Spring 2014

Hunting For Metamorphic JavaScript Malware

Mangesh Musale
San Jose State University

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

Part of the [Computer Sciences Commons](#)

Recommended Citation

Musale, Mangesh, "Hunting For Metamorphic JavaScript Malware" (2014). *Master's Projects*. 359.
DOI: <https://doi.org/10.31979/etd.ky84-3b5q>
https://scholarworks.sjsu.edu/etd_projects/359

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.

Hunting For Metamorphic JavaScript Malware

A Project

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

Mangesh Musale

May 2014

© 2014

Mangesh Musale

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Hunting For Metamorphic JavaScript Malware

by

Mangesh Musale

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2014

Dr. Mark Stamp Department of Computer Science

Dr. Thomas Austin Department of Computer Science

Dr. Sami Khuri Department of Computer Science

ABSTRACT

Hunting For Metamorphic JavaScript Malware

by Mangesh Musale

Internet plays a major role in the propagation of malware. A recent trend is the infection of machines through web pages, often due to malicious code inserted in JavaScript. From the malware writer's perspective, one potential advantage of JavaScript is that powerful code obfuscation techniques can be applied to evade detection. In this research, we analyze metamorphic JavaScript malware. We compare the effectiveness of several static detection strategies and we quantify the degree of morphing required to defeat each of these techniques.

ACKNOWLEDGMENTS

I am very thankful to my advisor Dr. Mark Stamp for his continuous guidance and support throughout this project and believing me. Also, I would like to thank the committee members Dr. Thomas Austin and Dr. Sami Khuri for monitoring the progress of the project and their valuable time.

TABLE OF CONTENTS

CHAPTER

1	Introduction	1
2	Background	3
2.1	Encrypted Malware	3
2.2	Polymorphic Malware	4
2.3	Metamorphic Malware	4
2.3.1	Subroutines Permutation	4
2.3.2	Instruction Reordering	5
2.3.3	Instruction Substitution	6
2.3.4	Register Swapping	6
2.3.5	Garbage Instruction Insertion	6
2.4	Detection Techniques	7
2.4.1	Signature Based Detection	7
2.4.2	Machine Learning Based Detection	7
2.4.3	Opcode Graph Based Detection	8
2.4.4	Simple Substitution Based Detection	8
3	Transcriptase	9
3.1	Instruction Permutation	10
3.2	Variable/Function Name Randomization	11
3.3	Meta Language Symbols	12
3.4	Variables/Functions Insertion	13

3.5	Miscellaneous Challenges	13
4	Rhino	14
4.1	Architecture	14
4.2	Modification	17
5	Statistical Malware Detection	20
5.1	Hidden Markov Model Method	20
5.1.1	Markov Chain	20
5.1.2	Hidden Markov Model	21
5.1.3	HMM Implementation	24
5.1.4	Log Likelihood Per Opcode	24
5.2	Opcode Graph Similarity	25
5.2.1	Opcode Graph	25
5.2.2	Similarity Calculation	27
5.2.3	Implementation	28
5.3	Simple Substitution Distance	28
5.3.1	Jakobsen Algorithm	29
5.3.2	Implementation	30
5.4	Singular Value Decomposition	34
5.4.1	Algorithm	34
5.4.2	Implementation	37
6	Experiments	38
6.1	Receiver Operating Characteristics	39
6.2	Hidden Markov Model	40

6.3	Opcode Graph Similarity Method	41
6.4	Simple Substitution Distance Method	42
6.5	Singular Value Decomposition Method	44
7	Enhancement	46
7.1	Hidden Markov Model	46
7.2	Opcode Graph Similarity Method	48
7.3	Simple Substitution Distance Method	48
7.4	Singular Value Decomposition Method	50
8	Conclusion and Future Work	52
 APPENDIX		
	ROC curves	57
A.1	Hidden Markov Model	57
A.2	Opcode Graph Similarity Method	62
A.3	Simple Substitution Distance Method	67
A.4	Singular Value Decomposition Method	71

LIST OF TABLES

1	Opcode Sequence	26
2	Digram Count Matrix	26
3	Digram Probability Matrix	27
4	Opcode Mapping (Putative Key)	31
5	Key Before Swapping	32
6	Key After Swapping	32
7	D Matrix before Swapping	32
8	D Matrix after Swapping	33
9	Machine Specifications	38
10	Source of Benign Files	39
11	Simple Substitution Method Scores	43
12	AUC for Hidden Markov Model	47
13	AUC for Opcode Graph Similarity Method	48
14	AUC for Simple Substitution Distance Method	49
15	AUC for Singular Value Decomposition Method	51

LIST OF FIGURES

1	Rhino Block Diagram	15
2	Abstract Syntax Tree	17
3	Sample Compilation Without Modification	19
4	Sample Compilation With Modification	19
5	Example of HMM	21
6	Markov Process	23
7	SVD Process	37
8	HMM Score Analysis (N=2)	40
9	ROC curve for HMM	41
10	Opcode Graph Similarity Analysis	42
11	ROC curve for Opcode Graph Similarity Method	42
12	Simple Substitution Method Scores Analysis	43
13	ROC for Simple Substitution Method	44
14	SVD Scatter Plot Analysis	45
15	ROC curve for SVD	45
16	Hidden Markov Model AUC Analysis	47
17	Opcode Graph Similarity Method AUC Analysis	49
18	Simple Substitution Distance Method AUC Analysis	50
19	Singular Value Decomposition AUC Analysis	50
A.20	ROC for 2500 Functions for HMM	57
A.21	ROC for 3000 Functions for HMM	57

