FIREFOX ADD-ON FOR METAMORPHIC JAVASCRIPT MALWARE DETECTION

Sravan Kumar Reddy Javaji
San Jose State University

Follow this and additional works at: http://scholarworks.sjsu.edu/etd_projects

Part of the Information Security Commons

Recommended Citation
Javaji, Sravan Kumar Reddy, "FIREFOX ADD-ON FOR METAMORPHIC JAVASCRIPT MALWARE DETECTION" (2015).
Master's Projects. 401.
http://scholarworks.sjsu.edu/etd_projects/401

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.
A Thesis
Presented to
The Faculty of the Department of Computer Science
San Jose State University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Sravan Kumar Reddy Javaji
May 2015
The Designated Thesis Committee Approves the Thesis Titled

FIREFOX ADD-ON FOR METAMORPHIC JAVASCRIPT MALWARE DETECTION

by

Sravan Kumar Reddy Javaji

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2015

Dr. Thomas Austin Department of Computer Science
Dr. Chris Pollett Department of Computer Science
Mr. Fabio Di Troia University of Sannio
ABSTRACT

Firefox Add-on for Metamorphic JavaScript Malware Detection

by Sravan Kumar Reddy Javaji

With the increasing use of the Internet, malicious software has more frequently been designed to take control of users' computers for illicit purposes. Cybercriminals are putting a lot of efforts to make malware difficult to detect. In this study, we demonstrate how the metamorphic JavaScript malware can affect a victim's machine using a malicious or compromised Firefox add-on. Following the same methodology, we develop another add-on with malware static detection technique to detect metamorphic JavaScript malware.
ACKNOWLEDGMENTS

I am very thankful to my advisor Dr. Thomas Austin for his continuous guidance and support throughout this project and believing me. Also, I would like to thank the committee members Dr. Chris Pollett and Mr. Fabio Di Troia for monitoring the progress of the project and their valuable time.
# TABLE OF CONTENTS

## CHAPTER

1. **Introduction** ................................................. 1
   1.1 Problem ................................................. 2
   1.2 Proposed Solution ....................................... 3
   1.3 A Browser Plugin for Detecting Malware ................. 4

2. **Background** .................................................. 5
   2.1 Encrypted Malware ....................................... 6
   2.2 Polymorphic Malware ..................................... 7
   2.3 Metamorphic Malware ..................................... 8
      2.3.1 Register renaming .................................... 8
      2.3.2 Dead code insertion ................................ 10
      2.3.3 Subroutine permutation ............................. 10
      2.3.4 Equivalent code substitution ....................... 11
      2.3.5 Transposition ........................................ 11
      2.3.6 Changing control flow ............................... 12
      2.3.7 Subroutine inlining and outlining ................. 12
   2.4 Transcriptase ............................................. 12
      2.4.1 Permutator .......................................... 15
      2.4.2 Variable/Function-Name randomization ........... 16
      2.4.3 Meta-Language Symbols .............................. 17
      2.4.4 Code Derivation .................................... 18
2.4.5 Variable/Function insertion ........................................ 19
2.5 Rhino ................................................................. 20
  2.5.1 Architecture .................................................... 21
  2.5.2 Modification ..................................................... 24

3 Firefox Add-on Development ............................................ 27
  3.1 Firefox vs Chrome .................................................. 28
  3.2 SDK vs XUL .......................................................... 29
  3.3 Chrome Authority Usage .......................................... 31
  3.4 Content Scripts .................................................... 32

4 Implementation .......................................................... 35
  4.1 Malicious add-on ..................................................... 35
  4.2 Transcriptase detection add-on .................................... 37
    4.2.1 Malware Detection Technique ............................... 38
    4.2.2 Opcode Graph Similarity Technique ......................... 38
    4.2.3 Opcode Graph ................................................. 39
    4.2.4 Similarity Score Calculation ............................... 41
  4.3 Transcriptase detection add-on architecture ..................... 43
  4.4 JavaScript extraction from web page ............................. 45
  4.5 Purpose of the Shell script ..................................... 48
  4.6 Page validation and clean-up step .............................. 50
  4.7 Performance improvements ...................................... 50
    4.7.1 Fingerprinting web pages .................................. 51
    4.7.2 Whitelisting websites ...................................... 51
4.8 Using other detection techniques .................................. 52

5 Testing .............................................................................. 53
  5.1 Generating Transcriptase variants ................................. 53
  5.2 Similarity scores and add-on performance ...................... 55
    5.2.1 Addition of 550 lines of dead code ....................... 55
    5.2.2 Addition of 5500 lines of dead code .................... 56
    5.2.3 Addition of 15000 lines of dead code ................. 57
  5.3 Test for False Positive rate of the add-on ..................... 59
  5.4 Splitting Transcriptase code ......................................... 59

6 Conclusion and Future Enhancements ................................. 70

APPENDIX

Code snippets ........................................................................ 75
  A.1 Python parser ............................................................. 75
# LIST OF TABLES

1. System Specifications ........................................ 53
2. Scores table of benign web pages ............................. 61
3. Scores table of malware web pages ........................... 62
4. Scores table of benign web pages ............................. 63
5. Scores table of malware web pages ........................... 64
6. Scores table of benign web pages ............................. 65
7. Scores table of malware web pages ........................... 66
8. Summary of max and min Scores ............................... 67
9. Scores table for popular web pages ............................ 68
10. Splitting Transcriptase into several files .................... 69
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JavaScript Sample and its obfuscated version.</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Annual Malware Growth</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Encrypted Malware Replication</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Examples of a polymorphic virus.</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Metamorphic malware with different signatures</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Register Renaming Example</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>Example of code metamorphosis of Evol virus</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>Example of subroutines permutation</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>Sample Java code</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>Sample Java code after applying Transposition</td>
<td>12</td>
</tr>
<tr>
<td>11</td>
<td>Example of changing control flow</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>Subroutine inlining example</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>Subroutine outlining example</td>
<td>14</td>
</tr>
<tr>
<td>14</td>
<td>Meta-language instructions example</td>
<td>15</td>
</tr>
<tr>
<td>15</td>
<td>Permutator output</td>
<td>16</td>
</tr>
<tr>
<td>16</td>
<td>Before Variable/Function-Name randomization</td>
<td>17</td>
</tr>
<tr>
<td>17</td>
<td>After Variable/Function-Name randomization</td>
<td>17</td>
</tr>
<tr>
<td>18</td>
<td>Meta-Language Symbols Example</td>
<td>18</td>
</tr>
<tr>
<td>19</td>
<td>Meta-Language Symbols processing Example</td>
<td>18</td>
</tr>
<tr>
<td>20</td>
<td>Block Diagram of Rhino Engine</td>
<td>21</td>
</tr>
<tr>
<td>21</td>
<td>Euclid’s GCD algorithm</td>
<td>22</td>
</tr>
<tr>
<td>22</td>
<td>Abstract Syntax Tree</td>
<td>23</td>
</tr>
<tr>
<td>Page</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>46</td>
<td>Benign Samples vs Malware Samples</td>
<td>57</td>
</tr>
<tr>
<td>47</td>
<td>Benign Samples vs Malware Samples</td>
<td>58</td>
</tr>
<tr>
<td>48</td>
<td>Graph showing min and max scores of split files</td>
<td>60</td>
</tr>
</tbody>
</table>
CHAPTER 1
Introduction

The arrival of the Internet has completely revolutionized our personal and professional lives. With the rapid growth of the Internet, all the market sectors, social networking services, advertising and non-commercial sectors are using this technology in their workflow. As we become more dependent on the online environment, we can see massive growth of opportunities for IT criminals to take advantage of user systems.

Internet users often share sensitive information like bank account details or other personal information, over the network. As personal computers and mobile phones became an important part in most people's lives, these computers became a hub of user's personal information. In this world of ubiquitous computers and persistent threats from hackers, protecting your computer is a must. Several websites are hacked to be used as distributors of malware, to infect the visitors unknowingly with viruses and malware. A single visit to a such a hacked web page is sufficient for an intruder to get control of a user's machine.

In late 2013, one of the bank's internal computers that are used by employees to process and record daily transactions, had been infected with malware [7]. The malware continuously monitored the bank's activities for several months, sending back images and video feeds to cybercriminals about the bank's daily routines. Then the cybercriminals impersonated bank officers, turned on several cash machines and also transferred millions of dollars from banks into dummy accounts of other countries.
Consider the fact that more than 6,600 benign websites are getting hacked every single day [6]. These legitimate websites are turned into distributors of malware by malicious hackers. Malicious code can be injected into legitimate Javascript of a benign web page. When a user visits such a compromised website, this malicious JavaScript will be executed in the victim’s web browser. Execution of such malicious JavaScript can infect the victim’s personal computer. Most of the times, malicious JavaScript redirects the victim’s web browser to load more malicious code from a remote server. This can be achieved through several means, such as adding an HTML iframe element to a page. Always cybercriminals try to obfuscate the malicious content from detection. HTML provides very few ways to obfuscate the code such as adding an HTML iframe element to a page but the huge number of methods in JavaScript makes it easy to heavily obfuscate the malicious code into Javascript.

1.1 Problem

Malware is malicious software, specifically designed to gain access to the data or to damage the resources without the knowledge of victim [4]. Researchers developed various techniques for malware detection like signature based detection and heuristic analysis. To overcome the malware detection techniques, malware writers came up with different types of malwares among which metamorphic malware is an advanced version. Metamorphic malware is capable of changing its internal structure without altering its functionality from infection to infection. Due to the metamorphic nature, such malware is very difficult to detect. With a huge set of functions, JavaScript makes it possible for virus writers to develop malicious JavaScript code with metamorphic feature. Transcriptase is such a case, which is
Figure 1: JavaScript Sample and its obfuscated version.

designed to infect other JavaScript files in the same folder [5].

1.2 Proposed Solution

In spite of the fact that malware can change their internal structure the priority order of the important commands cannot be changed. Case in point, consider a sample JavaScript code below,

From the code in Figure 1, it is clear that even though the variable names are changed and the order of declaration of $x$ and $y$ (or $x\_\text{renamed}$ and $y\_\text{renamed}$) are changed, the arithmetic operation always follows the declaration of those two variables else the code may give syntactical errors or the wrong result.

So, we can make use of the opcodes statistical information to detect the malware.
1.3 A Browser Plugin for Detecting Malware

Generally JavaScript malware will be injected into benign web pages. When a user visits this infected website, the malicious JavaScript code will be executed in the browser. To prevent this, we can develop an add-on which will monitor the JavaScript content of every web page and the browser can disable the JavaScript execution before the page gets loaded. The add-on will analyze the JavaScript in the background and will enable the JavaScript load, if the web page is found to be benign or else warn the user about malicious content without loading the JavaScript. This procedure can secure the victim’s computer from malware infection. For this research, we can use the Transcriptase metamorphic JavaScript malware.

The remaining of the paper is organized as follows. In Chapter 2, we provide background information on metamorphic malware and with an emphasis on Transcriptase metamorphic JavaScript malware that forms the basis for the research. Also in this chapter, we cover the Rhino Javascript engine. Chapter 3 outlines the details of Firefox add-on development. Then in Chapter 4, we discuss two different static metamorphic detection techniques that we apply to detect the metamorphic JavaScript malware. In Chapter 5, we present the accuracy and performance details of Firefox add-on using opcode similarity detection technique. Our experimental results for the original metamorphic JavaScript appear. Chapter 6 contains the conclusion and consideration for future work.
Malware is a software program intended to do pernicious activities on a client’s computer with the proposition of removing data and misusing assets without his assent. Viruses and worms are the best known types of malware on account of the way in which they spread, instead of their behavior. Malware is now and then utilized widely against government or corporate websites to gather protected data or to disturb their operations by large. Also, malware is regularly utilized against people to steal data, for example, personal identification numbers or bank or credit card details and passwords. As per a survey on data breaches led by Verizon in 2014 [9], Citadel is the preferred banking malware among attackers for stealing individual information. And for stealing money from bank accounts, Zeus is the favorite banking malware.

Figure 1 shows how the malware is swiftly growing in volume day-by-day. In Figure 1, the x-axis specifies the year and the y-axis indicates the number of malware samples generated in the specified year on the x-axis.

Virus writers are aware that signature-based detection with heuristic analysis can be the basis of modern malware detection techniques. So, virus developers have created numerous procedures and techniques to evade signature-based detection. In January 2015, AV-TEST’s CEO said, "Many of the new malware samples are just variants of existing viruses. They have been modified so that they are no longer detected and thus, AV signature updates are required" [10]. Some of the noteworthy techniques used by virus writers to evade signature detection are encryption,
polymorphism and metamorphism.

