVDB ID is the identification
Table 4.1: Description of malware samples

<table>
<thead>
<tr>
<th>Sample</th>
<th>File Type</th>
<th>CVE Number</th>
<th>OSVDB ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>HTML</td>
<td>-</td>
<td>64839</td>
</tr>
<tr>
<td>$M_2$</td>
<td>HTML</td>
<td>CVE-2006-3730</td>
<td>27110</td>
</tr>
<tr>
<td>$M_3$</td>
<td>HTML</td>
<td>-</td>
<td>80662</td>
</tr>
<tr>
<td>$M_4$</td>
<td>HTML</td>
<td>CVE-2007-3071</td>
<td>38803</td>
</tr>
<tr>
<td>$M_5$</td>
<td>HTML</td>
<td>CVE-2007-3703</td>
<td>37707</td>
</tr>
<tr>
<td>$M_6$</td>
<td>HTML</td>
<td>-</td>
<td>61964</td>
</tr>
<tr>
<td>$M_7$</td>
<td>PDF</td>
<td>CVE-2008-2992</td>
<td>49520</td>
</tr>
</tbody>
</table>

All the samples use heap-spray for payload delivery. Next we created two sets of obfuscated programs from these. Programs in the first set had two different obfuscators applied to them, each of them using existing techniques such as string obfuscation and code unfolding. Those in the second set were obfuscated using the techniques proposed here as described at the end of the previous section. It should be noted that applying proposed obfuscation doesn’t affect the reliability of malware, which was tested by running obfuscated exploit in browser with targeted plugins installed.

We used three malware detectors, covering a wide spectrum of detection technologies, for our experiments: VirusTotal [56] is an online portal to a collection of anti-virus software with up-to-date exploit databases that exemplifies current commercial malware detection technology; Zozzle [14] is a machine learning based static detector (we used the same trained classifier as evaluated in [14] for our experiment); and Wepawet [59], a hybrid detection system based on JSAND [12], that represents a state-of-the-art combination of static and dynamic analyses. We believe these three detectors, range from traditional signature matching to state-of-the-art static and dynamic analyses, represent the current state of detection techniques. Therefore it allows us to have a comprehensive evaluation on the effectiveness of proposed obfuscation techniques against different approaches.

Table 4.2 shows the detection rates for these three malware detectors. There is no result for neither version of $M_7$ from Zozzle, since Zozzle is designed for detecting
## Table 4.2: Detection Results

HTML based malware only. It can be seen that, while the malware samples obfuscated by existing techniques were identified as malicious with 100% detection rates by both VirusTotal and Wepawet, and with 33% detection rate by Zozzle, another group of malware samples, protected by new obfuscation techniques, were able to bypass all the targeting detectors, with the exception of \( M_7 \), which was detected by two of the anti-virus software on VirusTotal. It turns out, however, that this has nothing to do with any malicious content: the only reason the PDF file \( M_7 \) is identified as malicious is that it contains JavaScript code. We confirmed this with a PDF file containing just a Fibonacci number program written in JavaScript, which is identified as malicious by the same two anti-virus software with identical exploit names.

One concern about emulation-based obfuscation is the runtime overhead. The interpretation process introduces extra operations, for example, the access of interpreter’s stack, function invocations, etc. Although existing JavaScript malware are not computation-intensive, this decreased performance can still be a potential problem for applying emulation-based obfuscation to JavaScript. To better understand this issue, we conducted another experiment to compare the overhead between code-

<table>
<thead>
<tr>
<th>Malware Sample</th>
<th>Existing Obfuscation</th>
<th>New Obfuscation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VirusTotal</td>
<td>Wepawet</td>
<td>Zozzle</td>
</tr>
<tr>
<td>( M_1 )</td>
<td>5 / 40</td>
<td>√</td>
</tr>
<tr>
<td>( M_2 )</td>
<td>4 / 41</td>
<td>√</td>
</tr>
<tr>
<td>( M_3 )</td>
<td>5 / 42</td>
<td>√</td>
</tr>
<tr>
<td>( M_4 )</td>
<td>5 / 42</td>
<td>√</td>
</tr>
<tr>
<td>( M_5 )</td>
<td>5 / 42</td>
<td>√</td>
</tr>
<tr>
<td>( M_6 )</td>
<td>5 / 42</td>
<td>√</td>
</tr>
<tr>
<td>( M_7 )</td>
<td>10 / 42</td>
<td>√</td>
</tr>
</tbody>
</table>

√: detected  
×: undetected

For fractions present in columns “VirusTotal”, the denominator is the number of anti-virus software available on VirusTotal, and the numerator is the number of anti-virus software that identify corresponding sample as malicious.
Figure 4.4: Comparison of running time between code-unfolding and emulation unfolding based obfuscation and emulation on several different browsers, result is shown in Figure 4.4.

We used the same testcases as discussed above, but with anti-analysis defense removed from emulated samples (since it is not limited to emulated code). We can see that the running times for the two different obfuscation techniques are comparable, even for older browsers. The samples obfuscated using emulation are slightly slower than code-unfolding based samples, but the differences are not very large. This suggests that such obfuscation techniques could realistically be deployed using current technology.

4.4 Countermeasures

Although our experimental results show existing approaches are not very effective against proposed obfuscation techniques, it is worth paying attention to the feasibility and cost of adjusting existing tools to handle those new techniques.

It has been shown that emulated programs are difficult to reverse-engineer: an examination of the executed code reveals only the structure and logic of the bytecode interpreter. Current approaches to dealing with emulation-based obfuscations of
native-code malware make strong assumptions about the structure of the interpreter [42, 46]. Therefore, for those machine learning based techniques that focus on code structure and patterns of operations, retraining classification models or adding new features within their current analysis framework are not likely to succeed, since those techniques do not attempt to penetrate logic of interpretation. However, it is possible that other approaches may be used to attack this approach to obfuscation, as discussed below.