A.22	ROC for 3500 Functions for HMM	57
A.23	ROC for 4000 Functions for HMM	57
A.24	ROC for 4500 Functions for HMM	58
A.25	ROC for 5000 Functions for HMM	58
A.26	ROC for 5500 Functions for HMM	58
A.27	ROC for 6000 Functions for HMM	58
A.28	ROC for 6500 Functions for HMM	58
A.29	ROC for 7000 Functions for HMM	58
A.30	ROC for 7500 Functions for HMM	59
A.31	ROC for 8000 Functions for HMM	59
A.32	ROC for 8500 Functions for HMM	59
A.33	ROC for 9000 Functions for HMM	59
A.34	ROC for 9500 Functions for HMM	59
A.35	ROC for 10000 Functions for HMM	59
A.36	ROC for 10500 Functions for HMM	60
A.37	ROC for 11000 Functions for HMM	60
A.38	ROC for 11500 Functions for HMM	60
A.39	ROC for 12000 Functions for HMM	60
A.40	ROC for 12500 Functions for HMM	60
A.41	ROC for 13000 Functions for HMM	60
A.42	ROC for 13500 Functions for HMM	61
A.43	ROC for 14000 Functions for HMM	61
A.44	ROC for 14500 Functions for HMM	61

A.45	ROC for 15000 Functions for HMM	61
A.46	ROC for 100 Functions for OGS	62
A.47	ROC for 200 Functions for OGS	62
A.48	ROC for 300 Functions for OGS	62
A.49	ROC for 400 Functions for OGS	62
A.50	ROC for 500 Functions for OGS	63
A.51	ROC for 600 Functions for OGS	63
A.52	ROC for 700 Functions for OGS	63
A.53	ROC for 800 Functions for OGS	63
A.54	ROC for 900 Functions for OGS	63
A.55	ROC for 1000 Functions for OGS	63
A.56	ROC for 2000 Functions for OGS	64
A.57	ROC for 3000 Functions for OGS	64
A.58	ROC for 4000 Functions for OGS	64
A.59	ROC for 5000 Functions for OGS	64
A.60	ROC for 6000 Functions for OGS	64
A.61	ROC for 7000 Functions for OGS	64
A.62	ROC for 8000 Functions for OGS	65
A.63	ROC for 9000 Functions for OGS	65
A.64	ROC for 10000 Functions for OGS	65
A.65	ROC for 11000 Functions for OGS	65
A.66	ROC for 12000 Functions for OGS	65
A.67	ROC for 13000 Functions for OGS	65

A.68	ROC for 14000 Functions for OGS	66
A.69	ROC for 15000 Functions for OGS	66
A.70	ROC for 5000 Functions for SSM	67
A.71	ROC for 5500 Functions for SSM	67
A.72	ROC for 6000 Functions for SSM	67
A.73	ROC for 6500 Functions for SSM	67
A.74	ROC for 7000 Functions for SSM	68
A.75	ROC for 7500 Functions for SSM	68
A.76	ROC for 8000 Functions for SSM	68
A.77	ROC for 8500 Functions for SSM	68
A.78	ROC for 9000 Functions for SSM	68
A.79	ROC for 9500 Functions for SSM	68
A.80	ROC for 10000 Functions for SSM	69
A.81	ROC for 10500 Functions for SSM	69
A.82	ROC for 11000 Functions for SSM	69
A.83	ROC for 11500 Functions for SSM	69
A.84	ROC for 12000 Functions for SSM	69
A.85	ROC for 12500 Functions for SSM	69
A.86	ROC for 13000 Functions for SSM	70
A.87	ROC for 13500 Functions for SSM	70
A.88	ROC for 14000 Functions for SSM	70
A.89	ROC for 14500 Functions for SSM	70
A.90	ROC for 15000 Functions for SSM	70

A.91	ROC for 15500 Functions for SSM	70
A.92	ROC for 100 Functions for SVD	71
A.93	ROC for 200 Functions for SVD	71
A.94	ROC for 300 Functions for SVD	71
A.95	ROC for 400 Functions for SVD	71
A.96	ROC for 500 Functions for SVD	72
A.97	ROC for 600 Functions for SVD	72
A.98	ROC for 700 Functions for SVD	72
A.99	ROC for 800 Functions for SVD	72
A.100	ROC for 900 Functions for SVD	72
A.101	ROC for 1000 Functions for SVD	72
A.102	ROC for 1100 Functions for SVD	73
A.103	ROC for 1200 Functions for SVD	73
A.104	ROC for 1300 Functions for SVD	73
A.105	ROC for 1400 Functions for SVD	73
A.106	ROC for 1500 Functions for SVD	73
A.107	ROC for 1600 Functions for SVD	73
A.108	ROC for 1700 Functions for SVD	74
A.109	ROC for 1800 Functions for SVD	74
A.110	ROC for 1900 Functions for SVD	74
A.111	ROC for 2000 Functions for SVD	74
A.112	ROC for 2100 Functions for SVD	74
A.113	ROC for 2200 Functions for SVD	74

A.114	ROC for 2300 Functions for SVD	75
A.115	ROC for 2400 Functions for SVD	75
A.116	ROC for 2300 Functions for SVD	75
A.117	ROC for 2400 Functions for SVD	75
A.118	ROC for 2300 Functions for SVD	75
A.119	ROC for 2400 Functions for SVD	75
A.120	ROC for 2500 Functions for SVD	76
A.121	ROC for 2600 Functions for SVD	76
A.122	ROC for 2700 Functions for SVD	76
A.123	ROC for 2800 Functions for SVD	76
A.124	ROC for 2900 Functions for SVD	76
A.125	ROC for 3000 Functions for SVD	76
A.126	ROC for 3500 Functions for SVD	77
A.127	ROC for 4000 Functions for SVD	77
A.128	ROC for 4500 Functions for SVD	77
A.129	ROC for 5000 Functions for SVD	77
A.130	ROC for 5500 Functions for SVD	77
A.131	ROC for 6000 Functions for SVD	77
A.132	ROC for 6500 Functions for SVD	78
A.133	ROC for 7000 Functions for SVD	78
A.134	ROC for 7500 Functions for SVD	78
A.135	ROC for 8000 Functions for SVD	78
A.136	ROC for 8500 Functions for SVD	78