2.1 Encrypted Malware

The Cascade virus, which initially appeared in late 1986, was the first malware that used encryption to scramble its contents [11]. The Cascade virus is comprised of two parts. The first part is a decryptor and the second part is encrypted malware code. The reason for encryption is to conceal the malware signature, so as to evade signature identification. Later, this technique was adopted by almost every encrypted malware. For the most part, virus writers use extremely simple and weak encryption methods, for example, a repeated XOR with a fixed bit pattern. Cascade malware also used XOR operation as encryption routine because of its symmetrical and reversible feature. In the event that the encryption key of malware was changed after every infection, the encrypted body signature also gets changed. If in case the same decryptor was used, signature detection can make use of the decryptor code's signature to detect the malware.
Figure 3: An Encrypting malware spreads without changing decryptor but the key within decryptor varies from infection to infection. As the key value changes, the encrypted virus body also changes [14].

2.2 Polymorphic Malware

Similar to an encrypted malware, polymorphic malware incorporates an encrypted virus code and a decryptor. Additionally in a polymorphic virus, the decryptor is morphed. During polymorphic malware propagation, not only is the virus code encrypted, but the decryptor also varies from infection to infection. As there is no fixed signature or no fixed decryptor to scan for, no two infections look alike to be exploited by the antivirus program for detection purpose [14]. Polymorphic virus uses code obfuscation techniques, for example, including junk codes or substitution of instructions, to mutate its decryptor [18]. Several techniques are utilized to decrypt the polymorphic virus, such as, cryptanalysis (also called x-ray), emulation and dedicated decryption routines [21].

The first polymorphic malware, 1260, was developed by Mark Washburn in 1990 [15]. And the first widespread polymorphic infection was caused by Tequila and Maltese Amoeba virus, in 1991 [14].
2.3 Metamorphic Malware

Virus writers took the next step and developed an advanced variant of polymorphic malware, known as metamorphic malware. Generally, before infection, polymorphic malware encrypt the virus code and morph the decryptor, while metamorphic malware morph the whole virus code. According to Igor Muttik, "Metamorphics are bodypolymorphics", since polymorphism is applied to the entire virus body [22]. Metamorphic viruses utilizes several code morphing techniques that constitute instruction reordering, data reordering, subroutine inlining, subroutine outlining, register renaming, instruction substitution and dead code insertion [23]. Figure 5 illustrates the metamorphic malware with different signatures.

2.3.1 Register renaming

In December 1998, a metamorphic malware named Win95/Regswap was developed by Vecna [22]. Regswap used register renaming technique to morph the
virus code. In this technique, instructions gets modified to use different registers. As just the register operands gets altered that too in some part of the code instead of whole code, so the complexity of final modified code wouldn’t be high. Figure 6 depicts how the register renaming technique transforms the code.

The bold areas in figure 6 illustrates the similarities of the two different code versions. Thus, a wildcard string, such as 5?B?, could be useful to detect the
malware code [22].

2.3.2 Dead code insertion

Dead code can be a single instruction or a block of instructions, for example, adding NOPs [23], adding 0 to a register, moving between same registers, ORing register with itself, shifting register left 0 bits, jumping to next instruction, incrementing a register immediately followed by decrementing the same register by same value. Inserting dead code or do-nothing instructions is the easiest approach to morph the virus code without modifying its functionality [23]. The Win32/Evol virus, which was found around July 2000 [24], used dead code insertion to obfuscate the signature of a code as illustrated in Figure 7.

2.3.3 Subroutine permutation

Code may contain several subroutines (or) functions and changing the order of this subroutines may not impact the execution of code. Subroutine permutation approach makes use of this advantage i.e., altering the order of subroutines, to
change its internal structure without modifying the functionality of code. A code with \( n \) different subroutines can generate \( n! \) different permutations of subroutines, thus large number of versions of the same code can be generated [4].

### 2.3.4 Equivalent code substitution

Different variants of code can be generated by replacing instruction or block of instructions with an equivalent code. In assembly language there are numerous semantically equivalent instructions, for instance, ‘INC ecx’ is same as ‘ADD ecx, 1’, ‘XOR R1, R1’ is same as ‘MOV R1, 0’ [25].

### 2.3.5 Transposition

Morphed copies of virus code can also be created by changing the order of instructions in the code provided that there is no dependency among instructions, so this approach is also known as instruction permutation [23]. For instance, the code in figure 9 can be transformed to figure 10, as the declaration order of variables doesn’t affect the arithmetic calculation [4].
2.3.6 Changing control flow

The next code obfuscation method involves insertion of a conditional or unconditional branch instruction after a block of instructions. Further, instruction blocks referenced by this branching instructions can be permuted to change the control flow [23]. Zperm malware used this approach to change the internal structure of a code [25]. Figure 11 is an example of changing control flow.

2.3.7 Subroutine inlining and outlining

In subroutine inlining procedure, subroutine/function call replaces its code [23]. Figure 12 illustrates the concept of Subroutine inlining.

On the other hand, code outlining changes a block of instructions into subroutine (or function) and a subroutine call will be included for the newly created subroutine. Figure 13 illustrates how the code outlining approach works.

2.4 Transcriptase

Transcriptase is a metamorphic virus implemented in JavaScript. Whenever Transcriptase is executed, a morphed version of the malware virus gets prepended
Figure 11: Example of changing control flow [25].

```assembly
;Original Program
instruction 1 ; entry point
instruction 2
instruction 3
instruction 4
instruction 5

;Modified Program
instruction 2
jump 3
instruction 4
jump 5
instruction 1 ; entry point
jump 2
instruction 3
jump 4
instruction 5
```

Figure 12: Subroutine inlining example [25].

```assembly
/* some instructions */ /* some instructions */
call S1               mov eax, ebx
        mov eax, ebx
        add eax, 12h
/* some instructions */ push eax
S1:
mul ecx
mov eax, ebx
mov edx, eax
add eax, 12h /* some instructions */
push eax
ret
S2:
mul ecx
mov edx, eax
ret
```

(a) before transformation   (b) after transformation

Figure 12: Subroutine inlining example [25].

to all the JavaScript files in the folder [4]. The infected JavaScript file will become the variant of Transcriptase. By this way Transcriptase propagates and infects the benign JavaScript files.
The metamorphic engine attached to Transcriptase is a self-hosted compiler, which contains its own meta-language source-code. Transcriptase obtains information of its code from meta-language and changes its internal structure.

The format of every line of the meta-code looks like [26]:

\[(\text{Identifier}|\text{Restrictions})\text{instruction}\]

For instance, below are sample meta-instructions:

\[(200)|\text{var b=0}\]
\[(300|200)c+1(b)\]

An Identifier and Restrictions are used by the Permutation function to do code obfuscation. The identifier is unique for every instruction in the entire code and restrictions specify the instructions on which the corresponding instruction is depending. The "instruction" contains the details used to create an actual code.

The compiler creates the new malicious JavaScript code with three steps:

1. Permutation and Variable/Function-Name randomization
2. Code Creation

3. Variable/Function insertion

2.4.1 Permutator

In this phase, the compiler parses through every meta-language instruction, scope by scope (global scope for global instructions and sub-scope for if/for/while/functions) and retrieves the identifiers and restrictions details for each meta-instruction. Later these identifier and restriction details are used by the compiler to perform the permutation of code.

If the restriction details are empty for all the meta-language instructions, then it specifies that all the instructions in the code do not have any dependency instructions. So, the entire code can be permuted in all the possible combinations. For instance, if there are \( n \) lines of code, then the permutator can create \( n! \) variations of the original code.

For instance, consider the code specified in Figure 14 [4]. It contains restriction details for some of the meta-instructions, which means that those instructions have dependencies on other instructions. So all the combinations of the code cannot produce the correct behaviour.
(200) var b=-1
(400) c+1(b)
(100) var a=5
(500) c+n(b,a) // instruction c+n(b,a) means increment b by a:
    i.e. b+=a
(300) var x=8
(600) xWScript.Echo(x)

Figure 15: Possible output of the permutator after parsing the code in Figure 14 [4].

From the code in Figure 14, instruction 400 depends on instruction 200, because the variable "b" has to be defined before it can be incremented; instruction 500 depends on both the instructions 400 and 100; and instruction 600 depends on instruction 300. Figure 15 contains the code, which could be one of the possible output generated by the permutator [4]:

As the growth-rate of the permutation function is very fast, even for the large number of instruction this technique works effectively [26].

2.4.2 Variable/Function-Name randomization

In this step, the keywords like "var", "while", "for", and "def" are searched initially in the code by the compiler and the details of existing hard-coded names in those instructions are retrieved. In other words, the details of all the variable names and function names are gathered by the compiler. These names are replaced with random names by the compiler and also all the valid occurrences of these hard-coded names in the current scope are replaced.

For instance, consider the code in Figure 16. First, the compiler searches for the "var", "function", and "def" keywords and the hard-coded names - num, multiply, inputparam, and twiceval are retrieved. Then it replaces these hard-coded
```javascript
var num=20;
function multiply(inputparam)
{
    return 2 * inputparam;
}
def twiceval=multiply(num)
```

Figure 16: Before Variable/Function-Name randomization [26].

```javascript
var ljkjuytbenst=20;
function trqwsdexcv(awsrsfagfqwxczv)
{
    return 2 * awsrsfagfqwxczv;
}
def bxswdqtyzyqtc=trqwsdexcv(ljkjuytbenst)
```

Figure 17: After randomly changing the Variables/Function-Names of the code in Figure 16 [26].

names with some random names like "ljkjuytbenst" or "awsrsfagfqwxczv". The compiler also replaces all occurrences of the hard-coded names, for example, there are two instances of "inputparam" present in the function, both these names are replaced with the random name. One of the possible changes to the code during this phase is shown in Figure 17.

2.4.3 Meta-Language Symbols

After rearrangement of the instructions and hard-coded names replacement, the compiler generates a valid JavaScript code by parsing through every meta-instruction. These meta-instructions contain meta-level symbols and each of these symbols has specific meaning like number, element, object etc. While parsing meta-instructions, compiler processes these meta-level symbols.

For example, consider the code in Figure 18. Here meta-symbol #n...n#
var number=#n1n#
var str=#"Hello VXers"#
var exp=#x1true1x#
x#01WScript#.Echo(#°+str+°)#

Figure 18: Meta-Language Symbols Example [26].

could become

function SomeFunction(SomeArg){WScript.Echo(SomeArg);}
SomeFunction(str)

Figure 19: Meta-Language Symbols processing Example [26].

specifies any value present in between #n’s (i.e., in place of ...) specifies Number, meta-symbol #"..."# specifies any value present in the place of dots (or ...)
specifies string, any value in #xN...Nx# specifies elements, the values between 
#01... #. ...10# specifies Objects and the symbol "+...+" specifies that the 
variable inside must be given as an argument for a function, if the instruction is 
derived into a function [26].

Figure 19 illustrates how the meta-symbols with objects are processed.

2.4.4 Code Derivation

After processing all the meta-language symbols, the compiler generates 
JavaScript code by parsing the meta-language instructions. During this phase, 
compiler deals with some more meta-instructions that have specific properties as 
mentioned below [26]:

while(initial$var1!var2?operator@action)NNN
1. "initial" specifies the code that is to be executed before the while loop like variable declaration

2. "var1" and "var2" along with the "operator" specifies the while loop condition.

3. "action" specifies the end of the loop instruction like counter increment.

4. "NNN" specifies total number of lines in the loop

\[ \text{cO}(\text{n}|\text{n}|\text{s}) \]

This specifies general way of representing number/string arithmetic instruction. "O" specifies operator like +, -, *, /.

Below are some of the meta-instructions that follow this format,

1. c+1(var1): increment var1 by 1

2. c+n(var1,var2): increment var1 by the number var2

3. c+s(var1,var2): concatenate var1 with the string var2

4. c-1(var1): decrement var1 by 1

5. c-n(var1,var2): decrement var1 by the number var2

6. c*1(var1): multiply var1 by 1

7. c*n(var1,var2): multiply var1 by the number var2

8. c/1(var1): divide var1 by 1

9. c/n(var1,var2): divide var1 by the number var2

2.4.5 Variable/Function insertion

Several variables and functions are defined during the compilation phase because of meta-instructions and obfuscation. These variables are saved in special
arrays instead of being stored in the code. At the end of code derivation phase, they are placed into the code.