**Interpreter Detection.** Because of the difficulty of retrieving real logic from emulated programs, an attacker could attempt to identify the presence of an interpreter via structure analysis or pattern matching on the control flow graph of the code, and mark all programs present such pattern as malicious without further analysis. However, there are some problems must be addressed by such interpreter identification approach.

First, this approach assumes that all programs obfuscated by emulation are malicious and interpreter structure only exists in emulated code. However, both of the assumptions are false. As discussed in Section 2.1, obfuscation provides a simple way for web programmers to protect their code from potential plagiarism, this is especially true for JavaScript programs because it is distributed in the form of source code. Also, there is legitimate usage of interpreter in many scenarios other than obfuscation. In [52], Terrace et al. proposed an approach that runs untrusted third-party script inside a JavaScript interpreter (which itself is written in JavaScript), the interpreter is used as a secure sandbox thus gives programmer complete control over third-party script. As another example, Bellard [7] implemented an interpreter-based PC emulator completely in JavaScript which is able to run a compiled Linux kernel as well as various binary tools in a web browser. As we can see, the presence of interpreter in JavaScript not only cannot be used to identify malicious content, it is not even an indication for the usage of emulation-based obfuscation, serious collateral damage might be caused to legitimate programs if it was adopted for malware detection. Some might argue that obfuscation is mainly applied to mal-
ware, and suggest a whitelist approach in addition to interpreter identification for JavaScript malware detection. Assuming their statement on obfuscation is true, while this approach might work for now, we believe as web becomes one of the main platforms for application software, interpreter or similar structure will be a common component among various emerging applications, therefore the whitelist approach will soon become impractical.

Secondly, identifying interpreter structure is not as trivial as it might appear to be. First of all, some of the key patterns in a straightforward interpreter implementation (e.g. Figure 4.1) can be eliminated. For example, function `setTimeout()` (or alternatives such as [6]) can be used at the end of each bytecode handler to execute following bytecode, thus get rid of the dispatching-loop. Furthermore, such analyses can be thwarted by code embedding transformations [15], where the code for a program \( P \) is disguised by intermixing it with code for other programs. The basic transformations are shown in Figure 4.5: \( G \) is (the control flow graph for) some fragment of code, which can be introduced either somewhere within a basic block.

\[
\begin{align*}
B = \begin{cases}
  l_1 \\
l_2 \\
\vdots \\
l_k \\
l_{k+1} \\
l_n
\end{cases} & \Rightarrow \quad \begin{cases}
  l_1 \\
l_2 \\
\vdots \\
l_k \\
l_{k+1} \\
l_n
\end{cases}
\end{align*}
\]

\( (a) \) Vertex splitting

\[
\begin{align*}
v_1 \xrightarrow{e} v_2 & \Rightarrow \quad \begin{cases}
  v_1 \\
e_1 \\
v_2
\end{cases}
\end{align*}
\]

\( (b) \) Edge splitting

Figure 4.5: Basic code embedding transformations
(Figure 4.5(a), “vertex splitting”), or along a control flow edge (Figure 4.5(b), “edge splitting”). As long as $G$ satisfies some non-interference and termination properties (which can be guaranteed by construction), the functionality of $P$ is unaffected by this transformation. By iterating this transformation, we can introduce enough code into enough different points of $P$ that its the instruction sequence and control flow structure of the resulting composite program is radically different from that of $P$. So far, we don’t have concrete evidence to support this claim, as we are still trying to evaluate this code embedding transformation approach quantitatively, therefore we leave detailed evaluation for future work.

**Countering Anti-Analysis Defenses.** The intuition behind the anti-analysis defense is that we can detect the presence of runtime analysis by its execution overhead. Therefore, one straightforward way to attack anti-analysis defense would be improving the performance of the detector to the level comparable to real browsers. However, this approach is limited by two factors. First, newer browsers usually adopt JIT compilation and frequently executed bytecode (such as anti-analysis defense) is translated into machine code for direct execution on the hardware, but many dynamic detectors rely on the information collected from the JavaScript interpretation therefore are not capable to monitor the code execution at machine code level. Second, based on the result shown in Section 4.2.2, even though it is not unreasonable to believe the overhead of runtime analysis can be reduced, thus makes it hard for anti-analysis defense code to reliably determine whether it is being monitored or just executed in an outdated web browser or a very slow machine, the trade off is less information from execution, which will inevitably lead to reduced detection accuracy.

Since it appears to be challenging to reduce the runtime overhead of analysis, an attacker might try to eliminate the performance gap completely by incorporating detection into the browser of regular user. In other words, the detection will be conducted on the fly, instead of in an emulated sandbox. This way, if the malicious content is executed, it will expose itself to the detector, otherwise, the exploit
failed. However, this approach would result in deteriorated execution performance of JavaScript programs (in contrast to delayed loading caused by sandboxing), which is usually unacceptable for complex applications. Also, many of the existing sandboxing-based detection techniques cannot be adopted directly, since they need the complete execution of the malware, which is infeasible in a real browser. Lastly, this approach is not suitable for offline detectors, since the malicious behavior would be concealed by anti-analysis defense thus evade detection.

As opposite to proposed methods above, browsers can try to slow down script execution at the beginning to prevent being attacked by such potential approaches. This method should be capable to defeat anti-analysis defense implemented in our malware samples, it also has a serious flaw however. As described in Section 4.2.2, anti-analysis defense has to be placed before any malicious behavior is exposed, but it is not necessarily means the code needs to be at the beginning of the script. To counter this slow down attack, an approach similar to code embedding transformations discussed above can be applied. For example, malware (including anti-analysis defense) can be embedded in a JavaScript game, and the execution will only be directed to this malicious part after user start playing the game. Obviously, slow down the entire execution is not a realistic option.