A.137	ROC for 9000 Functions for SVD	78
A.138	ROC for 9500 Functions for SVD	79
A.139	ROC for 10000 Functions for SVD	79
A.140	ROC for 10500 Functions for SVD	79
A.141	ROC for 11000 Functions for SVD	79
A.142	ROC for 12500 Functions for SVD	79
A.143	ROC for 12000 Functions for SVD	79
A.144	ROC for 12500 Functions for SVD	80
A.145	ROC for 13000 Functions for SVD	80
A.146	ROC for 13500 Functions for SVD	80
A.147	ROC for 14000 Functions for SVD	80
A.148	ROC for 14500 Functions for SVD	80
A.149	ROC for 15000 Functions for SVD	80

CHAPTER 1

Introduction

Internet has evolved into the greatest medium of communication and data exchange that the world has ever known. Since its inception, the Internet has matured from just an academic experiment into an effective communication system. Since the late 1990s, it has become a vast interconnected source of information and services widely used for commercial and personal purposes. This evolution has led to the emergence of social networking, online banking and advertising, among various other commercial and non-commercial uses.

Transactions over the Internet often involve the transfer of sensitive data which attackers like to tap and exploit. For example, bank account information, medical records and passwords are routinely transferred over the network. Unfortunately, user's personal computer is a weak link in this system wherein personal computers typically run a large number of applications which are rarely managed properly. Single visit to a compromised web page is sufficient to infect a web browser, particularly the one that has not been updated with recent security patches.

Web browsers are often infected through malicious code inserted into legitimate JavaScript on a website [24]. When a user visits such a compromised website, these malicious JavaScript programs are automatically loaded with HTML code in the web browser. Execution of such malicious JavaScript can expose the personal data of the user. The majority of malicious JavaScript redirects the web browser to load more malicious content from a remote server. This can be achieved through several means, such as adding an HTML `iframe` element to a page [19]. While there are relatively

few ways to obfuscate HTML code [35], JavaScript makes it possible by providing a vast array of methods to hide the payload.

In this research, we analyze a metamorphic JavaScript malware that is designed to infect other JavaScript files in the same folder. We consider a variety of static detection techniques and quantify the effectiveness of each. We also measure the degree of morphing required to defeat each of these detection strategies.

This paper is organized as follows. In Chapter 2, we provide background information on malware with an emphasis on metamorphic malware. Chapter 3 outlines the specific metamorphic JavaScript malware that forms the basis for the research in this paper. Then in Chapter 4, we cover the Rhino JavaScript engine and discuss the modifications required for our experiments. Next, we discuss four different metamorphic detection techniques that we apply to the metamorphic JavaScript detection problem. Specifically, we consider Hidden Markov Model, an Opcode Graph similarity method, a distance measure inspired by Simple Substitution Cryptanalysis, and Singular Value Decomposition. These topics are covered in Chapter 5. Our experimental results for the original metamorphic JavaScript appear in Chapter 6. We consider an enhanced metamorphic JavaScript malware in Chapter 7 and provide the experimental results for this challenging case. Chapter 8 contains our conclusions and a consideration for future work.

CHAPTER 2

Background

Malware is a software program designed to do malicious activities on a user's computer with the intention of extracting information and exploiting resources without his consent. Such programs either stop normal execution of benign programs or they start executing along with benign code to do malicious activities. They destroy files by infecting them or by deleting them, stealing sensitive user information. According to the execution pattern and internal structure, malicious softwares are categorized as follows:

2.1 Encrypted Malware

Signature detection is one of the trivial methods for malware detection in which, a pattern of bytes or instruction sequence is extracted as a signature of that particular malware. Then, every file is compared against the extracted signatures to check whether a given file is a malware or a benign file. To overcome this detection method, malware writers came up with Encrypted Malware. In such a type of malware, the entire body of malware is encrypted by using one of the encryption techniques. Code to decrypt this encrypted malware is also attached to its body. As a result of this, signature based detection method is unable to find a pattern in the file and decryption code will decrypt the malware at the time of execution, which will not affect the normal execution of malware. Cascade was the first virus of this kind which appeared in 1987 [20].

2.2 Polymorphic Malware

As the decryption code is not encrypted in Encrypted Malware, antivirus writers were able to extract the pattern from decryption code. The need to hide decryption code gave birth to Polymorphic Malware. In Polymorphic Malware, code to decrypt malware is morphed after every infection. Morphing is done in such a way that the internal structure of code gets changed completely without any change in the functionality [20]. This makes signature extraction hard. First polymorphic virus named 1260 was developed by Washburn in 1990 [28].

2.3 Metamorphic Malware

Metamorphic Malware is an advanced version of Polymorphic Malware. In this malware, internal structure of the malware gets changed after every execution. But the overall functionality remains the same [16]. There are various methods by which this can be achieved. Let us discuss a few of them.

2.3.1 Subroutines Permutation

Function definition can be inserted between any two statements in the code. Using this feature, malware writers reorder the function definitions, keeping the order of function call unique. This will give structurally different but functionally same software. Suppose, we have n different functions in the code, then these functions can be arranged in $n!$ different ways. This gives $n!$ different morphed versions of the same code. Metamorphic malware such as win32/Ghost has used this type of subroutine permutation technique [37].

2.3.2 Instruction Reordering

In this method, independent instructions are reordered in a program scope. Here, independent instructions refer to those instructions whose execution does not depend on any previous instruction. This reordering will give us quite a large number of morphed versions of the same code [20]. If we have n independent instructions in the scope of a program, then we can generate $n!$ different morphed versions. Illustrated below is a simple example of addition of two numbers.

```
1: int a = 10;  
2: int b = 20;  
3: int c = a + b;
```

Here in statements 1 and 2, we declare two variables `a`, `b` and assign them constant values 10 and 20 respectively. Declaring a variable does not depend on any other statement. Therefore, in this example, statement 1 and 2 are independent instructions which can be arranged in any order. But, statement 3 depends on statement 1 and 2. Statement 3 adds two variables `a` and `b`, but to add them they must be declared and assigned some value. Therefore, statement 3 must appear after statement 1 and 2. Considering all these dependencies, the above code can be reordered to achieve the same functionality as shown below.

```
2: int b = 20;  
1: int a = 10;  
3: int c = a + b;
```

2.3.3 Instruction Substitution

In this method, morphed copies are generated by replacing an instruction or a group of instructions by a functionally equivalent instruction or group of instructions [32]. For example,

SUB EBX EBX

can be replaced as

AND EBX 0x0000

SUB instruction subtracts the content of register B from the register B itself i.e., the code wants to clear register B. Same effect can be achieved through bitwise multiplication of register B contents with zero.