Functions can be included between instructions in the global scope. Variables can be included between instructions in the current scope, before they are used for the first time [4]. This phase takes lot of time to complete, as the whole code is checked for multiple times to find suitable positions for the variable/function insertions [26].

2.5 Rhino

During 1997, Netscape began working on developing a variant of Netscape Navigator written in Java [27]. In order to implement the navigator in Java, they built a JavaScript engine entirely in Java, named Rhino. Rhino is open source software and is currently maintained by Mozilla.

Most of the time, JavaScript is utilized as a part of HTML for making interactive webpages. Anyhow, Rhino is not used to create or manipulate webpages; it is an implementation of the core JavaScript. Rhino has the below aspects [27]:

1. Supports JavaScript 1.7 features

2. Allows direct scripting of Java

3. The Rhino Shell can execute the JavaScript code interactively or in batch mode

4. The JavaScript Compiler can compile the JavaScript code into Java classes

5. The JavaScript Debugger for debugging scripts in Rhino
In this research, we use Rhino to translate the JavaScript code into Java classes. The engine supports both compile mode and interpretive mode. During compile mode, the engine first translates each JavaScript file to separate Java class files. These .class files may be executed as Java programs using Rhino runtime support routines. During interpretive mode, JavaScript is compiled and is stored as internal representation of the compiled form instead of byte codes. During runtime this compiled form is evaluated using rhino functions.

2.5.1 Architecture

The four basic blocks in the Rhino JavaScript engine are - the parser, the byte-code generator, the interpreter and the JIT [4]. Figure 20 depicts the block diagram of the Rhino Engine [29].

**Parser**: The input for this module is JavaScript code and the output generated is the Abstract Syntax Tree (AST). The AST is a tree representation of
the abstract syntactic structure of a program. For instance, Figure 21 represents the AST for the GCD code in Figure 22 [30].

![Euclid's GCD Algorithm](image)

**Figure 21: Euclid’s GCD algorithm.**

The non-leaf nodes in the AST specify the operations to be performed, for instance, equal, comparison, arithmetic operation and so on. The leaf nodes in AST specify operands in the source code, for instance, a and b variables [4].

**Byte-Code Generator:** The AST output generated from the parser acts as input to the Byte-Code Generator which then converts the tree into byte code. During the conversion process, the Byte-Code generator picks each code block in the tree and translates it into bytecode. For instance, the byte code for the instruction \( c = a - b \) is shown in Figure 23 [4].

The bytecode in Figure can be interpreted as -

1. Load the values that are stored at offset 1 and offset 2 into registers
2. Subtract these loaded values
3. Store the result at offset 3.
Figure 22: Abstract Syntax Tree for the GCD code in Figure 21 [30].

```plaintext
1 iload_1
2 iload_2
3 isub 3
4 istore_3
```

Figure 23: Sample byte code for the instruction \( c = a - b \) [4].

**Interpreter:** The input for the interpreter is the byte code output of the Byte-Code Generator. The byte code is then converted into machine level code using the Just-in-time (JIT) compiler. During runtime, this generated machine code gets executed. When the byte code in Figure 23 is given as input to the Interpreter, it generates the machine code as shown in Figure 24.
2.5.2 Modification

For this research, we need the opcodes of JavaScript code, so we used the version of Rhino that was modified by the author of [4].

Below changes were made in the modified version [4]:

1. The page load time is directly proportional to compiling time of JavaScript, so Rhino was optimized to convert only small part of JavaScript files. Because of this optimization, the original statistics of the JavaScript code that will be used as part of analysis will get affected. In order to solve this problem the engine was modified to compile JavaScript of any length.

2. Different optimization techniques were used to optimize class files for speed execution. As these optimization techniques also affect the statistics of JavaScript code, they were disabled.

3. Generally opcodes are generated by decompiling the class files. This method consumes a lot of time as this process require the generation of class file and again decompilation of these class files. As the class files generated are of no use, the opcodes are extracted from the code during compilation itself. By following this approach, the time used for decompilation of class files is saved.

4. As the opcodes are extracted before the class file is created, this modification also solved the problem associated with class files optimization.
5. The opcodes are tapped and redirected to the standard output. Thus all the opcodes are printed on the screen. This output can also be saved in a file using the unix redirect operator (>).

The below command is used to run the Rhino engine:

```
java -cp <path_to_rhino_js.jar> org.mozilla.javascript.tools.jsc.Main
<JavaScript_File_Path>
```

Figures 25 and 26 are the screenshots of compiling JavaScript code with the original version and the modified version of Rhino respectively.

Figure 25: JavaScript compilation with Rhino.
Figure 26: JavaScript compilation with modified version of Rhino.
CHAPTER 3
Firefox Add-on Development

An add-on is a piece of software that augments another application. Based on the browser, different terms are used to refer to this software, like add-on, plug-in, or extension. An add-on cannot be executed as stand-alone software. Add-ons are used for different purposes like blocking advertisements and popups, downloading videos, and also to integrate several social network sites.

Below are some of the applications of add-ons

1. An add-on can change the browser interface, which includes changing themes, the look and feel of buttons, the menu bar, and tabs.

2. They are also capable of adding new features to the browser, like providing easy usage of various softwares in the form of toolbars.

3. Add-ons can also modify the behavior of browsers, like customizing the search option or page redirection.

4. Add-ons used as plug-ins let the browser support internet content. These include Flash, Silverlight, and music players like QuickTime, real player, online games, and many more.

5. On many browsers, online privacy is protected using add-ons. There are many types of add-ons that help to control and secure browsing and avoid attacks like preventing the user movements tracking on the browser.
Browser Extensions are supported by Microsoft Internet Explorer starting with version 5 released in 1999. From 2004, Mozilla Firefox supports add-ons. The Opera browser supports extensions starting with the desktop version 10 which was released in 2009. Google Chrome and Safari added support for extensions in 2010. Each browser has different variations in the browser extension syntax and they are compatible to their browser alone i.e., an extension built for one browser doesn’t work on another. For using search engine tools irrespective of browsers, a project named ‘Mycroft’ [13] has been proposed, which is a database of over 20,000 search engine add-ons supported by multiple browsers.

3.1 Firefox vs Chrome

In new era, out of all the browser extensions, Firefox and Chrome are most well known because of their popularity, security, and appearance. Below are some of the notable comparisons between them regarding extensions:

1. Firefox has an outstanding extension base, i.e.,
https://addons.mozilla.org/en-US/firefox/, that offers more capable add-ons compared to all other browsers.

2. Firefox add-ons are very powerful and can perform anything that a Firefox process allows. Security features can be integrated into a Firefox add-on in a much more effective manner than Chrome extensions. So, it is possible to develop more advanced add-ons in Firefox, which would not be achievable on different browsers. Unlike Firefox, Chrome does not trust extensions completely and they provide very constrained APIs. For instance, without the user’s approval, extensions in Chrome can't access the resource present outside of Chrome's sandbox, but a Firefox add-on can access the resource in the
filesystem without the user's permission.

For example, even though there are many Chrome extensions like ScriptSafe, NoScript Lite, which are similar to Firefox's "NoScript", till now no chrome extension is able to provide all the features of Firefox's NoScript because of the Chrome's constrained extension APIs.

3. By providing constrained extension APIs, Chrome presents a permission system and restricts its extensions a bit more for security. Whereas in Firefox, as the add-ons has more privileges, there are chances of infecting a victim's machine. At worst, we may have to re-install the operating system to undo the effect created by a malicious add-on. To avoid these potential issues and to ensure that the add-ons are safe to install, they are manually reviewed before they publicly appear in the Mozilla add-ons gallery.

To detect JavaScript malware, we have to generate opcodes for the JavaScript code in the webpage and to save these opcodes we may need access to the user's filesystem and also we need to execute some other external scripts to validate the JavaScript code based on the saved opcodes. These tasks are only possible with the powerful APIs provided by Firefox, so we decided to implement a Firefox add-on to detect metamorphic malware.

3.2 SDK vs XUL

There are two main ways to build Firefox extensions. The traditional way is using XPCOM (Cross Platform Object Model) and XUL (XML User Interface Language). Much of Mozilla's documentation is focused on XUL add-ons, because this has been around for many years. More details about XUL add-on development
can be found at [32]. The add-on SDK is the newer kind and was built under the Jetpack Project. Jetpack's main agenda is to make it easy to build Firefox add-ons by using HTML, CSS, and JavaScript [3].

It is advisable to use the Add-on SDK because of the advantages it provides compared to XUL [33]:

1. Simplicity: High-level JavaScript APIs provided by the SDK like basic user interface components and their functionality simplify all the common tasks in add-on development.

2. Compatibility: Electrolysis [31], also called e10s, is the project under which Firefox is being developed with a new multiple process architecture. The API's provided by this SDK are designed to be forward-compatible with this new architecture.

3. Security: It is not easy to build insecure add-ons using the SDK. Even the insecure Add-on that was compromised can do much less damage to the victim's machine.

4. No restarts required: To install extensions developed using the SDK, we do not need to restart the browser.

5. Mobile Support: Add-ons can be developed for Firefox mobile using the experimental support provided by SDK 1.5

However, XUL provides a huge number of options for the UI when compared to the SDK, and that’s the reason XUL is used for developing add-ons that require a rich user interface.
In this research, all the add-ons are built using the SDK as it provides simple APIs for developing most of the common tasks.

### 3.3 Chrome Authority Usage

Chrome Authority has nothing to do with Google Chrome. The Mozilla Developer Network (MDN) defines "chrome" as any visible parts of a browser other than the web pages, for instance, tabs, menu bar, and toolbars.

From the beginning of developing the SDK, it was assumed that developers may need to access the underlying browser (or) XPCOM services. So, the add-on SDK was developed to provide "chrome privileges" to the most powerful low-level APIs. The "chrome privileges" grants low-level APIs to access the Components object that gives unrestricted access to the user system.

With chrome privileges, an add-on can perform any function the browser is capable of. These privileges can be obtained by the add-on using the "chrome" module as shown below [1]:

```javascript
var {Cc, Ci} = require("chrome");
```

The "chrome" module returns a Components object, which can be unpacked using the destructuring assignment feature provided by Mozilla JavaScript to obtain the Components.* aliases:

1. Cc, otherwise called Components.classes
2. Ci, otherwise called Components.interfaces
3. Cu, otherwise called Components.utils.
4. Cr, otherwise called Components.results.

5. Cm, otherwise called Components.manager.

It is not advisable to use chrome authority in the add-on code unless it is required because the add-ons that uses chrome authority require extra security review before they are made available for distribution to the public.

### 3.4 Content Scripts

The add-on's main code, including "main.js" and other modules in "lib", can use the SDK high-level and low-level APIs, but can't access web content directly. Whereas content scripts can't use the SDK's APIs, but can access web content.

So if we have to build an Add-on that works based on the content of the web page, then we have to make use of the content scripts to access the web page contents. Content scripts are placed in the data subdirectory and they can be loaded into Add-on using contentScript or contentScriptFile option.

**Communicating with the add-on:**

To enable add-on scripts and content scripts to communicate with each other, each end of the conversation has access to a port object.

1. port.emit() is used to send messages from one side to the other
2. port.on() is used to receive messages sent from the other side

Messages are asynchronous i.e., after sending the message, sender continues processing without waiting for a reply from the recipient.

The add-on code in Figure 28 adds a button to Firefox. When the user clicks
this button, add-on attaches a content script to the active tab, sends the **addon-message** to content script. When content script receives add-on message, it will retrieve the first paragraph from the loaded web page and send it to add-on along with **script-response** message. As soon as the add-on receives a response from content script, it logs the first paragraph.
Figure 28: Communication among add-on and content script using code [1].
CHAPTER 4
Implementation

This chapter focuses on the implementation details of two add-ons:

1. a malicious add-on, which can infect the victim’s filesystem.

2. a Transcriptase detection add-on, which can detect the metamorphic malware embedded in the web page.

4.1 Malicious add-on

Generally, browsers like Firefox, Chrome, and Opera do not allow access to the client filesystem using JavaScript. Even though creating a file is possible in IE using ActiveX objects, the client must enable ActiveX scripts on their system for the ActiveX object related code to execute properly [17].

Firefox add-ons are very powerful because of the high-level APIs that the SDK provides. The SDK has a file I/O module which provides access to the client’s filesystem.

A malicious add-on was created to demonstrate the way that a victim’s machine may get infected by a malicious add-on. The basic functionality of this add-on is it provides the statistic value i.e., the total JavaScript bytes in the page loaded by the user, as shown in Figure 29.