**Brute-Force Attack on Implicit Conditionals.** Attackers may apply brute-force approach to explore all possible emulated execution paths, thus defeating implicit conditionals. However, there are several limitations on the brute-force attack. First, attackers have to identify the logic of interpreter before exhaustive search. More specifically, initialization and usage of entry point, instruction pointer increment values have to be identified, along with other structures. This problem itself is not trivial, which usually involves manual analysis and assumptions on interpreter structure. Second, assuming automatic identification of interpretation logic can be achieved, the overhead of brute-force attack might be prohibitive for online and large-scale detection.
Importantly, the attacks described above are all fairly heavy-weight. This suggests that deploying them for routine use in browsers, in the form of online analysis algorithms, may be too expensive to be practical.
CHAPTER 5

DETECTING ENVIRONMENT-DEPENDENT MALWARE

As discussed in Section 2.1, web-based malware very often use runtime code unfolding to obfuscate their code. Due to the difficulty of statically analyzing such obfuscations, most detectors resort to dynamic analysis of the code in a sandboxed environment. While this makes it effective against existing malware with dynamically generated code, it also suffer from various shortcomings that make it possible for malware to evade them, as shown in Chapter 4. In particular, one such anti-analysis defense is to make the malware environment-dependent, such that the malicious behavior is exposed only when the malware is executed in the right environment. Malware utilize this technique is called environment-dependent malware.

One way to get around this problem is to use multi-path exploration techniques to increase code coverage. However, such approaches generally involve sophisticated and heavyweight techniques that incur significant performance overheads. While useful for offline forensic analysis, their runtime cost renders them of limited utility for routine use in online detectors (i.e., where detection is carried out when a web page is loaded for viewing).

This Chapter presents work published in MALWARE 2013 (see Lu et al. [26]) that takes a different approach to client-side defense against environment-dependent malware. In contrast to the attempts on increasing code coverage in detectors, our technique, called environmental predicates, 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).
The remainder of this chapter is organized as follows. Section 5.1 describes the details of environmental predicates and their construction. Section 5.2 presents the experimental evaluation. And Section 5.3 discusses the possible attack models.

5.1 Environmental Predicates

As discussed above, performance issue is an important factor in usability considerations for client-side malware detection, which is why anti-malware software usually resort to light-weight techniques with higher false negative rate instead of more heavy-weight techniques such as multi-path exploration. While achieving higher detection rate is important for client-side detectors, we believe it might not be in the end users’ best interest, especially with the cost of degenerated browser performance, since there is a subtle difference between detecting malware and avoiding exploitation. Instead of worrying about the detection rate of their detector, end users are usually more concerned about whether their system will actually be compromised. In other words, it is acceptable that a faster detector misidentified a piece of web-based malware as benign, as long as it doesn’t behave maliciously on that specific machine.

Based on the observation above, we propose environmental predicates, a different approach to client-side defense against environment-dependent malware. We assume a system architecture where malware detection is performed via dynamic analysis in a sandboxed environment external to the browser, which then communicates the results of its analysis back to the browser. Instead of using expensive and heavyweight techniques to increase malware detection rates, environmental predicates extend and augment existing sandbox based detection 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\(^1\): if the check is satisfied, the web page can be viewed safely.

The idea behind environmental predicates is that if a sandboxed detector de-

\(^1\)Obviously if a document is found to be malicious in the course of the sandboxed analysis, the question of then opening it outside the sandbox does not arise.
termines that an execution of the code along some execution path $p$ is benign, and we can efficiently determine that the in-browser execution path will not deviate from $p$, then we can conclude that the in-browser execution will also be benign. The lightweight nature of the in-browser check means that it can be repeated each time the document is viewed—so that if, at some point, the execution environment changes in a way that would cause the code to take a different execution path, this can be detected and the document reanalyzed. The cost of analyzing a web page can thus be amortized across multiple views of that page, and reanalysis is not necessary as long as environmental conditions do not change in a way that could cause the execution path of the code to change.

In this dissertation, our study focuses on environment-dependent data that could be used to identify the browser, a specific browser version, installed plugins, or even operating system. More specifically, all fields of `navigator` object are identified as environment-dependent data. Although `ActiveXObject` is not identified as an environment-dependent data source since it does not exist in our Firefox based sandbox, our technique can still handle branches dependent on `ActiveXObject` (alternatively, it can be implemented to return a dummy object as environment-dependent data)

5.1.1 Definition

Given a JavaScript program $P$ that takes a (non-malicious) execution path $\pi$ during sandboxed analysis, an environmental predicate $f$ with respect to a set $S$ of environmental inputs for the pair $(P, \pi)$ 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 $\pi$; 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 \textit{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. The decision vector is obtained from the sandbox execution log. The code in \( P \) that affects environment-dependent branches is extracted from \( P \) using a program transformation technique called slicing. Checkpoints are then added to the slices during the process of environmental predicate construction. Figure 5.2 shows an example of an environmental predicate.

Therefore, based on the definition above, an environmental predicate must have following two properties to be safe and effective:

**Property 1.** If an environmental predicate \( f \) generated from the execution of a JavaScript program \( P \) in environment \( E \), where no malicious behavior is exposed, then \( f \) must be benign regardless of whether or not \( P \) is malicious.

**Property 2.** An environmental predicate \( f \) returns Boolean value \textit{true} when executed in the environment \( E \), if and only if the execution of original JavaScript program \( P \) would have made identical environment-dependent decisions in \( E \) to its execution in \( E_b \), from which \( f \) is created.