2.3.4 Register Swapping

In this method, morphed copies are generated by changing registers used in the instructions. This is a very simple way of code obfuscation. Win95/Regs wap [29] malware is such a kind of malware. Consider an example where we want to fetch the value from stack and load it into one of the registers for the next operation. POP EAX can be replaced with POP EBX if EBX register is free at this moment. One of the drawbacks of this method is it does not alter the opcode sequence.

2.3.5 Garbage Instruction Insertion

In this method, morphed copies of malware are created by inserting instructions in software which will not affect execution [6]. Such ‘do-nothing’ instructions are used in creating morphed copies of malware such as Win95/ZPerm [29]. Examples of such instructions are NOP instruction or ADD EBX 0. Instructions like INC and DEC can be used in succession to increase the value in one of the registers and bring it back to

original.

2.4 Detection Techniques

There are several techniques to detect a malware. Different characteristics of malware are used in detection techniques such as statistical analysis on opcodes, pattern matching of bytes for a particular malware family. Some of the popular techniques are mentioned below:

2.4.1 Signature Based Detection

In this technique, content of malware is compared against the signature dictionary to check whether the given file is malware or benign. A dictionary for known malware signatures is maintained and updated by anti-virus vendors. A signature is an algorithm or hash (a number derived from a string of text) that uniquely identifies a specific virus. Depending on the type of scanner being used, it may be a static hash, which in its simplest form, is a calculated numerical value of a snippet of code unique to the virus.

2.4.2 Machine Learning Based Detection

Machine learning is a very widely used concept in computer science. Hidden Markov Model (HMM) is one of the most popular machine learning techniques used to detect malware [27]. In this technique, a HMM is trained against known malware opcode sequence. This trained model is used to score files to be analyzed. If the score is more than a predefined threshold, then it is fairly good to say that the given file belongs to a malware family.

2.4.3 Opcode Graph Based Detection

In this technique, a digram frequency graph is constructed from opcode sequence of known malware. Similar graph is constructed for the file to be analyzed. Score of this file is calculated by taking absolute difference between corresponding elements in the graph. Lesser the score, higher is the probability of the file being a malware [22]. We will discuss more about this technique in Section 5.2.

2.4.4 Simple Substitution Based Detection

This technique is based on the assumption that frequency distribution of opcodes hold true to a certain extent for every malware family. In this technique, a digram frequency graph is constructed from an opcode sequence of a known malware and the file to be analyzed. A frequency count is taken for each unique opcode in the known malware. Frequency distribution of opcodes from known malware samples is mapped with frequency distribution of opcodes from files which are to be tested. The score for each file is calculated by the same method used in Jakobsen's algorithm for Simple Substitution Cryptanalysis [13]. If the score for a file is less than a precalculated threshold, then it is reasonable to conclude that the file belongs to a malware family. Implementation details for this method are described in Section 5.3.

CHAPTER 3

Transcriptase

Transcriptase malware was originally designed by Sperl [25]. Transcriptase is a metamorphic malware implemented in JavaScript. Execution of this script infects all the JavaScript files in the folder where the malware script is placed. As a result of this infection, a morphed version of the malware script gets attached to benign JavaScript files in the folder. Whenever this infected JavaScript gets executed in any other folder, it infects other benign JavaScript files. For each infection, malware script generates a new morphed version. The malware script carries its code in a custom designed meta-language.

The morphing scheme used in this malware helps to evade signature based detection method. The meta-language code here is compiled using a compiler written in JavaScript. Compiler code is also a part of the malware body. Advantage of defining a custom meta-language is that the malware writer can add extra information required to create the morphed versions. The general syntax used for an instruction in this malware sample is shown below

(Identifier |Restrictions) meta-instruction

Identifier and Restrictions are used by the permutation function for code obfuscation. Identifier is the unique identification for a statement in the script and Restriction is a list of statements which must be executed before the given statement. Details of this scheme is described in Section 3.1. These meta-instructions are used to create actual instructions in the script. Compiler creates a new malicious JavaScript in three steps as shown below:

- Permutation and Variable/Function-Name Randomization
- Code Creation
- Variable/Function Insertion

3.1 Instruction Permutation

In this part of the code, compiler parses the meta language code, scope by scope and collects all the information about instructions and their corresponding restrictions. It analyzes the code in the given scope and starts permuting instructions considering their dependencies. For example, consider the following code

```
(100|)var a=1;
(200|)var b=0;
(300|)def c=1;
(400|200)c+1(b);
(500|400,100)c+n(b,a);
(600|500)xWScript.Echo(b);
```

Here, first number is the identifier for the instruction. This remains unique in the entire JavaScript. Numbers after ‘|’ indicate dependencies of the given instruction. Instruction with identifier 200 does not have any number after ‘|’, shows that it does not depend on any other instruction. Instruction 200 is `var b = 0`. This is declaration of variable and assigns it a value 0. To declare a variable and assign it some constant value does not have any dependency on previous instructions.

But if we look at instruction 400, `c + 1(b)`, to execute this instruction we need the value of variable `b`. Variable `b` is declared and assigned a value at instruction 200. Therefore, 200 appears in instruction 400 after ‘|’ to indicate dependency.