The user expects this functionality and installs the malicious add-on, but this add-on also has hidden functionality. Whenever it finds that the web page content has Transcriptase in it, it finds all the JavaScript files present in the filesystem and...
prepends them with Transcriptase code and thus it infects the victim’s filesystem. Figure 30 shows the size of our sample JavaScript files before infection and figure 31 shows the size of these files after infection. There is a huge difference in file
sizes before and after infection. This infection will remain unknown to the user until the infected files are checked.

4.2 Transcriptase detection add-on

As discussed in Section 2.4, the Transcriptase virus uses different techniques to change its internal structure in order to evade the signature based detection strategy. The Transcriptase detection add-on can detect the Transcriptase malware included in the webpage and notifies the user about the presence of malware without loading the JavaScript malware.
4.2.1 Malware Detection Technique

Despite the fact that metamorphic malware continuously changes its internal structure to stay undetected, still for maintaining its functionality malware places similar instructions (that implements the functionality) somewhere in the code. Thus, all the morphed copies maintain the same statistical distribution of instructions. Different malware detection strategies are designed to make use of these statistical properties like Hidden Markov Models, Opcode Graph Similarity, Simple Substitution Distance, and Singular Value Decomposition.

As mentioned in [4], if the files are having highly similar opcode statistics, then Opcode Graph Similarity and Singular Value Decomposition can classify them better than the Hidden Markov Model and Simple Substitution Distance. Opcode Graph Similarity and Singular Value Decomposition are very sensitive to deadcode, but from the results mentioned in [4] these strategies won’t be able to distinguish between benign code and virus code only after adding 5000 and 9000 deadcode functions into the virus code respectively. And it is extremely uncommon for a web page to have this much dead code. Adding to this, the ROC curves in [4] shows that Opcode Graph Similarity performs better than Singular Value Decomposition with less than 1000 dead code function insertions.

We used Opcode Graph Similarity technique in the Transcriptase detection add-on.

4.2.2 Opcode Graph Similarity Technique

In [2], Anderson introduced a malware detection technique which is based on analysis of graphs that are constructed using the opcodes of the malware code and test code. In this technique, initially opcodes are extracted from the malware code
and a weighted directed graph is built using the sequence of opcodes. Similarly, a
graph is built for the code to be tested. The Manhattan distance between these two
weighted graphs specifies the test file score [4].

4.2.3 Opcode Graph

A weighted directed graph built using the sequence of opcodes is known as the
‘Opcode Graph’. Each node of this graph specifies a distinct opcode in opcode
sequence. A directed edge exists from node\textsubscript{A} to node\textsubscript{B}, if node\textsubscript{B}‘s opcode follows the
node\textsubscript{A}‘s opcode in the opcode sequence. The weight of the edge from node\textsubscript{A} to
node\textsubscript{B} specifies the total number of times that node\textsubscript{B}‘s opcode follows node\textsubscript{A}‘s
opcode in the entire code.

1 \textbf{PUSH}
2 MOV
3 \textbf{SUB}
4 AND
5 MOV
6 TEST
7 JZ
8 INT
9 MO
10 VZX
11 \textbf{AND}
12 MO
13 VZX
14 OR
15 MO
16 CALL
17 \textbf{LEAVE}
18 RETN
19 ALIGN
20 \textbf{PUSH}
21 \textbf{MOV}
22 \textbf{MOV}
23 \textbf{PUSH}
24 \textbf{PUSH}
25 SUB
26 MOV
27 MOV
28 CALL
29 AND
30 SUB
31 CALL
32 \textbf{MOV}
33 \textbf{CALL}
34 \textbf{MOV}
35 XOR
36 \textbf{MOV}
37 \textbf{MOV}
38 \textbf{MOV}
39 CALL
40 \textbf{MOV}

Figure 32: Sample opcode sequence.

Figure 32 shows the sample opcode sequence. The adjacency matrix in
### Figure 33: Weight counts adjacency matrix for opcodes in Figure 32.

<table>
<thead>
<tr>
<th></th>
<th>ALIGN</th>
<th>AND</th>
<th>CALL</th>
<th>INT</th>
<th>JZ</th>
<th>LEAVE</th>
<th>MOV</th>
<th>MOVZX</th>
<th>OR</th>
<th>PUSH</th>
<th>RETN</th>
<th>SUB</th>
<th>TEST</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>AND</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CALL</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>INT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JZ</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LEAVE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOV</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MOVZX</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PUSH</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RETN</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SUB</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TEST</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>XOR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 33 specifies the weights of the edges formed between these opcodes. For instance, we can find that intersection entry between the CALL row and MOV column has a weight value of 3, which means there are three occurrences of a MOV instruction immediately followed by a CALL instruction in the opcode sequence i.e., at line numbers 15, 27, and 32 in Figure 32.

All the weight counts in Figure 33 are converted into probability values by dividing each row entry by the corresponding row sum. Figure 34 shows the weight probabilities for the Figure 33. Each weight probability specifies the probability of occurrence of a particular opcode, immediately after the selected opcode [4].

Figure 35 shows the opcode graph for the probability matrix in Figure 34.
### 4.2.4 Similarity Score Calculation

After creating probability matrices for the malware file and the test file, similarity between two files is calculated by taking the Manhattan distance between two probability matrices. Consider $A$ as the probability matrix of file 1 and each element in $A$ is denoted as $A_{i,j}$ where $i$ and $j$ specifies $i^{th}$ row and $j^{th}$ column respectively. Similarly $B$ is the probability matrix of file 2 and each element in $B$ is denoted as $B_{i,j}$. Similarity between matrix $A$ and $B$, is calculated as below [8],

$$\text{Similarity score} = \frac{1}{N^2} \left( \sum_{i,j=0}^{N-1} |a_{i,j} - b_{i,j}|^2 \right)$$

where $N$ is total number of distinct opcodes present in the combination of both files.

Before using the similarity score, we have to determine the threshold score which distinguishes between benign files and malware files. The threshold value is

---

<table>
<thead>
<tr>
<th></th>
<th>ALIGN</th>
<th>AND</th>
<th>CALL</th>
<th>INT</th>
<th>JZ</th>
<th>LEAVE</th>
<th>MOV</th>
<th>MOVZX</th>
<th>OR</th>
<th>PUSH</th>
<th>RETN</th>
<th>SUB</th>
<th>TEST</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$2/3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/3$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CALL</td>
<td>0</td>
<td>$1/5$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/5$</td>
<td>$3/5$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>INT</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JZ</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LEAVE</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOV</td>
<td>0</td>
<td>$4/14$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$5/14$</td>
<td>$1/14$</td>
<td>0</td>
<td>$1/14$</td>
<td>0</td>
<td>$1/14$</td>
<td>$1/14$</td>
<td>$1/14$</td>
<td>$1/14$</td>
</tr>
<tr>
<td>MOVZX</td>
<td>0</td>
<td>$1/2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>OR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PUSH</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$2/4$</td>
<td>0</td>
<td>0</td>
<td>$1/4$</td>
<td>0</td>
<td>$1/4$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RETN</td>
<td>$1/1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SUB</td>
<td>0</td>
<td>$1/3$</td>
<td>$1/3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TEST</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>XOR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>$1/1$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 34: Probability matrix for weights adjacency matrix in Figure 33.
Figure 35: opcode graph for the probability matrix in Figure [8].

determined as follows [8],

1. Construct opcode graphs for all the variants of metamorphic malwares.

2. Construct opcode graphs for all the benign files.

3. Calculate the similarity scores for all pairs of malwares.

4. Calculate the similarity scores for every benign file against malware from step 1.

5. Determine a threshold value using the scores calculated in steps 3 and 4.
4.3 Transcriptase detection add-on architecture

Figure 36 shows the detection add-on architecture. Each component in the architecture is made up of one or more files. Figure 37 depicts the directory structure of detection add-on.

1. "content-script.js" uses jQuery code to find the JavaScript content in web page, so included "jquery-1.10.0.min.js" file in the "/data" directory to enable this functionality. content-script.js represents "Content Script" component.

2. The Java files in the add-on are used to download the files, and to calculate the similarity score. All these files are placed in the "/data/java" directory. "Java Downloader" is a combination of DownloadThread, DownloaderApp,
Figure 37: Transcriptase detection add-on directory structure

ImageDownloader, and Lock class files. CheckOpcodes and GetScore class files represents "opcode graph similarity detector".

3. "js.jar" helps the add-on to use the Rhino JS engine.

4. "opcode.sh" is a shell script to invoke Java files. As shell scripts won't work on Windows, the "opcode.bat" file is included to invoke Java files on Windows.

5. The "malware_opcodes.txt" file specifies "Malware opcodes" component. This
file contains the opcodes of Transcriptase family malware which is required to validate web page JS content using the opcode graph similarity technique.

More details about these files and the architecture components are covered in subsequent sections.

4.4 JavaScript extraction from web page

As soon as the user enters a web page link, the browser loads the JavaScript content along with HTML and CSS on the page. Before extracting the JS from web page, we have to disable JS load in the browser to prevent the execution of JS malware code.

Enabling/disabling JS feature deals with browser preferences. The preferences system of Mozilla browser can be accessed using XPCOM interfaces like \textit{nsIPrefService} and \textit{nsIPrefBranch}. The below code is used to disable JavaScript code,

\begin{verbatim}
  var prefSrv = this.prefService = Cc["@mozilla.org/preferences-service;1"]
    .getService(Ci.nsIPrefService);
  var PBI = Ci.nsIPrefBranch2;
  this.mozJSPref = prefSrv.getBranch("javascript.").QueryInterface(PBI);
  this.mozJSPref.setBoolPref("enabled", false);
\end{verbatim}

Figure 38: Add-on code to disable JavaScript load on web page.

The code in Figure 38 is interpreted as follows: Line 1 in the above code retrieves the preference services of Mozilla. \textit{nsIPrefBranch2} interface, in line 2, allows the add-on to listen to the changes to preferences. Line 3 retrieves the "javascript" preference and queues \textit{nsIPrefBranch2} interface using QueryInterface(). Later, setBoolPref() method is used to disable the JS by setting "false" to...
"enabled". Similarly the below line of code enables JS load,

```javascript
this.mozJSPref.setBoolPref("enabled", true);
```

After disabling JavaScript, as explained in Section 3.4, the content script i.e., "content-script.js", is used to extract JS. "content-script.js" uses jQuery element selector to find all the `<script>` elements and extract the JavaScript instructions contained in `<script>` tags as shown below:

```javascript
var code="";
$("script").each(function(){
    code=code+$(this).html();
});
```

Sometimes, JavaScript code is also placed in an external file and the location of the external JavaScript file is specified in the web page using a src attribute of a `<script>` element as shown below:

```html
<script src="external_javascript.js"></script>
```

There is a chance that this external files may contain JavaScript malware code, so using the code in Figure 39, all the external file's URLs are extracted from the web page. Following is the explanation of the code snippet in Figure 39:

1. "window.location.protocol" returns the protocol of the current web page URL along with colon(:). For instance, "http:", "https:“, "ftp:”.
   "window.location.host" returns the host name of the web page. For instance, the hostname of "http://www.somewebsite.com/tryit.jsp?filename=sample_code" is
var baseUrl = window.location.protocol + "//" + window.location.host + "/";
var Urls = "";
var regex = new RegExp("^\(?[a-z]+:)?://", "i");

$("script[src]").each(function(){
    var sourceurl = $(this).attr("src");
    if(!regex.test(sourceurl)) {
        Urls=Urls+baseUrl;
    }
    Urls=Urls+(sourceurl.replace(/\^/+, "")+"\n");
});

Figure 39: JavaScript to extract all the external script file URLs

"www.somewebsite.com". So, line 1 creates a base URL of the web page.

2. line 4 uses a jQuery element selector to retrieve the external file locations defined in src attribute of <script> tag.

3. line 6 uses regex to test whether the external files location is relative or absolute path.

4. Line 8 contains the logic for prepending the base URL to an external file location, if the external file location is relative path.

5. Finally, the "Urls" variable contains all the external local URLs, and these URLs will be saved in a temporary file.

Later, the add-on invokes the opcodes.sh file as shown in Figure 41, which performs the following two functions:

**Creates a temporary file**

"TmpD" returns the temporary directory location of the OS. In line 1, "opcodes.tmp" filename is concatenated to temporary directory path and
getFile() method returns a nsIFile object referring to "<TMP_DIR>/opcodes.tmp" location. Then createUnique() method creates the requested temporary file.