Property 1 above follows from our assumption that the malware detector deployed in the sandbox is able to detect exposed malicious behavior during an execution. Based on this assumption, if a program \( P \) is identified by the detector as benign, then none of the code of \( P \) that executed in detector sandbox can be malicious. Therefore, the environmental predicate \( f \) for that particular execution of \( P \) must be benign since it is constructed exclusively from the execution trace. This assumption is not unreasonable based on the evaluation results from existing web-based malware detection techniques. Property 2 follows by structural induction on the abstract syntax trees of the original program \( P \) and the environmental predicate \( f \). We can show that environmental predicates constructed using our algorithm, which is
discussed in Section 5.1.3, satisfy this requirement for the environment-dependent data we focus on.

5.1.2 Construction of Environmental Predicate

The construction of environmental predicates is based on a JavaScript analysis framework described in Chapter 3 and consists of the following components, as shown in Figure 5.1:

1. [Trace Collection] 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. \textit{[Environmental Predicate Generation]} Finally, generate environmental predicate based on labeled (relevant) syntax tree nodes.

In this Chapter, we focus on code trimming and environmental predicate generation. Detailed discussion on other components can be found in Chapter 3.

\textbf{Code Trimming.} 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 [24]. 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 a program analysis technique known as dynamic slicing [3] 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.

There are two kinds of slicing algorithms discussed in the literature: \textit{forward slicing}, which identifies the code that is influenced by the value of a variable or expression at a given program point; and \textit{backward slicing}, which identifies the code that influences the value of a variable or expression at a given program point. 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 $T$;

1. $S := \emptyset$;
2. $U := \emptyset$;
3. for $i := 1$ to length of $T$ do
   4.   $instr := i$-th instruction instance in $T$;
   5.   if $instr$ is the source of environment-dependent data then
   6.     $U := U \cup \text{Forward-Slicing}(T, instr, G)$;
   7.   end
4. end
5. for $i := \text{length of } T$ to 1 do
6.   $instr := i$-th instruction instance in $T$;
7.   if $instr$ is a decision-making branch instruction $\land instr \in U \land instr \notin S$ then
8.     $S := S \cup \text{Backward-Slicing}(T, instr, G)$;
9.   end
10. end

**Algorithm 3:** Environment-dependent decisions identification algorithm.

The pseudocode of our algorithm for identifying environment-dependent branches is shown in Algorithm 3, 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.

In addition, all decisions made by the environment-dependent branches have to be logged in execution order. This can simply be done by traversing the trace after set $U$ is constructed in Algorithm 3. For every branch in $U$, the actual decision is retrieved from the trace and stored in a vector for further use. In our proof-of-concept system, we use integers 1 and 0 for branch taken and not-taken respectively, as well as whether the instruction threw an exception. In addition, we use the name of target function for function invocation.

**Environmental Predicate Generation.** 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 5.1 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. 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 5.2 shows an example of actual environmental predicates for try-catch branch.
Figure 5.2: An example of environmental predicate

Function environmentalPredicate:
```javascript
function environmentalPredicate() {
    vecIndex = 0;
    DecisionVector = [1];
    try{
        try{
            x = new ActiveXObject("ShockwaveFlash.ShockwaveFlash");
        } catch(e) {

        }
    } catch(e) {
        if(e == "wrong path")
            throw "wrong path";
        if(decisionVector[vecIndex++] != 1)
            throw "wrong path";
    }
    } catch(e) { return false; }
    return true;
}
```

(a) Original program

(b) Environmental predicate

mechanism, which is generated from the execution of the original program in which new statement threw an exception.

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.
<table>
<thead>
<tr>
<th>Original Program</th>
<th>After Checkpoint insertion</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( e ){</td>
<td>if ( e ){</td>
</tr>
<tr>
<td>} else {</td>
<td>if (decisionVector[vecIndex] +] != 0 )</td>
</tr>
<tr>
<td>}</td>
<td>throw &quot;wrong path&quot;;</td>
</tr>
<tr>
<td></td>
<td>S'</td>
</tr>
<tr>
<td>S1</td>
<td>} else {</td>
</tr>
<tr>
<td></td>
<td>if (decisionVector[vecIndex] +] != 1 )</td>
</tr>
<tr>
<td></td>
<td>throw &quot;wrong path&quot;;</td>
</tr>
<tr>
<td>S2</td>
<td>}</td>
</tr>
</tbody>
</table>

|                       | while ( e ){                |
|                       |     if (decisionVector[vecIndex] +] != 0 ) |
|                       |     throw "wrong path";     |
|                       |     S'                       |
|                       | }                          |
| while ( e ){          | if (decisionVector[vecIndex] +] != 1 ) |
| } S1                  |     throw "wrong path";     |

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

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

|                       | try{                        |
|                       |     S1                      |
|                       | } catch ( e ){              |
|                       |     S2                      |
|                       | }                           |

| /* S1 threw an unhandled exception during execution */ | S1 throw "wrong path"; |
5.1.3 Analysis of Environmental Predicate

In this section, we present an analysis to verify the correctness of our environmental predicate based approach on detecting environment-dependent control-flow discrepancy in the execution of a JavaScript program (compares to the sandboxed execution), which resulted from if-else, loop, try-catch and switch constructs as well as function invocation (above five mechanisms are called “branch” for short). The analysis is based a collection of environment-dependent data source, including all fields of navigator object and ActiveXObject object. And we assume the non-environment-dependent inputs to original program and corresponding environmental predicate among multiple executions, regardless of the hosting environment, are identical.

Lemma 1: If there is one environment-dependent branch executed during sandboxed execution $X_{sandbox}$ of program $P_{origin}$, then environmental-predicate $P_{pred}$, generated from $X_{sandbox}$, can determine at runtime whether the decisions made by the corresponding branch in $P_{pred}$ matches the decisions made by $P_{origin}$ during $X_{sandbox}$.