Permutation function reads this information and uses it to rearrange the instruction. As we observe, there is no number after ‘|’ in instructions 100, 200 and 300, it means that these instructions are independent of any other instructions in the script. So, they can be rearranged as

```
(300|)def c=1;
(100|)var a=1;
(200|)var b=0;
(400|200)c+1(b);
(500|400,100)c+n(b,a);
(600|500)xWScript.Echo(b);
```

Here, we changed the order of instructions as 300, 100, 200 without any effect on the functionality of the script. These three independent instructions can be arranged in 3! different ways. Due to the fast growth-rate of the permutation function, this technique is very useful in case of large number of instructions.

3.2 Variable/Function Name Randomization

The compiler searches for keywords like **var**, **def** and **function** in all instructions of the script. Then it gets the hard coded strings used to define them and replaces them with random strings. It then replaces all occurrences of these hard-coded strings in the current scope with the newly generated random string. For example, consider the following function definition.

```
function makeSquare (number){return number*number;}
var num=1;
def sqr=makeSqare(num);
```

Compiler searches for `var`, `def` and `function` keywords. Here, it will find `makeSquare`, `number`, `num` and `sqr` variable names. Then it will generate random strings and map each string with one variable name and replace it in the current scope. After that, it will search for the same variable name in upper scopes till the global scope is reached. After this step, the new form of code will be

```
function pkjuoledrbnx(qerdslds){return qerdslds* qerdslds;}
var hxedlkerd = 1;
def cadwkgd = pkjuoledrbnx(hxedlkerd);
```

3.3 Meta Language Symbols

After permutation of instructions and function/variable name randomization, the compiler parses the meta-language code and derives a valid JavaScript statement for every statement in meta-language. While deriving the actual instruction, several meta-language symbols are processed each having a special meaning. Compiler interprets them at runtime and generates appropriate javascript literals. For instance:

```
var number = #n1n#;
var str = #"Hello_World"#;
var exp = #xntruenx#;
```

Here, `#n...n#` is interpreted as the value at position ... is a numerical value and `#"..."#` is interpreted as the value at position ... is a string value. Furthermore, `#xn ... nx#` is interpreted as the value at ... is a JavaScript literal like `true` or `false`, where *n* can be any integer. At this step, each symbol is replaced with original JavaScript language literal.

3.4 Variables/Functions Insertion

During the compilation, many variables and functions are defined (either due to the meta-code itself, or due to obfuscation). They are not yet placed in the code, but are saved in special arrays. At the end of the code creation, they are placed in the code. Functions can be inserted between any two instructions in the global scope. Variables can be inserted between any two instructions in the current scope before they get used for the first time. This insertion takes a long time because the whole code has to be analyzed several times to find potential positions [17].

3.5 Miscellaneous Challenges

At the time of code creation, the compiler creates random functions. These functions perform arithmetic calculations and return integer values between 0 to 255. They are then used to represent ASCII values of characters to derive strings. This makes the code highly obfuscated.

Compiler creates an array of functions. Use of this array of functions in constructing strings makes the code highly unreadable. It is very hard to read and understand such a code in one parse [17].

CHAPTER 4

Rhino

In 1997, Netscape started working on a project to bring about a version of Netscape Navigator developed fully in Java. For this, they came up with a JavaScript engine which was also developed in Java. This JavaScript engine was named as the Rhino Project which is now maintained by Mozilla [21]. Rhino compiles JavaScript into Java classes and can operate in two modes—compiled mode and interpreted mode. In compiled mode, Rhino first compiles JavaScript and then converts it into Java bytecode. Then this bytecode can be run as a Java program. This improves the execution time of JavaScript.

4.1 Architecture

Rhino JavaScript Engine consists of four basic building blocks—Parser, Byte-Code Generator, Interpreter and JIT. The JavaScript source code is first fed to a Parser which parses the code and converts it into an Abstract Syntax Tree (AST). Then this AST is fed to the ByteCode Generator which generates the bytecode. This bytecode is used by the Rhino Interpreter which converts it into machine code with the help of JIT. Finally, this machine code gets executed [33]. Block diagram of Rhino JavaScript Engine can be visualized in Figure 1.

Parser: Parser takes JavaScript code as input and generates AST. Parser knows the meaning of every JavaScript literal and its functionality. First it tokenizes the JavaScript and then operates on each token to convert it into AST. Consider the following example of finding Greatest Common Divisor(GCD) by binary algorithm:

Figure 1: Rhino Block Diagram

```
function GCD(num1, num2)
{
    if (num2 > num1)
    {
        var temp = num1;
        num1 = num2;
        num2 = temp;
    }
    while (true) {
        num1 %= num2;
        if (num1 == 0) return num2;
        num2 %= num1;
        if (num2 == 0) return num1;
    }
}
```


Parser parses the code and generates an AST as shown in Figure 2. In this tree, each non-leaf node represents the operation to be performed like subtraction, comparison etc. and each leaf node represents the operands like num1 and num2. This tree can be viewed as one of the paths that the program will traverse.

ByteCode Generator: Once AST is formed, then the ByteCode Generator will convert this into bytecode. ByteCode Generator converts AST into bytecode, block by block. For example, it will pick a block of the tree which subtracts two variables and convert it into bytecode as shown below.

```
    iload_1
    iload_2
    isub 3
    istore_3
```

These bytecodes can be interpreted as—load the first variable stored at offset 1 and 2 into a register. Then subtract them and store the value at offset 3.

Interpreter: Interpreter takes the output generated by ByteCode Generator as input. Interpreter then with the help of JIT converts the bytecode into machine level code. This machine code then gets executed at runtime. Above generated bytecode will be converted as shown below.

```
MOV EAX 0xFF20
MOV EBX 0xFF24
SUB EAX EBX
MOV ECX EAX
```

The main challenge faced by Rhino JavaScript engine is that the JavaScript does not have classes and it is a dynamic language. Because of this nature of JavaScript, there are lot of complex optimization techniques that have to be used to squeeze out performance. Few of them are listed below:

Figure 2: Abstract Syntax Tree

- Starting time of JavaScript engine
- Generation of temporary classes
- Optimization of compiled code
- Garbage Collection
- Sandboxing the execution for security

4.2 Modification

Main purpose of this experiment is to compile JavaScript in machine level code which would extract opcodes from the compiled file to train Hidden Markov Model.