**Invokes opcode.sh**

The nsIProcess interface is used to execute a process. nsIProcess requires executable name to execute and if the executable file requires any parameters then these parameters need to be passed as args[] to the nsIProcess.

1. Line 3 and 4 creates a nsIFile object referring to executable "/bin/sh".
2. Line 5 and 6 creates an instance of process and initializes it to "/bin/sh" executable.
3. Line 7 and 8 adds both "opcode.sh" file path and temporary file path to "args" array. Then the process is executed using run() which executes the below command internally,

```
$ /bin/sh
/Users/sravan2j/Downloads/TranscriptaseDetectionAddon/data(opcode.sh
/tmp/opcodes.tmp
```

Figure 40: command that invokes opcode.sh internally

4.5 **Purpose of the Shell script**

The opcodes.sh code, shown in Figure 42, performs the following three functions:

1. Line 1 executes following Java files - DownloadThread.class, DownloaderApp.class, ImageDownloader.class, and Lock.class, to download
Cu.import("resource://gre/modules/FileUtils.jsm");

// create a temporary file
var file = FileUtils.getFile("TmpD", ["opcodes.tmp"]);
file.createUnique(Ci.nsIFile.NORMAL_FILE_TYPE, 0600);

var file = Cc["@mozilla.org/file/local;1"].
    createInstance(Ci.nsILocalFile);
file.initWithPath("/bin/sh");

var process = Cc["@mozilla.org/process/util;1"]
    .createInstance(Ci.nsIProcess);
process.init(file);

var args =
    ["/Users/sravan2j/Downloads/TranscriptaseDetectionAddon/data/opcode.sh"];
// append temporary file path to parameters
args.push(tmpFile.path);
process.run(true, args, args.length);

Figure 41: add-on code that creates temporary file and invokes opcode.sh with temporary file

all the external scripts. This java files uses multi threading approach to
download all the external scripts in parallel which reduces the total download
time.

2. After the above step, the entire JavaScript content will be saved in the
/tmp/JSStatements.js file. Line 2 takes the JSStatements.js file as input and
generates opcodes for the JavaScript code in JSStatements.js using the Rhino
JS engine. The output of this step is the "/tmp/opcodes.txt" file.

3. Line 3 executes CheckOpcodes java code which calculates the similarity score
using the opcode similarity technique, between malware_opcodes.txt and the
opcodes.txt file. The output of this step is redirected to "$1", which refers to
the arguments passed to opcodes.sh. The bash command in Figure 40 shows
that the /tmp/opcodes.tmp file is passed as an argument while calling opcode.sh.

```bash
1. java -Xmx500m -cp "." data/DownloaderApp data/externalUrls.txt
2. java -cp "./data/js.jar" org.mozilla.javascript.tools.jsc.Main
   /tmp/JSStatements.js > /tmp/opcodes.txt
3. java -cp "./data" CheckOpcodes data/malware_opcodes.txt /tmp/opcodes.txt > $1
```

Figure 42: opcode.sh shell script code

4.6 Page validation and clean-up step

The add-on gets the similarity score from the opcodes.tmp file. If the score is less than the threshold value i.e., 0.01, then the web page is considered as a malicious page or else it is a benign page.

1. If the page is benign, then it enables JavaScript and reloads the web page.

2. If the page is malicious, then the web page won't be loaded; instead a prompt is displayed to the user regarding the malware.

At the end, all the temporary files created will be removed as part of the clean-up step.

4.7 Performance improvements

As the add-on performs lot of steps to validate the web page, the execution time will be more. So, instead of validating every web page every time, we can skip the validation during the following scenarios:
4.7.1 Fingerprinting web pages

The hashcode of the benign web page should be saved in the user directory. In the future, when user visits the same web page and if the internal content of the page is not changed from the last visit, then the hash code of the page remains the same as the one that was saved on user's machine. In this case, we can safely skip the validation of the web page.

The disadvantage with this approach is that it consumes the user's system memory as it saves the hashcode for every web page the user visits.

We can also improve this approach by saving the hash codes in the cloud repository. Whenever any user visits the web page, the add-on connects with the cloud repository and checks if this web page was already validated by any user earlier or not. If it was validated, is the web page hash code the same? And what is the validation result? If the hash code is not in the cloud, then it will be validated by the current user's plugin and the result will be stored in the cloud, so that this data will be useful for other users. Because of this approach, the user's system memory will be saved and also at any point of time, the web page is validated only once by any user. Necessary security measures should be taken inorder to prevent the attacks like man-in-the-middle attack, cloud data tampering.

4.7.2 Whitelisting websites

Some popular websites are highly secured and regularly monitored, like Google, Facebook, Amazon etc. These websites can be added to benign page list by the user, so that they won't be validated by the add-on.

The disadvantage of this approach is that it involves a risk of infection if the
whitelisted web pages are infected by malware.

4.8 Using other detection techniques

Currently this add-on uses only opcode graph similarity detection. Other detection techniques can be used in the add-on by simply changing line 3 of opcode.sh, shown in 42, to execute a program that implements another detection technique instead of executing the CheckOpcodes program. The new program should accept "malware_opcodes.txt" and "/tmp/opcodes.txt" as input files and the similarity score should be saved in the "/tmp/opcodes.tmp" file. No other changes are required.
CHAPTER 5

Testing

To check the accuracy and performance of the add-on, we used malware web pages and benign web pages. To create malware samples, we generated different variants of Transcriptase malware. For benign web pages, we retrieved the JavaScript dead code from http://tools.w3clubs.com/joyjo/.

Entire testing is performed on a system with the configuration specified in Table 1:

<table>
<thead>
<tr>
<th>Table 1: System Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Model</td>
</tr>
<tr>
<td>Processor</td>
</tr>
<tr>
<td>RAM</td>
</tr>
<tr>
<td>Storage</td>
</tr>
<tr>
<td>Firefox version</td>
</tr>
<tr>
<td>SDK version</td>
</tr>
<tr>
<td>Rhino version</td>
</tr>
<tr>
<td>Java version</td>
</tr>
</tbody>
</table>

5.1 Generating Transcriptase variants

Transcriptase was written in JScript, so in a windows system it can be executed by simply double clicking it. From my observation, the generation of each version takes around 15 minutes.

As explained in Section 2.4, Transcriptase carries its source code as meta
instructions and on each execution it creates different variant of its JS source, then
prepends that JS code to all the JavaScript files in its directory. So, I followed the
below steps to create 100 versions:


2. Executed Transcriptase, which infects the new empty JavaScript file and
   converts it to another variant of Transcriptase.

3. Move the older version Transcriptase (or creator Transcriptase) to different
   folder.

4. Then created an empty JavaScript file in the current folder where the new
   Transcriptase variant exists.

5. Executed the new variant to infect the empty JavaScript file. Go to Step 3 if
   the required number of variants aren't generated.

The code in Figure 43 automates the above mentioned steps.

```plaintext
FOR %%A IN (1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
   25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48
   49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
   73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96
   97 98 99 100) DO (  
   transcriptase.js  
   REN transcriptase.js "var%%~nA.*"  
   MOVE "var%%~nA.*" "C:\Users\Sravan\Downloads\Transcriptase\versions"  
   REN empty.js transcriptase.js  
   COPY "C:\Users\Sravan\Downloads\Transcriptase\template\empty.js" .  
)  
PAUSE
```

Figure 43: Batch script that automates the Transcriptase variants generations
5.2 Similarity scores and add-on performance

Included console.log() functions in the add-on to log the following details - opcode similarity score and the add-on execution time taken to validate the web page. The below sub section deals with comparison of these details for benign and malware web pages. For testing the add-on, we used 100 samples of benign web page and morphed malware web pages. Malware web pages are morphed by adding randomly generated junk code to it.

5.2.1 Addition of 550 lines of dead code

For this experiment, we used benign web page samples with 550 lines of junk code and also added the same amount of randomly generated junk code to malware
web pages. Tables 2 and 3 contain the details of scores and execution time for the benign and malware samples, respectively. From the table values, we can see that the scores for benign web pages are in the order of $10^{-3}$ whereas the scores for malware web pages are in the order of $10^{-4}$. The graph in Figure 45 clearly shows that the add-on is able to distinguish malware web pages and benign web pages correctly. Only 3 out of 100 malware samples have score similar to benign web pages.

![Figure 45: Benign samples scores vs malware samples scores with the addition of 550 lines of dead code](image)

5.2.2 Addition of 5500 lines of dead code

This experiment is same as the above experiment except that here 5500 lines of dead code was included in malware and benign web pages instead of 500 lines. Tables 4 and 5 contains the details of scores and execution time for this experiment. From the table values, we can see that the scores for benign web pages and malware
web pages are still in the order of $10^{-3}$ and $10^{-4}$, respectively. The graph in Figure 46 clearly shows that even after adding 5500 lines of code, the add-on is able to distinguish malware web pages and benign web pages correctly. Only 3 out of 100 malware samples have scores similar to benign web pages.

![Figure 46: Benign samples scores vs malware samples scores with the addition of 5500 lines of dead code](image)

5.2.3 Addition of 15000 lines of dead code

Here, 15000 lines of dead code were included in malware and benign samples. From the tables 6 and 7, it is clear that scores of malware web pages are varied by a very negligible value when compared to previous experiments. So as shown in Figure 47, the add-on is still able to distinguish malware web pages and benign web pages correctly.

As mentioned in Section 4.2.4, the scores of malware files and benign files can
be compared to calculate the threshold score value of the opcode similarity technique. From the tables 2, 3, 4, 5, 6, and 7, except 9 malwares all other malware scores are in between 0.000369822260 and 0.000369822750. The 9 exception malware scores are between 0.001423994000 and 0.001423994700. And the benign web page scores are between 0.001479288900 and 0.001479291400. The same information is represented with table 8

The threshold value can be any value between 0.00142 and 0.00147. If the lower percentage of false negative rate is acceptable, then the threshold score can be chosen between 0.00036 and 0.00147. In case of malware detection, it is always better to have fewer false negatives, so I use 0.00145 as the threshold score value for the add-on.
5.3 Test for False Positive rate of the add-on

I tested the add-on on popular web sites to detect the "false positive" rate of the add-on. The popular web site links are retrieved from [34].

The table 9 contains the scores and execution time details. All the scores in the table are more than the chosen threshold value (i.e., 0.00145), which means that the add-on validated all the web pages as benign web pages i.e., add-on has zero false positive rate.

5.4 Splitting Transcriptase code

Transcriptase can be split into several external JS files and then the external files can be included in a web page. So, the following experiment was performed to calculate the scores of split files by dividing Transcriptase into a various number of files.

The code was split based on functions count. The experiment was started by dividing the Transcriptase code into two files with almost equal number of functions and then continued till the split files count reaches 76. A parser was developed in Python to detect the valid start and end point of the JavaScript functions and to properly split the Transcriptase code. Thus the resultant split files are syntactically correct. Corresponding parser code is shown in Appendix A, Section A.1.

Table 10 shows the results for various splits. The "Max" and "Min" column specify the maximum and minimum similarity score among the split files, respectively. The "count" column specifies the number of files the Transcriptase code was split into.

Figure 48 is a graphical representation of the Table 10 values. The graph...
clearly shows that even when the code was split, the **minimum score among the** split files is always less than threshold which means that there always exists at least one split file with score a less than threshold score. Thus, it is possible to detect the malware even by testing all the external files separately. So, we can validate all the external scripts parallely to increase the performance of the add-on.