This claim can be justified by the structure of environmental predicate. According to our checkpoint insertion algorithm, for a specific branch under consideration, a checkpoint is inserted at every possible branch target. One and only one of the checkpoints will be executed right after each time the branch is executed and check whether the branch decision matches the decision logged from the sandboxed execution. At this point, only the checkpoint on the path matching logged decision in decision vector will allow execution to continue without throwing a “wrong path” exception. This observation is consistent among all five branch mechanisms considered in this dissertation. In other words, any control-flow discrepancy will cause a “wrong path” exception to be thrown at runtime, which will then be caught by outermost catch-clause of the environmental predicate.

On the basis of Lemma 1, we first show that Property 1 of the environmental predicate from Section 5.1.1 holds for untrimmed environmental predicate.
Lemma 2: Given the program $P_{\text{origin}}$ and the untrimmed environmental predicate $P_{\text{untrim}}$ constructed from an execution of $P_{\text{origin}}$ in environment $E_{\text{sandbox}}$ (we call this execution $X_{\text{sandbox}}$ and the executed sub-program of $P_{\text{origin}}$ as $P_{\text{exe}}$). If during the execution of $P_{\text{untrim}}$, every environment-dependent branch executed in $P_{\text{untrim}}$ makes identical decision to its corresponding branch in $P_{\text{exe}}$ during $X_{\text{sandbox}}$, then $P_{\text{untrim}}$ returns true, otherwise it returns false.

We proof this claim by induction on the number of statements executed during $X_{\text{sandbox}}$. As discussed in Section 5.1.2, untrimmed environmental predicate is constructed by keeping all executed statements of $P_{\text{exe}}$ and insert checkpoints at every branch (thus all the branch decisions are recorded). Therefore, for every statement $S_i$ executed in $P_{\text{exe}}$, there is a corresponding statement $S'_i$ in $P_{\text{untrim}}$, such that $S_i$ and $S'_i$ are semantically equivalent (they might not be syntactically identical because of the decompilation).

The basis of the induction is when $P_{\text{exe}}$ has only one executed statement $S_1$, then there are 2 possible cases:

- $S_1$ is not a branch: then the program locations after the execution of $S_1$ and $S'_1$ are always the same, there is no control flow discrepancy, and $P_{\text{untrim}}$ always returns true.

- $S_1$ is a branch: in this case, we need to further differentiate two scenarios. First scenario is when $S_1$ is not environment-dependent. Since execution environment is the only possible difference between the execution of $P_{\text{untrim}}$ and $X_{\text{sandbox}}$, the branch condition of $S_1$ and $S'_1$ evaluate to same value, which means they always make identical decisions. In other words, the program locations after the execution of $S_1$ and $S'_1$ are equivalent, therefore $P_{\text{untrim}}$ always returns true. Second scenario is when $S_1$ is environment-dependent. Then values used for evaluation of branch conditions of $S_1$ and $S'_1$ might be different. Based on the Lemma 1, if it causes control flow discrepancy at $S'_1$, then the assertion of checkpoint at $S'_1$ can detect it and $P_{\text{untrim}}$ returns false. Otherwise, if $S_1$ and $S'_1$ still make the identical decisions despite the difference
in values used for evaluation of branch conditions, then there is no control flow discrepancy at $S'_1$. In this case, the program locations after the execution of $S_1$ and $S'_1$ are equivalent, therefore $P_{\text{untrim}}$ returns true.

The basis shows that for $P_{\text{exe}}$ with only one executed statement, the claim is true. Now assume as induction hypothesis that this claim holds when $P_{\text{exe}}$ has $n$ ($n \geq 1$) statements. To proof our claim, we need to show it is also true if $P_{\text{exe}}$ has $n + 1$ ($n \geq 1$) statements.

$P_{\text{exe}}$ has $n + 1$ ($n \geq 1$) statements is equivalent to appending a statement $S_{n+1}$ to $P'_{\text{exe}}$ with $n$ statements. According to our hypothesis, any control flow discrepancy caused by environment dependent branch from first $n$ statements of $P_{\text{untrim}}$ can be detected and $P_{\text{untrim}}$ returns false. If the control reaches $S'_{n+1}$ in $P_{\text{untrim}}$, then there was no control flow discrepancy from $S'_1$ to $S'_n$. In addition, the program locations after the execution of $S_n$ and $S'_n$ are equivalent. At this point, the return value of $P_{\text{untrim}}$ completely relies on the execution of $S'_{n+1}$, such that $P_{\text{untrim}}$ returns false if $S'_{n+1}$ is an environment-dependent branch and made a decision different from $S_{n+1}$ in $X_{\text{sandbox}}$, $P_{\text{untrim}}$ returns true otherwise, as discussed in basis. Therefore, for $P_{\text{exe}}$ with $n + 1$ statements, during the execution of $P_{\text{untrim}}$, if every environment-dependent branch executed in $P_{\text{untrim}}$ makes identical decision to its corresponding branch in $P_{\text{exe}}$ during $X_{\text{sandbox}}$, then $P_{\text{untrim}}$ returns true, otherwise it returns false.

Since both the basis and the inductive step have been proved, it has now been proved by induction that statement holds true for programs with $n$ executed branches, where $n \geq 1$. ■

**Lemma 3:** Given the program $P_{\text{origin}}$ and the untrimmed environmental predicate $P_{\text{untrim}}$ constructed from an execution of $P_{\text{origin}}$ in environment $E_{\text{sandbox}}$ (we call this execution $X_{\text{sandbox}}$ and the executed sub-program of $P_{\text{origin}}$ as $P_{\text{exe}}$). If both $P_{\text{origin}}$ and $P_{\text{untrim}}$ are executed in the same environment $E$, then every environment-dependent branch executed in $P_{\text{untrim}}$ makes identical decision to its corresponding branch in $P_{\text{origin}}$, until any discrepancy from $X_{\text{sandbox}}$ is detected by $P_{\text{untrim}}$. 
Similarly, induction on the number of statements executed during $X_{sandbox}$ is applied to prove this claim. We will first prove the claim on $P_{exe}$, then show that $P_{exe}$ and $P_{origin}$ are indeed equivalent for our purpose. The basis of the induction is when $P_{exe}$ has only one executed statement $S_1$, then there are 2 possible cases, similar to the basis in the proof of Lemma 2:

- $S_1$ is not a branch: since both programs are executed in the same environment, the program states and program locations after the execution of $S_1$ and $S'_1$ are always the same.