As Rhino is completely developed in Java, and being an open source software, its code is readily available and easily comprehensible.

There are certain limitations in Rhino which are purposely implemented at the time of its development. As the latency in compiling JavaScript directly affects the page load time, being a part of the web browser, Rhino compiles only small JavaScript files. There are different levels of optimizers which optimizes class files for faster execution. This optimization sometimes loses original statistical properties of JavaScript which are used in analysis.

To deal with this problem, we decided to modify Rhino in such a way that it can compile JavaScript of any length and disable optimization. As we are not interested in the class file, we decided to extract opcodes at compile time. This saved the time of decompilation of class file to extract opcode sequence. This also solved the problem of optimization, as we were extracting opcodes before it creates a class file. We tapped the opcode generated by opcode generator and redirected it to the standard output stream. This will print all the generated opcodes to a screen. The generated output can be collected in a file using redirect operation in Unix. Command to run Rhino:

```
java -classpath <path to js.jar> org.mozilla.javascript.tools.jsc.Main  
<Path to JavaScript file>
```

Figure 3 shows standard output of Rhino after compilation. After compiling JavaScript, we will find a new class file generated in the same folder. Modified Rhino generates a class file same as the class file without modification. But in addition to the class file, a list of opcodes is also generated. This list can be observed in Figure 4. We can write this output to some file using redirect operator ($>$).

A terminal window titled 'mangesh@mangesh-ThinkPad-T530: ~/Documents/cs297/rhino'. The prompt is 'mangesh@mangesh-ThinkPad-T530:~/Documents/cs297/rhino\$'. The user enters the command 'java -classpath js.jar org.mozilla.javascript.tools.jsc.Main ./test.js'. The prompt returns to 'mangesh@mangesh-ThinkPad-T530:~/Documents/cs297/rhino\$' without any output.

```
mangesh@mangesh-ThinkPad-T530:~/Documents/cs297/rhino$ java
-classpath js.jar org.mozilla.javascript.tools.jsc.Main ./test
.js
mangesh@mangesh-ThinkPad-T530:~/Documents/cs297/rhino$
```

Figure 3: Sample Compilation Without Modification

A terminal window titled 'mangesh@mangesh-ThinkPad-T530: ~/Documents/cs297/rhino'. The prompt is 'mangesh@mangesh-ThinkPad-T530:~/Documents/cs297/rhino\$'. The user enters the command 'java -classpath js.jar org.mozilla.javascript.tools.jsc.Main ./test.js'. The output is a list of instructions: 'aload_0', 'invokespecial', 'aload_0', 'iconst_0', 'putfield', 'return', 'new', 'dup', and 'invokespecial'.

```
mangesh@mangesh-ThinkPad-T530:~/Documents/cs297/rhino$ java
-classpath js.jar org.mozilla.javascript.tools.jsc.Main ./test
.js
aload_0
invokespecial
aload_0
iconst_0
putfield
return
new
dup
invokespecial
```

Figure 4: Sample Compilation With Modification

CHAPTER 5

Statistical Malware Detection

Signature based detection method is widely used for malware detection. But metamorphic malwares can easily evade the detection by altering their internal code by using one of the techniques described in Section 2.3. Using these methods, the internal structure can be changed but to retain original functionality, same instructions (instructions which are responsible for functionality) have to be used somewhere in code. This keeps the statistical distribution of instructions unique in all the morphed copies. Using these statistical properties, following malware detection techniques are designed.

5.1 Hidden Markov Model Method

Hidden Markov Model(HMM) is a pattern recognition technique which can be used to detect metamorphic malware. Recently, lot of research has been going on to enhance this technique [2, 3, 8, 30, 31]. Sridhara [26] describes a method to train HMM against opcode sequence extracted from the given malware family. This HMM model is used to score the files and classify whether a file belongs to the given malware family. If the score is greater than a predefined threshold, then it is concluded that the file belongs to a given malware family. Scores are measured in terms of Log Likelihood Per Opcode (LLPO).

5.1.1 Markov Chain

Consider a sequence of symbols. Each symbol is associated with a probability which governs the transition from one symbol to other. Transition from one symbol

Figure 5: Example of HMM

to another depends upon the probability of current symbol or in other words, not on any previous symbol in the sequence. Such sequence is called as the ‘Markov Chain’ of first order [18]. Similarly n^{th} order Markov Chain is defined as a sequence of symbols in which transition from one symbol to other depends on previous $n - 1$ symbols. In Hidden Markov Model terminology, each symbol is called as State.

Consider an example shown in Figure 5 [2]. In this example, we have two coins one of which is biased and the other is normal. For the biased coin, probability of getting a Heads after tossing it is 0.7 and getting a Tails is 0.3. On other hand for the normal coin, probability of getting Heads or Tails after tossing it is 0.5. We tossed a coin for 5 times which resulted in a sequence (H, T, H, H, T). H and T in this sequence represents Heads and Tails respectively. In this sequence, getting Heads or Tails depends on which coin we choose and each coin is associated with a probability of getting either a Heads or a Tails. Therefore, this sequence forms a Markov Chain. For this experiment, process of tossing coins is a Markov Process.