![Figure 48: Graph showing min and max scores of Transcriptase split files.](image-url)
<table>
<thead>
<tr>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00147929</td>
<td>1136</td>
<td>0.00147929</td>
<td>1135</td>
<td>0.00147929</td>
<td>1482</td>
<td>0.0014792903</td>
<td>1232</td>
</tr>
<tr>
<td>0.00147929</td>
<td>1172</td>
<td>0.0014792896</td>
<td>1124</td>
<td>0.0014792896</td>
<td>1304</td>
<td>0.0014792907</td>
<td>1115</td>
</tr>
<tr>
<td>0.00147929</td>
<td>1150</td>
<td>0.0014792907</td>
<td>1113</td>
<td>0.00147929</td>
<td>1245</td>
<td>0.00147929</td>
<td>1308</td>
</tr>
<tr>
<td>0.00147929</td>
<td>1125</td>
<td>0.0014792903</td>
<td>1178</td>
<td>0.0014792907</td>
<td>1274</td>
<td>0.0014792896</td>
<td>1159</td>
</tr>
<tr>
<td>0.0014792893</td>
<td>1080</td>
<td>0.0014792896</td>
<td>1133</td>
<td>0.0014792896</td>
<td>1235</td>
<td>0.00147929</td>
<td>1125</td>
</tr>
<tr>
<td>0.00147929</td>
<td>1223</td>
<td>0.0014792903</td>
<td>1133</td>
<td>0.0014792903</td>
<td>1242</td>
<td>0.0014792903</td>
<td>1228</td>
</tr>
<tr>
<td>0.0014792896</td>
<td>1125</td>
<td>0.00147929</td>
<td>1134</td>
<td>0.0014792903</td>
<td>1246</td>
<td>0.0014792896</td>
<td>1180</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1132</td>
<td>0.0014792893</td>
<td>1106</td>
<td>0.0014792903</td>
<td>1250</td>
<td>0.0014792893</td>
<td>1131</td>
</tr>
<tr>
<td>0.001479291</td>
<td>1120</td>
<td>0.00147929</td>
<td>1142</td>
<td>0.0014792903</td>
<td>1257</td>
<td>0.00147929</td>
<td>1136</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1198</td>
<td>0.0014792896</td>
<td>1159</td>
<td>0.0014792903</td>
<td>1255</td>
<td>0.0014792907</td>
<td>1132</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1134</td>
<td>0.0014792896</td>
<td>1125</td>
<td>0.0014792903</td>
<td>1214</td>
<td>0.0014792896</td>
<td>1141</td>
</tr>
<tr>
<td>0.0014792907</td>
<td>1131</td>
<td>0.0014792907</td>
<td>1122</td>
<td>0.0014792903</td>
<td>1152</td>
<td>0.00147929</td>
<td>1134</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1108</td>
<td>0.00147929</td>
<td>1176</td>
<td>0.0014792896</td>
<td>1116</td>
<td>0.0014792896</td>
<td>1124</td>
</tr>
<tr>
<td>0.00147929</td>
<td>1110</td>
<td>0.0014792903</td>
<td>1134</td>
<td>0.0014792907</td>
<td>1138</td>
<td>0.0014792903</td>
<td>1410</td>
</tr>
<tr>
<td>0.00147929</td>
<td>1124</td>
<td>0.0014792893</td>
<td>1113</td>
<td>0.00147929</td>
<td>1105</td>
<td>0.0014792907</td>
<td>1123</td>
</tr>
<tr>
<td>0.0014792893</td>
<td>1122</td>
<td>0.00147929</td>
<td>1168</td>
<td>0.0014792907</td>
<td>1093</td>
<td>0.00147929</td>
<td>1150</td>
</tr>
<tr>
<td>0.0014792907</td>
<td>1116</td>
<td>0.00147929</td>
<td>1135</td>
<td>0.0014792903</td>
<td>1175</td>
<td>0.0014792903</td>
<td>1175</td>
</tr>
<tr>
<td>0.0014792896</td>
<td>1117</td>
<td>0.0014792907</td>
<td>1118</td>
<td>0.0014792907</td>
<td>1121</td>
<td>0.0014792903</td>
<td>1116</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1198</td>
<td>0.0014792903</td>
<td>1148</td>
<td>0.0014792903</td>
<td>1127</td>
<td>0.0014792893</td>
<td>1125</td>
</tr>
<tr>
<td>0.00147929</td>
<td>1122</td>
<td>0.0014792907</td>
<td>1102</td>
<td>0.0014792903</td>
<td>1167</td>
<td>0.0014792896</td>
<td>1143</td>
</tr>
<tr>
<td>0.00147929</td>
<td>1137</td>
<td>0.0014792893</td>
<td>1144</td>
<td>0.0014792903</td>
<td>1105</td>
<td>0.0014792903</td>
<td>1223</td>
</tr>
<tr>
<td>0.00147929</td>
<td>1204</td>
<td>0.0014792896</td>
<td>1098</td>
<td>0.0014792907</td>
<td>1113</td>
<td>0.0014792903</td>
<td>1120</td>
</tr>
<tr>
<td>0.0014792907</td>
<td>1130</td>
<td>0.00147929</td>
<td>1152</td>
<td>0.0014792896</td>
<td>1111</td>
<td>0.0014792907</td>
<td>1140</td>
</tr>
<tr>
<td>0.00147929</td>
<td>1104</td>
<td>0.0014792903</td>
<td>1129</td>
<td>0.00147929</td>
<td>1140</td>
<td>0.00147929</td>
<td>1213</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1119</td>
<td>0.00147929</td>
<td>1215</td>
<td>0.00147929</td>
<td>1149</td>
<td>0.00147929</td>
<td>1119</td>
</tr>
</tbody>
</table>

Table 2: Table illustrating the scores and add-on execution time for 100 benign web pages, in four columns (i.e., 25 samples per column). Benign webpages are generated with 550 lines of dead code.
Table 3: Table illustrating the scores and add-on execution time for 100 malware web pages, in four columns (i.e., 25 samples per column). Malware webpages are morphed with 550 lines of dead code.
<table>
<thead>
<tr>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0014792907</td>
<td>2287</td>
<td>0.0014792903</td>
<td>1937</td>
<td>0.00147929</td>
<td>2379</td>
<td>0.0014792903</td>
<td>3148</td>
</tr>
<tr>
<td>0.00147929</td>
<td>2447</td>
<td>0.0014792893</td>
<td>2813</td>
<td>0.0014792896</td>
<td>1966</td>
<td>0.0014792907</td>
<td>2210</td>
</tr>
<tr>
<td>0.00147929</td>
<td>2215</td>
<td>0.0014792903</td>
<td>2153</td>
<td>0.0014792903</td>
<td>2728</td>
<td>0.0014792903</td>
<td>2094</td>
</tr>
<tr>
<td>0.0014792907</td>
<td>2376</td>
<td>0.0014792903</td>
<td>2604</td>
<td>0.00147929</td>
<td>2167</td>
<td>0.0014792903</td>
<td>2068</td>
</tr>
<tr>
<td>0.00147929</td>
<td>2188</td>
<td>0.0014792914</td>
<td>2221</td>
<td>0.0014792903</td>
<td>2022</td>
<td>0.0014792907</td>
<td>2462</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>2661</td>
<td>0.00147929</td>
<td>2152</td>
<td>0.0014792907</td>
<td>2383</td>
<td>0.0014792893</td>
<td>2060</td>
</tr>
<tr>
<td>0.0014792907</td>
<td>1989</td>
<td>0.0014792903</td>
<td>2404</td>
<td>0.0014792903</td>
<td>2033</td>
<td>0.00147929</td>
<td>2074</td>
</tr>
<tr>
<td>0.00147929</td>
<td>2137</td>
<td>0.00147929</td>
<td>2260</td>
<td>0.00147929</td>
<td>2463</td>
<td>0.0014792903</td>
<td>2254</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>2001</td>
<td>0.0014792907</td>
<td>2110</td>
<td>0.0014792903</td>
<td>2535</td>
<td>0.0014792893</td>
<td>2564</td>
</tr>
<tr>
<td>0.0014792907</td>
<td>2003</td>
<td>0.0014792893</td>
<td>2226</td>
<td>0.0014792903</td>
<td>2026</td>
<td>0.0014792903</td>
<td>2084</td>
</tr>
<tr>
<td>0.00147929</td>
<td>2646</td>
<td>0.0014792903</td>
<td>2134</td>
<td>0.0014792907</td>
<td>2030</td>
<td>0.0014792903</td>
<td>2406</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1969</td>
<td>0.0014792903</td>
<td>2115</td>
<td>0.0014792907</td>
<td>1944</td>
<td>0.0014792903</td>
<td>2437</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1972</td>
<td>0.0014792903</td>
<td>2003</td>
<td>0.0014792893</td>
<td>2239</td>
<td>0.0014792889</td>
<td>2317</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1937</td>
<td>0.0014792893</td>
<td>1989</td>
<td>0.0014792893</td>
<td>2270</td>
<td>0.0014792903</td>
<td>2001</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1992</td>
<td>0.0014792903</td>
<td>2599</td>
<td>0.00147929</td>
<td>1954</td>
<td>0.0014792903</td>
<td>1933</td>
</tr>
<tr>
<td>0.0014792896</td>
<td>2612</td>
<td>0.0014792903</td>
<td>2599</td>
<td>0.00147929</td>
<td>1961</td>
<td>0.00147929</td>
<td>1994</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>2092</td>
<td>0.0014792896</td>
<td>2971</td>
<td>0.0014792893</td>
<td>2240</td>
<td>0.0014792903</td>
<td>1667</td>
</tr>
<tr>
<td>0.0014792907</td>
<td>2453</td>
<td>0.0014792903</td>
<td>2388</td>
<td>0.0014792907</td>
<td>2280</td>
<td>0.00147929</td>
<td>2279</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>2028</td>
<td>0.0014792903</td>
<td>1983</td>
<td>0.0014792903</td>
<td>2379</td>
<td>0.0014792903</td>
<td>1968</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>2018</td>
<td>0.0014792903</td>
<td>1996</td>
<td>0.0014792903</td>
<td>1985</td>
<td>0.0014792903</td>
<td>1978</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>2010</td>
<td>0.0014792903</td>
<td>1990</td>
<td>0.0014792903</td>
<td>1978</td>
<td>0.0014792907</td>
<td>1946</td>
</tr>
<tr>
<td>0.0014792896</td>
<td>1981</td>
<td>0.0014792903</td>
<td>2034</td>
<td>0.0014792903</td>
<td>1995</td>
<td>0.0014792903</td>
<td>1980</td>
</tr>
<tr>
<td>0.0014792893</td>
<td>1967</td>
<td>0.0014792903</td>
<td>1980</td>
<td>0.0014792907</td>
<td>1987</td>
<td>0.0014792903</td>
<td>1942</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>1938</td>
<td>0.0014792903</td>
<td>2005</td>
<td>0.0014792903</td>
<td>1962</td>
<td>0.0014792903</td>
<td>1989</td>
</tr>
<tr>
<td>0.0014792896</td>
<td>2324</td>
<td>0.0014792903</td>
<td>2103</td>
<td>0.0014792907</td>
<td>2497</td>
<td>0.0014792903</td>
<td>1985</td>
</tr>
</tbody>
</table>