- $S_1$ is a branch: When $S_1$ is not environment-dependent, $S_1$ and $S'_1$ make identical decision. Therefore, the program states and program locations after the execution of $S_1$ and $S'_1$ are always the same. When $S_1$ is environment-dependent, since $P_{exe}$ and $P_{untrim}$ are executed in the same environment, $S_1$ and $S'_1$ always make same decision. However, it needs to be noted that if the decision made is different from the one in $X_{sandbox}$, then a environment-dependent control flow discrepancy occurred. Based on Lemma 2 above, $P_{untrim}$ would immediately detect it and return false. Otherwise, the program states and program locations after the execution of $S_1$ and $S'_1$ are the same.

The basis shows that for $P_{exe}$ with one executed statement the claim is true. Now assume as induction hypothesis that this claim holds when $P_{exe}$ has $n$ ($n \geq 1$) executed statements. To proof our claim, we need to show it is also true if $P_{exe}$ has $n + 1$ ($n \geq 1$) executed statements.

$P_{exe}$ has $n + 1$ ($n \geq 1$) statements is equivalent to appending a statement $S_{n+1}$ to $P'_{exe}$ with $n$ statements. According to our hypothesis, every environment-dependent branch executed among first $n$ statements in $P_{untrim}$ makes identical decision to its corresponding branch in $P_{exe}$ until a discrepancy occurs. If a discrepancy does occur among first $n$ statements in $P_{untrim}$, then the execution of $P_{untrim}$ would stop at the point of discrepancy. Therefore, in this case, the claim for $P_{exe}$ with $n + 1$ statements is true regardless of whether $S_{n+1}$ and $S'_{n+1}$ make same decision. If there is no discrepancy among first $n$ statements in $P_{untrim}$, then according to the
hypothesis, the program states and program locations of $P_{exe}$ and $P_{untrim}$ after the execution of $S_n$ and $S'_n$ are the same. As described in the basis, $S_{n+1}$ and $S'_{n+1}$ would also make same decision.

Now both the basis and the inductive step have been proved, the proof is done for $P_{exe}$ and $P_{untrim}$. However the original statement in Lemma 3 requires $P_{origin}$ instead of $P_{exe}$. The difference between $P_{exe}$ and $P_{origin}$ is, since $P_{exe}$ is recovered from the execution $X_{sandbox}$ of $P_{origin}$, code in $P_{origin}$ that is not executed in $X_{sandbox}$ due to branches is not in $P_{exe}$. As discussed in the proof above, when executed, non-branch and non-environment-dependent branch statements in $P_{origin}$ (as well as $P_{exe}$) will not cause the execution path to deviate from the path in $X_{sandbox}$. For environment-dependent branches, the code following the branch is relevant only if the execution of $P_{untrim}$ continues, that is, when the decision made by the branch is identical to corresponding decision in $X_{sandbox}$, in which case, the execution path will not deviate from $X_{sandbox}$. Otherwise, if the decision is different from $X_{sandbox}$, which means, in next step, $P_{origin}$ might branch into code in $P_{origin} - P_{exe}$. However, $P_{untrim}$ will catch this discrepancy and stop execution, thus any code follows in $P_{origin}$ is irrelevant. In other words, when comparing decisions between $P_{origin}$ and $P_{untrim}$, the code in $P_{origin} - P_{exe}$ will not be executed until a branch discrepancy occurs. Therefore, to prove our claim, $P_{exe}$ is equivalent to $P_{origin}$. ■

The final step of our analysis is to establish the equivalence between environmental predicate and untrimmed environmental predicate.

**Lemma 4:** Given environmental predicate $P_{pred}$ and untrimmed environmental-predicate $P_{untrim}$, both are constructed based on the same execution of program $P_{origin}$ in environment $E_{sandbox}$ (we call this execution $X_{sandbox}$). When executed in the same environment $E$, every environment-dependent branch executed in $P_{pred}$ makes identical decision to its corresponding branch in $P_{untrim}$.

We prove this claim by contradiction. By definition, $P_{pred}$ is a sub-program of $P_{untrim}$, such that $P_{pred}$ consists of environment-dependent branches and code these branches depend on (both data and control dependency) from $P_{untrim}$. Now
we assume some environment dependent branches in $P_{\text{pred}}$ would make decisions different from corresponding branches in $P_{\text{untrim}}$, when both executed in the same environment. We focus on the first branch executed which shows this discrepancy, because according to the definition of the environmental predicate, at least one of the $P_{\text{pred}}$ and $P_{\text{untrim}}$ would stop execution and returns false after the branch, thus any following comparison would be meaningless. Let’s call the corresponding branches in $P_{\text{pred}}$ and $P_{\text{untrim}}$ as $S_b$ and $S'_b$ respectively. There are two different cases. First is all the statements before $S'_b$ have an one-to-one correspondence with the statements before $S_b$, that is, those two program are identical up to statement $S_b$ and $S'_b$, then given the fact that both programs are fed with identical environment-dependent and non-environment-dependent input, the branch conditions for $S_b$ and $S'_b$ must evaluate to the same value, therefore they make same branch decision, which contradicts our assumption. Second case is when a collection of statements $C$ before $S'_b$ have no corresponding statement in $S_b$. As shown above, if we remove $C$ from $P_{\text{untrim}}$, the $S_b$ and $S'_b$ always make identical decision. Based on the definition of environmental predicate, $C$ is not part of $P_{\text{pred}}$ means $S_b$ doesn’t depend on any code in $C$, therefore the existence of $C$ doesn’t affect the values used by $S'_b$. We conclude that in this case $S_b$ and $S'_b$ make identical decision, again contradicting to the assumption.