5.1.2 Hidden Markov Model

Hidden Markov Model(HMM) is a statistical model which represents a Markov Process whose states are unknown [27]. In [27] describes the notation used in HMM

as follows:

$$\begin{aligned}
T &= \text{length of the observation sequence} \\
N &= \text{number of states in the model} \\
M &= \text{number of observation symbols} \\
Q &= \{q_0, q_1, \dots, q_{N-1}\} = \text{distinct states of the Markov process} \\
V &= \{0, 1, \dots, M-1\} = \text{set of possible observations} \\
A &= \text{state transition probabilities} \\
B &= \text{observation probability matrix} \\
\pi &= \text{initial state distribution} \\
O &= (o_0, o_1, \dots, o_{T-1}) = \text{observation sequence.}
\end{aligned}$$

To get a better insight about HMM, consider the same example shown in Figure 5. Here, length of the observation sequence T is 5 as the coin is tossed for 5 times in this process. We have two different symbols in the observation sequence i.e. H and T. Therefore, value of M for this Markov Process is 2 with $V=\{H, T\}$. Transition matrix A for this process is

$$A = \begin{bmatrix} 0.95 & 0.05 \\ 0.20 & 0.80 \end{bmatrix}$$

where, the first row represents transition probabilities for the normal coin and the second row represents transition probabilities of the biased coin. Value at position (i, j) represents transition of state from i^{th} state to j^{th} state. Observation probability matrix B for this process is

$$A = \begin{bmatrix} 0.5 & 0.5 \\ 0.7 & 0.3 \end{bmatrix}$$

where, the first row represents probability of getting each observation for normal coin and the second row represents probability of getting each observation for the biased coin. Information in this matrix can be understood as—if we choose a biased coin, then probability of getting Heads is 0.7. We can pick any coin out of the given two coins. So initial state distribution will be

$$\pi = [0.5 \quad 0.5]$$

Figure 6: Markov Process

Matrices A, B, π are row-stochastic which means that each row in these matrices is a probability distribution. Here, we know the values of A, B, π and observation sequence but not the coin used for each tossing. This part of process is hidden. A model is defined using these known values as

$$\lambda = (A, B, \pi)$$

Figure 6 gives a graphical view of the Hidden Markov Process. The state and observation of HMM at any point of time t is represented by X_t and O_t respectively. Initial state X_0 and matrix A determines the hidden part of the Markov Process, as represented in the left half of Figure 6.

5.1.3 HMM Implementation

Application of HMM for metamorphic malware detection is explained in greater detail in [34]. HMM model is trained using opcodes, extracted from morphed copies of malware belonging to a family. This trained model represents statistical properties of the malware family. We can compute a score for any file to determine how similar the file is to the malware family using trained HMM model. Then we can compare this score with a predetermined threshold to classify the file.

A large number of morphed versions of given malware JavaScript are generated and compiled using JavaScript compiler to obtain Java bytecode [21]. Then the opcode is extracted from these Java bytecodes. All the opcodes in training dataset are concatenated and then HMM is trained on this concatenated list of opcodes to get statistical model of the given metamorphic JavaScript malware.

To detect whether the given file belongs to the same malware family or not, we used previously generated model to calculate LLPO. If the LLPO of the program is within a particular threshold, the file is classified as a virus.

5.1.4 Log Likelihood Per Opcode

Scoring observation sequences and training the HMM involves computation of large number of products of probability distribution of opcodes. As T increases, the resulting value of this product exponentially tends to become 0. This introduces a problem of underflow of results. To avoid this problem, forward and backward algorithms normalize the result of each iteration. This process is called scaling [27]. After application of scaling, $P(O|\lambda)$ will be calculated as

$$1 / \prod_{k=0}^{T-1} c_k$$

where c_k is the scaling factor at time k . However, this computation is also susceptible to underflow and to avoid that, we compute Log Likelihood as

$$\log[P|\lambda] = - \sum_{k=0}^{T-1} \log c_k$$

Log Likelihood is length dependent, as the sum of log transition probabilities and log observation probabilities will be higher for a longer sequence. As the sequence in the test set may be of different lengths compared to the sequence used to train the model, Log Likelihood will be divided by the number of opcodes in the sequence to obtain Log Likelihood per opcode which accounts for the length difference [34].

5.2 Opcode Graph Similarity

Anderson [1] describes a graph based method for malware detection. In this method, opcode sequence from given malware is extracted and a weighted directed graph is constructed using this sequence. Similar weighted graph is constructed for the file to be tested. Score for the file is calculated by taking Manhattan distance between these two graphs. Algorithm to calculate score is described in Section 5.2.2.

5.2.1 Opcode Graph

A weighted directed graph constructed from extracted opcode sequence is called as ‘Opcode Graph’. Each node of this graph is represented by a unique opcode from the opcode sequence. A directed edge is drawn from one opcode to another, if and only if, second opcode follows the first opcode in the sequence. Weight of the edge is determined by the number of occurrences of digram. Consider the opcode sequence shown in Table 1. This opcode sequence contains 30 opcodes. To construct a directed graph of opcodes, we need the count of digrams. In this method, we will represent this graph as an adjacency matrix as shown in Table 2.

Table 1: Opcode Sequence

Sequence No	Opcode	Sequence No	Opcode
1	MOV	16	AND
2	SUB	17	PUSH
3	TEST	18	PUSH
4	CALL	19	MOV
5	SUB	20	CALL
6	ADD	21	CALL
7	PUSH	22	MOV
8	PUSH	23	SUB
9	AND	24	MOV
10	AND	25	SUB
11	AND	26	MOV
12	CALL	27	SUB
13	LEA	28	MOV
14	LEA	29	SUB
15	ADD	30	SUB

Table 2: Digram Count Matrix

	MOV	SUB	TEST	CALL	ADD	PUSH	AND	LEA
MOV	0	5	0	1	0	0	0	0
SUB	3	1	1	0	1	0	0	0
TEST	0	0	0	1	0	0	0	0
CALL	1	1	0	1	0	0	0	1
ADD	0	0	0	1	1	1	0	0
PUSH	1	0	0	0	0	3	1	1
AND	0	0	0	0	1	0	0	0
LEA	0	0	0	0	1	0	0	1

If you observe the row corresponding to **SUB** and the column corresponding to **MOV**, then you will get a count 3 for digram (**SUB**, **MOV**). If you observe opcode sequence in Table 1, you will notice that **MOV** instruction appeared 3 times after **SUB** instruction at line numbers 24, 26 and 28. Similarly, the count for each digram is tabulated in Table 2.