Table 4: Table illustrating the scores and add-on execution time for 100 benign web pages, in four columns (i.e., 25 samples per column). Benign webpages are generated with 5500 lines of dead code.
<table>
<thead>
<tr>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00036982243</td>
<td>8233</td>
<td>0.00036982243</td>
<td>8232</td>
<td>0.0003698225</td>
<td>8499</td>
<td>0.00036982258</td>
<td>7472</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>7442</td>
<td>0.0014239947</td>
<td>8681</td>
<td>0.0003698225</td>
<td>7973</td>
<td>0.00036982226</td>
<td>8276</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>7297</td>
<td>0.0003698225</td>
<td>6720</td>
<td>0.0003698224</td>
<td>8310</td>
<td>0.00036982235</td>
<td>8916</td>
</tr>
<tr>
<td>0.0014239941</td>
<td>7827</td>
<td>0.0003698225</td>
<td>7818</td>
<td>0.00036982285</td>
<td>7843</td>
<td>0.00036982243</td>
<td>7925</td>
</tr>
<tr>
<td>0.00036982258</td>
<td>7850</td>
<td>0.00036982258</td>
<td>8466</td>
<td>0.00036982258</td>
<td>7476</td>
<td>0.00036982258</td>
<td>7405</td>
</tr>
<tr>
<td>0.00036982247</td>
<td>8132</td>
<td>0.00036982234</td>
<td>7717</td>
<td>0.00036982234</td>
<td>7933</td>
<td>0.0003698224</td>
<td>8014</td>
</tr>
<tr>
<td>0.00036982258</td>
<td>8478</td>
<td>0.0003698225</td>
<td>9749</td>
<td>0.00036982258</td>
<td>7844</td>
<td>0.00036982258</td>
<td>8181</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>8910</td>
<td>0.00036982234</td>
<td>7992</td>
<td>0.00036982234</td>
<td>9089</td>
<td>0.00036982258</td>
<td>7358</td>
</tr>
<tr>
<td>0.0003698224</td>
<td>7029</td>
<td>0.0003698225</td>
<td>7495</td>
<td>0.00036982243</td>
<td>7412</td>
<td>0.00036982266</td>
<td>7505</td>
</tr>
<tr>
<td>0.00036982234</td>
<td>8039</td>
<td>0.00036982258</td>
<td>7474</td>
<td>0.0003698225</td>
<td>9174</td>
<td>0.00036982275</td>
<td>8958</td>
</tr>
<tr>
<td>0.00036982234</td>
<td>7937</td>
<td>0.00036982243</td>
<td>7743</td>
<td>0.0003698225</td>
<td>7246</td>
<td>0.00036982258</td>
<td>8510</td>
</tr>
<tr>
<td>0.00036982258</td>
<td>8008</td>
<td>0.00036982266</td>
<td>7970</td>
<td>0.0003698225</td>
<td>7440</td>
<td>0.00036982258</td>
<td>7740</td>
</tr>
<tr>
<td>0.00036982243</td>
<td>8012</td>
<td>0.00036982266</td>
<td>8106</td>
<td>0.00036982258</td>
<td>8472</td>
<td>0.00036982258</td>
<td>8390</td>
</tr>
<tr>
<td>0.00036982257</td>
<td>7884</td>
<td>0.00036982258</td>
<td>7892</td>
<td>0.0003698224</td>
<td>7832</td>
<td>0.00036982258</td>
<td>8352</td>
</tr>
<tr>
<td>0.00036982241</td>
<td>8593</td>
<td>0.0003698225</td>
<td>7070</td>
<td>0.0003698225</td>
<td>6720</td>
<td>0.00036982243</td>
<td>8671</td>
</tr>
<tr>
<td>0.00036982249</td>
<td>8147</td>
<td>0.0003698225</td>
<td>7188</td>
<td>0.0003698225</td>
<td>8236</td>
<td>0.0003698224</td>
<td>8519</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>7143</td>
<td>0.00036982232</td>
<td>8449</td>
<td>0.00036982258</td>
<td>8064</td>
<td>0.00036982258</td>
<td>6830</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>8747</td>
<td>0.00036982266</td>
<td>7731</td>
<td>0.0003698225</td>
<td>8323</td>
<td>0.0003698224</td>
<td>8421</td>
</tr>
<tr>
<td>0.00036982258</td>
<td>7266</td>
<td>0.00036982258</td>
<td>8379</td>
<td>0.0014239943</td>
<td>7039</td>
<td>0.00036982243</td>
<td>8282</td>
</tr>
<tr>
<td>0.00036982258</td>
<td>7001</td>
<td>0.00036982243</td>
<td>9518</td>
<td>0.00036982251</td>
<td>7213</td>
<td>0.00036982249</td>
<td>7813</td>
</tr>
<tr>
<td>0.00036982249</td>
<td>6871</td>
<td>0.00036982249</td>
<td>9287</td>
<td>0.00036982243</td>
<td>9476</td>
<td>0.00036982252</td>
<td>8132</td>
</tr>
<tr>
<td>0.00036982243</td>
<td>7892</td>
<td>0.00036982252</td>
<td>8936</td>
<td>0.00036982266</td>
<td>8468</td>
<td>0.00036982243</td>
<td>7930</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>7989</td>
<td>0.0003698224</td>
<td>8250</td>
<td>0.00036982243</td>
<td>8304</td>
<td>0.00036982257</td>
<td>7418</td>
</tr>
<tr>
<td>0.00036982234</td>
<td>7814</td>
<td>0.0003698224</td>
<td>8658</td>
<td>0.00036982258</td>
<td>7766</td>
<td>0.00036982258</td>
<td>9372</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>8370</td>
<td>0.00036982258</td>
<td>8248</td>
<td>0.00036982258</td>
<td>7644</td>
<td>0.00036982266</td>
<td>8933</td>
</tr>
</tbody>
</table>

Table 5: Table illustrating the scores and add-on execution time for 100 malware web pages, in four columns (i.e., 25 samples per column). Malware webpages are morphed with 5500 lines of dead code.
### Table 6

<table>
<thead>
<tr>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0014792903</td>
<td>3241</td>
<td>0.001479291</td>
<td>3301</td>
<td>0.0014792903</td>
<td>3274</td>
<td>0.0014792903</td>
<td>3484</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>3180</td>
<td>0.001479292</td>
<td>3162</td>
<td>0.001479292</td>
<td>3355</td>
<td>0.0014792896</td>
<td>3342</td>
</tr>
<tr>
<td>0.00147929</td>
<td>3172</td>
<td>0.0014792896</td>
<td>3134</td>
<td>0.001479292</td>
<td>3345</td>
<td>0.0014792903</td>
<td>3415</td>
</tr>
<tr>
<td>0.0014792893</td>
<td>3291</td>
<td>0.0014792903</td>
<td>3150</td>
<td>0.001479292</td>
<td>3185</td>
<td>0.0014792896</td>
<td>3409</td>
</tr>
<tr>
<td>0.001479291</td>
<td>3161</td>
<td>0.0014792896</td>
<td>3545</td>
<td>0.0014792907</td>
<td>3184</td>
<td>0.0014792907</td>
<td>3536</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>3178</td>
<td>0.001479292</td>
<td>3104</td>
<td>0.001479292</td>
<td>3445</td>
<td>0.001479292</td>
<td>3540</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>3162</td>
<td>0.001479292</td>
<td>3107</td>
<td>0.001479292</td>
<td>3448</td>
<td>0.001479292</td>
<td>3786</td>
</tr>
<tr>
<td>0.00147929</td>
<td>3575</td>
<td>0.0014792907</td>
<td>3192</td>
<td>0.0014792907</td>
<td>3361</td>
<td>0.0014792907</td>
<td>3696</td>
</tr>
<tr>
<td>0.00147929</td>
<td>3563</td>
<td>0.0014792896</td>
<td>3131</td>
<td>0.001479292</td>
<td>3181</td>
<td>0.0014792907</td>
<td>3513</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>3271</td>
<td>0.0014792903</td>
<td>3216</td>
<td>0.0014792903</td>
<td>3383</td>
<td>0.0014792903</td>
<td>3405</td>
</tr>
<tr>
<td>0.0014792907</td>
<td>3188</td>
<td>0.0014792889</td>
<td>3320</td>
<td>0.0014792903</td>
<td>4725</td>
<td>0.0014792907</td>
<td>3608</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>3122</td>
<td>0.0014792896</td>
<td>3137</td>
<td>0.0014792903</td>
<td>3602</td>
<td>0.001479292</td>
<td>3275</td>
</tr>
<tr>
<td>0.00147929</td>
<td>3467</td>
<td>0.0014792903</td>
<td>3225</td>
<td>0.0014792907</td>
<td>3217</td>
<td>0.0014792907</td>
<td>3439</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>3197</td>
<td>0.0014792903</td>
<td>3328</td>
<td>0.001479292</td>
<td>3464</td>
<td>0.001479292</td>
<td>3700</td>
</tr>
<tr>
<td>0.0014792907</td>
<td>3149</td>
<td>0.0014792903</td>
<td>3329</td>
<td>0.001479292</td>
<td>3159</td>
<td>0.0014792907</td>
<td>3230</td>
</tr>
<tr>
<td>0.0014792893</td>
<td>3139</td>
<td>0.001479292</td>
<td>3201</td>
<td>0.0014792903</td>
<td>4870</td>
<td>0.001479292</td>
<td>3221</td>
</tr>
<tr>
<td>0.0014792893</td>
<td>3100</td>
<td>0.0014792896</td>
<td>3171</td>
<td>0.0014792896</td>
<td>3244</td>
<td>0.0014792907</td>
<td>3199</td>
</tr>
<tr>
<td>0.00147929</td>
<td>3206</td>
<td>0.0014792903</td>
<td>4664</td>
<td>0.001479292</td>
<td>3292</td>
<td>0.0014792907</td>
<td>3239</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>3139</td>
<td>0.001479292</td>
<td>3105</td>
<td>0.001479292</td>
<td>3417</td>
<td>0.001479292</td>
<td>3133</td>
</tr>
<tr>
<td>0.0014792896</td>
<td>3159</td>
<td>0.0014792893</td>
<td>3177</td>
<td>0.0014792903</td>
<td>3341</td>
<td>0.0014792903</td>
<td>3300</td>
</tr>
<tr>
<td>0.001479291</td>
<td>3102</td>
<td>0.001479292</td>
<td>3467</td>
<td>0.001479292</td>
<td>4321</td>
<td>0.0014792903</td>
<td>3420</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>3267</td>
<td>0.0014792903</td>
<td>3426</td>
<td>0.001479292</td>
<td>3587</td>
<td>0.0014792896</td>
<td>5329</td>
</tr>
<tr>
<td>0.00147929</td>
<td>3359</td>
<td>0.001479292</td>
<td>3421</td>
<td>0.0014792893</td>
<td>3333</td>
<td>0.0014792907</td>
<td>3614</td>
</tr>
<tr>
<td>0.0014792903</td>
<td>3412</td>
<td>0.001479292</td>
<td>3400</td>
<td>0.001479292</td>
<td>3289</td>
<td>0.0014792903</td>
<td>3450</td>
</tr>
</tbody>
</table>

Table 6: Table illustrating the scores and add-on execution time for 100 benign web pages, in four columns (i.e., 25 samples per column). Benign webpages are generated with 15000 lines of dead code.
<table>
<thead>
<tr>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
<th>Score</th>
<th>Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00036982258</td>
<td>9192</td>
<td>0.0003698224</td>
<td>9179</td>
<td>0.0003698225</td>
<td>9225</td>
<td>0.00036982266</td>
<td>8274</td>
</tr>
<tr>
<td>0.00036982258</td>
<td>9105</td>
<td>0.0003698225</td>
<td>7719</td>
<td>0.0003698225</td>
<td>9240</td>
<td>0.00036982258</td>
<td>9132</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>9104</td>
<td>0.00036982258</td>
<td>8207</td>
<td>0.0003698225</td>
<td>9458</td>
<td>0.00036982263</td>
<td>8176</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>9303</td>
<td>0.00036982243</td>
<td>9510</td>
<td>0.00036982266</td>
<td>8810</td>
<td>0.0003698225</td>
<td>8588</td>
</tr>
<tr>
<td>0.00036982258</td>
<td>8568</td>
<td>0.00036982258</td>
<td>9779</td>
<td>0.00036982243</td>
<td>8545</td>
<td>0.0003698224</td>
<td>8185</td>
</tr>
<tr>
<td>0.00036982241</td>
<td>9942</td>
<td>0.0003698225</td>
<td>9537</td>
<td>0.0014239943</td>
<td>8158</td>
<td>0.0003698224</td>
<td>8915</td>
</tr>
<tr>
<td>0.00036982266</td>
<td>9502</td>
<td>0.0003698226</td>
<td>9766</td>
<td>0.0003698225</td>
<td>8540</td>
<td>0.00036982266</td>
<td>9395</td>
</tr>
<tr>
<td>0.0003698224</td>
<td>9791</td>
<td>0.0003698225</td>
<td>8802</td>
<td>0.0003698225</td>
<td>9080</td>
<td>0.00036982258</td>
<td>8165</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>7951</td>
<td>0.00036982243</td>
<td>10122</td>
<td>0.0003698225</td>
<td>9233</td>
<td>0.00036982234</td>
<td>8113</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>8565</td>
<td>0.0003698225</td>
<td>9180</td>
<td>0.0003698224</td>
<td>8556</td>
<td>0.00036982258</td>
<td>9760</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>8059</td>
<td>0.00036982234</td>
<td>8254</td>
<td>0.0003698224</td>
<td>9702</td>
<td>0.00036982258</td>
<td>9330</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>8889</td>
<td>0.00036982258</td>
<td>8146</td>
<td>0.00036982258</td>
<td>9082</td>
<td>0.00036982258</td>
<td>8756</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>8884</td>
<td>0.00036982258</td>
<td>7851</td>
<td>0.0003698225</td>
<td>9281</td>
<td>0.00036982258</td>
<td>9336</td>
</tr>
<tr>
<td>0.00036982235</td>
<td>9126</td>
<td>0.00036982258</td>
<td>8672</td>
<td>0.00036982258</td>
<td>8465</td>
<td>0.00036982258</td>
<td>8828</td>
</tr>
<tr>
<td>0.00036982251</td>
<td>9631</td>
<td>0.00036982258</td>
<td>7948</td>
<td>0.00036982258</td>
<td>9770</td>
<td>0.00036982258</td>
<td>9277</td>
</tr>
<tr>
<td>0.0003698226</td>
<td>8128</td>
<td>0.00036982266</td>
<td>8689</td>
<td>0.00036982258</td>
<td>9211</td>
<td>0.00036982258</td>
<td>9478</td>
</tr>
<tr>
<td>0.00036982243</td>
<td>7919</td>
<td>0.00036982258</td>
<td>8264</td>
<td>0.00036982258</td>
<td>9809</td>
<td>0.00036982258</td>
<td>7847</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>9433</td>
<td>0.00036982258</td>
<td>8562</td>
<td>0.00036982258</td>
<td>9109</td>
<td>0.00036982258</td>
<td>9312</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>8202</td>
<td>0.00036982258</td>
<td>8119</td>
<td>0.001423994</td>
<td>9592</td>
<td>0.00036982258</td>
<td>8933</td>
</tr>
<tr>
<td>0.00036982243</td>
<td>8400</td>
<td>0.00036982266</td>
<td>8845</td>
<td>0.0014239942</td>
<td>1070</td>
<td>0.00036982258</td>
<td>9612</td>
</tr>
<tr>
<td>0.00036982252</td>
<td>9783</td>
<td>0.00036982266</td>
<td>8845</td>
<td>0.00036982266</td>
<td>9750</td>
<td>0.00036982258</td>
<td>8495</td>
</tr>
<tr>
<td>0.00036982258</td>
<td>8811</td>
<td>0.00036982266</td>
<td>8845</td>
<td>0.00036982243</td>
<td>9469</td>
<td>0.00036982258</td>
<td>8855</td>
</tr>
<tr>
<td>0.00036982275</td>
<td>10339</td>
<td>0.00036982258</td>
<td>9078</td>
<td>0.0003698224</td>
<td>8303</td>
<td>0.00036982258</td>
<td>7895</td>
</tr>
<tr>
<td>0.0003698225</td>
<td>8925</td>
<td>0.00036982258</td>
<td>9246</td>
<td>0.00036982258</td>
<td>8821</td>
<td>0.00036982258</td>
<td>10477</td>
</tr>
<tr>
<td>0.00036982243</td>
<td>9367</td>
<td>0.00036982258</td>
<td>8687</td>
<td>0.00036982258</td>
<td>8830</td>
<td>0.00036982258</td>
<td>8510</td>
</tr>
</tbody>
</table>