On the basis of Lemmas discussed above, the following theorem can be established (Property 2 of the environmental predicate proposed in Section 5.1.1):

**Theorem:** An environmental predicate $f$ returns Boolean value $\text{true}$ when executed in the environment $E$, if and only if the execution of original JavaScript program $P$ would have made identical environment-dependent decisions in $E$ to its execution in $E_b$, from which $f$ is created.

5.1.4 Implementation

While our main contribution 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 5.3 (the controller is omitted from Figure 5.3 for a clearer presentation of
workflow).

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.

Most existing malware detector using dynamic analysis is applicable for working
alongside our environmental predicate based technique. The detection false
negatives, caused by environment-dependent code due to limited code coverage in
sandboxed execution, is acceptable in our system, since the potential harm from
such miss can be detected using an environmental predicate. Moreover, the pro-
gram tracing overhead from environmental predicate generation can be hidden by
taking advantage of the sandbox execution initiated by the detector, i.e. both en-
vironmental predicate generator and detector can collect information from a single
sandboxed execution.

At a high level, the process of checking an incoming web-page evolves in following
steps, as shown in Figure 5.3:
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 in Chapter 4, 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.

5.2 Experimental Evaluation

To evaluate the efficacy of our approach, we implemented a client-side prototype system as described in Section 5.1.4, and test it on real malware. Since the primary
<table>
<thead>
<tr>
<th>Environment-Specific Data Referenced</th>
<th>branch Mechanism</th>
<th>Results (Firefox)</th>
<th>Results (IE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>navigator</td>
<td>if-else</td>
<td>× ×</td>
<td>× ×</td>
</tr>
<tr>
<td>navigator.userAgent</td>
<td>if-else</td>
<td>× ×</td>
<td>√ √</td>
</tr>
<tr>
<td>navigator.javaEnabled</td>
<td>if-else</td>
<td>√ √</td>
<td>√ √</td>
</tr>
<tr>
<td>navigator.language</td>
<td>if-else</td>
<td>× ×</td>
<td>√ √</td>
</tr>
<tr>
<td>navigator.systemLanguage</td>
<td>if-else</td>
<td>× ×</td>
<td>√ √</td>
</tr>
<tr>
<td>navigator.plugins</td>
<td>loop &amp; if-else</td>
<td>√ √</td>
<td>√ √</td>
</tr>
<tr>
<td>navigator.mimeTypes</td>
<td>loop &amp; if-else</td>
<td>√ √</td>
<td>√ √</td>
</tr>
<tr>
<td>ActiveXObject</td>
<td>try-catch &amp; if-else</td>
<td>× ×</td>
<td>√ √</td>
</tr>
<tr>
<td>navigator.appVersion</td>
<td>function</td>
<td>√ √</td>
<td>√ √</td>
</tr>
<tr>
<td>navigator.appName</td>
<td>switch</td>
<td>× ×</td>
<td>√ √</td>
</tr>
<tr>
<td>Date</td>
<td>implicit conditional</td>
<td>√ √</td>
<td>√ √</td>
</tr>
<tr>
<td>ActiveXObject</td>
<td>unhandled exception</td>
<td>× ×</td>
<td>√ √</td>
</tr>
</tbody>
</table>

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

Table 5.2: Summary of testcases and experiment results

focus of this work is on detecting environment-dependent code, we simply assume
the existence of a 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 col-
lected 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 be-
have 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.

For our purposes, it suffices to consider only situations where the detector clas-
sifies 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 5.2 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 5.2 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. We focus on five basic types of mechanisms, including \texttt{if-else}, loop, \texttt{switch}, function invocation and \texttt{try-catch}. (as discussed in Chapter 4, implicit conditional is implemented by a combination of these mechanisms).

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 $\sqrt{}$ in “discrepancy?” columns), all discrepancies have been detected by executing their environmental predicate (with $\sqrt{}$ 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.