Table 7: Table illustrating the scores and add-on execution time for 100 malware web pages, in four columns (i.e., 25 samples per column). Malware web pages are morphed with 15000 lines of dead code.
Table 8: Table illustrating the max and min scores for all the sample files, after comparing the scores from the tables 2, 3, 4, 5, 6, and 7.

<table>
<thead>
<tr>
<th>Category</th>
<th>Min Score</th>
<th>Max Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>291 malware samples</td>
<td>0.000369822260</td>
<td>0.000369822750</td>
</tr>
<tr>
<td>9 exceptional malware samples</td>
<td>0.001423994000</td>
<td>0.001423994700</td>
</tr>
<tr>
<td>300 benign samples</td>
<td>0.001479288900</td>
<td>0.001479291400</td>
</tr>
</tbody>
</table>
Table 9: Table contains scores and add-on execution time details for popular web pages.

<table>
<thead>
<tr>
<th>Web page</th>
<th>Score</th>
<th>Time (Milli seconds)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://www.google.com/">https://www.google.com/</a></td>
<td>0.025195263</td>
<td>2189</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="https://www.facebook.com/">https://www.facebook.com/</a></td>
<td>0.08652405</td>
<td>3190</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.baidu.com/">http://www.baidu.com/</a></td>
<td>0.051652893</td>
<td>4442</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.twitter.com/">http://www.twitter.com/</a></td>
<td>0.8132002</td>
<td>4076</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.taobao.com">http://www.taobao.com</a></td>
<td>0.29001402</td>
<td>3401</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.qq.com/">http://www.qq.com/</a></td>
<td>0.014076417</td>
<td>3177</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="https://www.linkedin.com">https://www.linkedin.com</a></td>
<td>0.0916255</td>
<td>4309</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="https://live.com">https://live.com</a></td>
<td>0.30142236</td>
<td>1379</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.sina.com.cn/">http://www.sina.com.cn/</a></td>
<td>0.04421566</td>
<td>2513</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://us.weibo.com/gb">http://us.weibo.com/gb</a></td>
<td>0.210642001</td>
<td>2921</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.hao123.com/">http://www.hao123.com/</a></td>
<td>0.046390533</td>
<td>5314</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.bing.com/">http://www.bing.com/</a></td>
<td>0.30142236</td>
<td>1323</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.apple.com/">http://www.apple.com/</a></td>
<td>0.0625</td>
<td>5505</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.aliexpress.com/">http://www.aliexpress.com/</a></td>
<td>0.06497499</td>
<td>1794</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.imdb.com/">http://www.imdb.com/</a></td>
<td>0.041259766</td>
<td>8022</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.alibaba.com/">http://www.alibaba.com/</a></td>
<td>0.0047562416</td>
<td>3206</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.ask.com/">http://www.ask.com/</a></td>
<td>0.051652893</td>
<td>4611</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="https://www.netflix.com/">https://www.netflix.com/</a></td>
<td>0.0625</td>
<td>10046</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.naver.com/">http://www.naver.com/</a></td>
<td>0.30142236</td>
<td>2504</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://diply.com/">http://diply.com/</a></td>
<td>0.05702829</td>
<td>8612</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="https://mail.google.com/">https://mail.google.com/</a></td>
<td>0.30142236</td>
<td>1713</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.youku.com/">http://www.youku.com/</a></td>
<td>0.051652893</td>
<td>4044</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.flipkart.com/">http://www.flipkart.com/</a></td>
<td>0.018838914</td>
<td>1604</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="https://www.amazon.com/">https://www.amazon.com/</a></td>
<td>0.100754</td>
<td>2839</td>
<td>Benign page</td>
</tr>
<tr>
<td><a href="http://www.wikipedia.org/">http://www.wikipedia.org/</a></td>
<td>0.094681033</td>
<td>1914</td>
<td>Benign page</td>
</tr>
</tbody>
</table>
Table 10: Splitting Transcriptase into several files

<table>
<thead>
<tr>
<th>Count</th>
<th>Min</th>
<th>Max</th>
<th>Count</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.70E-04</td>
<td>3.70E-04</td>
<td>20</td>
<td>1.68E-18</td>
<td>0.018838914</td>
</tr>
<tr>
<td>2</td>
<td>1.28E-18</td>
<td>1.16E-17</td>
<td>21</td>
<td>1.81E-17</td>
<td>0.018838914</td>
</tr>
<tr>
<td>3</td>
<td>1.77E-19</td>
<td>1.39E-17</td>
<td>22</td>
<td>1.69E-19</td>
<td>0.018838914</td>
</tr>
<tr>
<td>4</td>
<td>6.17E-19</td>
<td>1.75E-17</td>
<td>23</td>
<td>9.64E-20</td>
<td>0.018838914</td>
</tr>
<tr>
<td>5</td>
<td>4.41E-20</td>
<td>3.84E-04</td>
<td>24</td>
<td>7.37E-19</td>
<td>0.018838914</td>
</tr>
<tr>
<td>6</td>
<td>5.36E-18</td>
<td>0.00153787</td>
<td>26</td>
<td>1.61E-19</td>
<td>0.018838914</td>
</tr>
<tr>
<td>7</td>
<td>4.57E-19</td>
<td>3.84E-04</td>
<td>27</td>
<td>2.27E-17</td>
<td>0.018838914</td>
</tr>
<tr>
<td>8</td>
<td>2.19E-18</td>
<td>0.00153787</td>
<td>29</td>
<td>9.72E-19</td>
<td>0.018838914</td>
</tr>
<tr>
<td>9</td>
<td>1.86E-17</td>
<td>0.00153787</td>
<td>31</td>
<td>3.85E-18</td>
<td>0.018838914</td>
</tr>
<tr>
<td>10</td>
<td>1.69E-19</td>
<td>0.003460208</td>
<td>33</td>
<td>6.35E-18</td>
<td>0.018838914</td>
</tr>
<tr>
<td>11</td>
<td>1.73E-18</td>
<td>0.00615148</td>
<td>35</td>
<td>4.33E-18</td>
<td>0.018838914</td>
</tr>
<tr>
<td>12</td>
<td>3.91E-18</td>
<td>0.003460208</td>
<td>38</td>
<td>4.37E-18</td>
<td>0.018838914</td>
</tr>
<tr>
<td>13</td>
<td>5.64E-20</td>
<td>0.00615148</td>
<td>42</td>
<td>8.12E-18</td>
<td>0.018838914</td>
</tr>
<tr>
<td>14</td>
<td>5.47E-19</td>
<td>0.003460208</td>
<td>46</td>
<td>4.41E-20</td>
<td>0.018838914</td>
</tr>
<tr>
<td>15</td>
<td>1.88E-18</td>
<td>0.00615148</td>
<td>51</td>
<td>5.21E-18</td>
<td>0.018838914</td>
</tr>
<tr>
<td>16</td>
<td>8.01E-20</td>
<td>0.009611688</td>
<td>57</td>
<td>1.96E-17</td>
<td>0.018838914</td>
</tr>
<tr>
<td>17</td>
<td>1.26E-18</td>
<td>0.013840835</td>
<td>65</td>
<td>3.84E-04</td>
<td>0.024605926</td>
</tr>
<tr>
<td>18</td>
<td>5.07E-19</td>
<td>0.009611691</td>
<td>76</td>
<td>3.84E-04</td>
<td>0.024605926</td>
</tr>
<tr>
<td>19</td>
<td>8.01E-19</td>
<td>0.018838914</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The aim of this research was to build a Mozilla add-on to detect metamorphic JavaScript malware embedded in a web page. For this purpose, I implemented an add-on using the Mozilla add-on SDK. Internally, the add-on uses the Rhino JavaScript engine to generate opcodes for the JavaScript content of a web page. As the opcode graph similarity technique performs better while classifying the files with similar opcode statistics, this technique was used in the add-on as a malware detection technique. Test results from chapter 5 show that a threshold score value 0.00145 is able to classify the Transcriptase malware family viruses and benign web pages properly even after adding significant amount of junk code. A similar approach can be used for all the different types of metamorphic malware.

Test results also show that execution time for the add-on is around 1 to 4 seconds for benign web pages and 6 to 11 seconds for malware web pages. Even though the execution overhead seems significant, the user is able to view the HTML and CSS content of the page properly during the add-on execution period. As discussed in Section 4.7.1, future enhancements for this thesis can include extending the add-on to use the cloud to increase the add-on performance. This enhancement requires efficient security measures, so that an intruder can’t eavesdrop/tamper with the information passed to and from cloud.

Future enhancements also include eliminating the burden of validating some external JavaScript files by storing their links as white lists. For instance, several web pages may have JavaScript code to display Google Ads, as Google is secured.
and regularly monitored, we can safely consider all the external Google Ads related JavaScript files as benign files. This approach may also involve some risk if any of the web page in the white list is attacked.

Different malware detection techniques can be added to the add-on to increase the detection rate. As discussed in Section 4.8, the add-on provides simple way to include other detection techniques.
LIST OF REFERENCES


APPENDIX

Code snippets

A.1 Python parser

Listing A.1: The parser code detects the valid start and end point of the JavaScript functions and properly splits the Transcriptase code.

```python
with open("transcriptase.js", "r") as ins:
    total_functions = 1000
    line = ins.read()
    k = sys.argv[1]
    required = (int)(total_functions/k)
    cnt, braces, rbraces, sbraces, brackets_match, func_ind = 0,0,0,0,0,
    skip, eachfun_done = False, False
    data, skip_char = '', ''
    function_start = True
    for c in line:
        if cnt == required and eachfun_done == True:
            cnt = 0
            eachfun_done = False
            #write data into a file
            data = ''
        if (c=='\textquotedblleft' or c="\textquoteleft") and skip==False:
            skip = True
            data = data+c
        skip_char = c
```
continue

if skip == True:
    data = data+c
    if skip_char == c:
        skip = False
        skip_char = ''
    continue

if c == '(':  
    rbraces+=1
    elif c == ')':
        rbraces-=1
    elif c == '[':
        sbraces+=1
    elif c == ']':
        sbraces-=1
    elif c == '{':
        if function_start==True:
            function_start=False
            braces+=1
        elif c == '}':
            braces-=1
        if braces == 0 and sbraces ==0 and rbraces == 0:
            if function_start==False:
                eachfun_done = True
else:
    data = data +c
    continue
if func[func_ind] == c:
    func_ind+=1
else:
    func_ind=0
if func_ind == 8:
    total_functions+=1
    cnt+=1
    function_start=True
    eachfun_done = False
    func_ind=0
    data+=c
if data != ":
    cnt = 0
    eachfun_done = False
    #write data into a file
    data=",
    data=","