5.3 Attack Models

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. For example, JavaScript program execution under analysis of Wepawet [12] 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:

```javascript
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 example, 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. However, it should be noted that finding a threshold to evade environmental predicate without affecting malware’s infection rate is very difficult. This is because the configuration of computer systems in use ranges dramatically, thus the processing speed various as well. For example, if a threshold used by a piece of malware is set to a fairly small value, therefore effective on faster computers, it is very likely using the same threshold would lead the malware mistakenly identify the execution on slower computers as being monitored, thus miss the opportunity of a successful attack. On the other hand, if attacker chooses a large threshold to increase infection rate, the possibility of false negative from our approach decreases.

A similar issue arises with branches that are dependent on variable environment-dependent values, i.e. a race condition. One such example is to check if current time in milliseconds is divisible by 10. Then there is a chance that sandboxed execution and evaluation of environmental predicate have same execution path, but the actual execution in the browser behaves differently. While the kind of anti-analysis checks shown in this example would significantly reduce the infection rate of the malware, however, there might be other possible ways, that we have not thought of yet in this dissertation, to effectively exploit the race condition of environmental predicate in web browser.

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 \[24\] 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, even for the attacks that started at compromised legitimate websites, they are merely used as a starting point which redirect users to the malicious page on attacker’s server eventually. 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.
CHAPTER 6

CONCLUSIONS

The growing importance of the Internet in recent years has been accompanied by a corresponding increase in web-based malware delivery. This is typically done via “drive-by downloads” that exploit vulnerabilities in browsers and other client-side software. Such attacks are often carried out via scripts written in JavaScript, a language commonly used in client-side web applications. Thwarting such attacks requires the ability to detect malicious JavaScript code. However, this is not easy: attackers generally use a variety of evasion techniques, such as code obfuscation and environmental triggers, to create code that is highly obfuscated and inscrutable. While current proposals in the research literature for detecting web-based malware show some promising results, they also encounter some limitations.

There has been some recent work on automated behavioral analyses of obfuscated web-based malware that in many cases has a “deobfuscator” component, as well as standalone deobfuscation tools. However, these deobfuscators all rely on some simple and intuitive assumptions about the obfuscation and the structure of the obfuscated code. It is not difficult to construct obfuscations that violate them and thereby defeat the corresponding deobfuscators. Chapter 3 presents a semantics-based approach for automatic deobfuscation of JavaScript code. We use dynamic analysis and program slicing techniques to simplify away the obfuscation. The code so obtained is observationally equivalent to the original program for the execution path considered, but has obfuscations simplified away, thereby exposing the core logic of the computation performed by the original code. Moreover, this approach does not make any assumptions about the structure of the obfuscation. We evaluate our approach using a prototype implementation and test it against both synthetic programs and real malware code. The results show that our approach can penetrate multiple layers of complex obfuscations, some of which cannot be handled by existing
techniques, and extract the core logic of the underlying computation.

While this success is encouraging, experience with malware over the last two decades indicates that threats don’t go away: rather, they adapt to new platforms, targets, and opportunities, attackers often analyze existing detection techniques, identify implicit underlying assumptions, and devise techniques that violate these assumptions and thereby cause the defenses to fail. Therefore, a natural question to ask is: what are the weaknesses of current detection techniques, what kind of evasion techniques might malware use to exploit those weaknesses, and what might tomorrow’s malware look like? Chapter 4 explores this question by analysing existing detection techniques for JavaScript malware to examine their assumptions and study how these assumptions can be broken. Further, as a proof-of-concept, we present a combination of obfuscation and anti-analysis techniques, targeting various existing detectors. To prove the proposed techniques can become a real threat, we applied them on real web-based malware and then evaluated their effectiveness by submitting the obfuscated malware samples to state-of-the-art detectors. The results show our obfuscation techniques successfully avoid the detection of those malware samples from targeting detectors, which validates our statement that existing detection techniques can be easily circumvented.

Another widely utilized evasion technique for web-based malware is environmental trigger, where the malicious behaviors are triggered only in specific environment. Such malware 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. We propose a technique suitable for client-side defense to such environment-dependent malware in Chapter 5. In contrast to the attempts on increasing detection accuracy, the goal of this technique is to trade accuracy with performance without compromising user protection. This technique is designed to work alongside a lightweight detector, and provides a safe and efficient way to identify behavior discrepancy of JavaScript program in targeted browser and detector sandbox, therefore protecting users from possible detection false negatives caused by environment dependent
code. Experiments using a proof-of-concept implementation show that this technique can effectively detect environment-dependent behavior discrepancy of various forms, including those seen in real malware.
This section presents the rules used by our code transformation algorithm discussed in Section 3.2.5. The goal of the algorithm is to eliminate goto node in the AST. To achieve this goal, the transformation algorithm traverses the AST over and over and applies the following rules whenever possible.

(a) **Move target block after preceding goto.** This transformation moves a block node to the location right after its preceding goto node. To apply this transformation, node \( m \) and its preceding goto node \( n \) must satisfy following conditions:

- \( n \) must be the only preceding goto node of \( m \), and
- \( m \) is not already located after \( n \), and
- \( m \) is neither a function entry nor a loop header node, and
- if \( m \) is being moved into a loop node \( o \) from outside, then there must be a goto node \( p \) immediately follows \( o \), and a goto node \( q \) at the end of \( m \), such that \( m \) and \( n \) have the same target node.

(b) **Convert goto to continue.** To convert a goto node \( n \) to continue,

- \( n \) and its target node \( m \) must resides in the same loop node \( o \), and
- \( m \) must be the loop header of \( o \).

When this transformation applied, \( n \) has to be removed from \( m \)’s preceding goto list.
(c) **Convert goto to break.** To convert a goto node \( n \) to break,

- \( n \) must resides in the a loop node \( o \), and
- there must be a goto node \( p \) immediately follows \( o \), and,
- \( n \) and \( p \) have the same target node.

When this transformation applied, \( n \) has to be removed from its target node’s preceding goto list.

(d) **Combine infinite loop and if-else.** This transformation, if applied, removes an if-else statement and moves the goto out the loop. To combine a while loop \( p \) and if-else node \( q \), following conditions must be satisfied:

- \( p \) is an infinite loop, and
- \( q \) is the first statement in \( p \), and
- the else branch of \( q \) only has a goto node \( n \), and target of \( n \) is not part of loop \( p \).

then, the \( p \) and \( q \) are combined, by substituting loop condition of \( p \) with predicate of \( q \), substituting \( q \) with statements in the if branch of \( q \), and moving \( n \) to the location immediately follows and outside of \( p \). This rule also has a symmetric case, in which if and else is switched.

(e) **Move goto out of if-else.** This transformation could move gotos out of blocks and reduce the number of goto nodes as well. Given a if-else node \( p \), in which the if branch ends with node \( m \) and else branch ends with node \( n \), if both \( m \) and \( n \) are goto nodes and have the same target, then \( m \) is moved to the location right after \( p \), and \( n \) is removed from AST as well as the preceding goto list of its target.
The code transformation algorithm stops when none of the rules above can be applied to the AST. Then the syntax tree is traversed again, for each \texttt{goto} node $n$, we examine its target node $t$, if $t$ is the node immediately following $n$, then $n$ is removed from syntax tree. The resulting syntax tree is traversed one last time, for each node labeled by the decompiler described in Section 3.2.4, corresponding source code has been printed out.
REFERENCES