Using Table 2, we will convert counts into probabilities by dividing each row

entry by the sum of that row. The resulting values will indicate the probability of occurrence of a particular opcode, after the selected opcode. For example, probability of occurrence of TEST instruction after SUB instruction will be 1 divided by 7 (sum of the entries corresponding to row SUB). After converting counts into probabilities, Table 2 will now look like Table 3.

Table 3: Digram Probability Matrix

	MOV	SUB	TEST	CALL	ADD	PUSH	AND	LEA
MOV	0	5/6	0	1/6	0	0	0	0
SUB	3/7	1/7	1/7	0	1/7	0	0	0
TEST	0	0	0	1/1	0	0	0	0
CALL	1/4	1/4	0	1/4	0	0	0	1/4
ADD	0	0	0	1/3	1/3	1/3	0	0
PUSH	1/6	0	0	0	0	3/6	1/6	1/6
AND	0	0	0	0	1/1	0	0	0
LEA	0	0	0	0	1/2	0	0	1/2

5.2.2 Similarity Calculation

Probability matrices for files to be compared are created by the procedure described above. Similarity between two files is calculated by taking the Manhattan distance between two probability matrices. Suppose A is the probability matrix for file 1 and each element in A is represented as $a_{i,j}$ where i and j represents i^{th} row and j^{th} column. Similarly for file 2, B is the probability matrix and each element is represented as $b_{i,j}$. Similarity is calculated by

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

where N is the number of unique opcodes under consideration.

5.2.3 Implementation

To use this similarity score for metamorphic malware detection, we need to set a suitable threshold. To set the threshold value, we followed the steps below:

1. Calculate opcode graph for known metamorphic malware samples.
2. Calculate opcode graph for morphed versions of malware samples.
3. Calculate opcode graph for benign files.
4. Calculate score for morphed versions vs known metamorphic malware samples.
5. Calculate score for benign files vs known metamorphic malware samples.
6. Set threshold based on step 4 and step 5.

For our implementation, we chose the highest score from step 4 and the lowest score from step 5. Average of these two values is considered as the threshold. If score of any file is less than this threshold, it is classified as a malware.

5.3 Simple Substitution Distance

Simple Substitution Distance method is a hill climbing technique based on Jakobsen's algorithm for simple substitution cryptanalysis [13]. In this method, it is assumed that frequency order of opcodes hold true to a certain extent for every malware family. This allows us to calculate similarity between any two files using Jakobsen's algorithm. In Jakobsen's algorithm, the score gives the degree to which the putative plaintext matches the plaintext language statistics. The score given by Simple Substitution Distance method can be viewed as a measure of the distance between the opcode sequence of a given file and the opcode statistics of a metamorphic family [23].

5.3.1 Jakobsen Algorithm

Basic assumption in Jakobsen's algorithm is that the plain text is in English language and Ciphertext contains 26 different symbols. Each symbol maps to one of the 26 different letters in English. Algorithm starts with calculating the frequency of the symbols in the ciphertext and arranging them in the decreasing order of their occurrence frequency. Putative key is then calculated by mapping these symbols from the ciphertext to the letters in English arranged in the standard frequency order. This key is altered in each iteration after looking at the obtained plain text. Suppose key 'K' is initial key, then it can be represented as

$$K = k_1, k_2, k_3, \dots, k_{26}$$

where each k_n is mapping of one symbol from cipher text on some English letter.

In each iteration algorithm modifies key 'K' slightly. This new key is used to decrypt ciphertext and then the newly generated putative plain text is checked. If putative plain text is closer to the original plain text, then the change in key is retained and algorithm starts again considering the obtained modified key as the putative key.

A scheme is followed to modify the key. Initially, symbols adjacent to each other are swapped and cipher text is decrypted to check correctness of the key. Suppose, we have the putative key as shown above, then initially swapping is done in k_1 and k_2 . In next iteration, swapping will be done in k_2 and k_3 and so on. This can be viewed as first round of swapping in which we swap adjacent element. In next round, elements at places i^{th} and $(i + 2)^{th}$ are swapped. In the successive rounds, distance between to be swapped elements is increased by 1. In all, there can be $\binom{26}{2}$ different

ways in which the key can be altered. This scheme of swapping can be visualized as

round 1:	$k_1 k_2$	$k_2 k_3$	$k_3 k_4$	$\dots$	$k_{23} k_{24}$	$k_{24} k_{25}$	$k_{25} k_{26}$
round 2:	$k_1 k_3$	$k_2 k_4$	$k_3 k_5$	$\dots$	$k_{23} k_{25}$	$k_{24} k_{26}$	
round 3:	$k_1 k_4$	$k_2 k_5$	$k_3 k_6$	$\dots$	$k_{23} k_{26}$		
	$\vdots$	$\vdots$					
round 24:	$k_1 k_{24}$	$k_2 k_{25}$	$k_3 k_{26}$				
round 25:	$k_1 k_{25}$	$k_2 k_{26}$					
round 26:	$k_1 k_{26}$						

To test the correctness of the key, algorithm creates a digram matrix of English letters from plain text. Each entry in matrix represents frequency count of that digram in plain text. This matrix is referred as E matrix in the algorithm. Similarly, another digram matrix is created from decrypted text. This matrix is called as D matrix in the algorithm. Correctness of the key is calculated as

$$\text{score (k)} = \sum_{i,j} |d_{i,j} - e_{i,j}|$$

where, $d_{i,j}$ and $e_{i,j}$ refers to i^{th} and j^{th} element in D and E matrices respectively. In each iteration of the algorithm, the score is calculated. If the score is smaller than the score obtained in previous iteration, it is retained else discarded. In other words, lower the score, better is the decrypted text. At the end of the algorithm, if the obtained decrypted text is the same as the original text, then the score will become zero. As a result, we will get the key and decrypted text, which will be similar to the original plain text.

5.3.2 Implementation

In this method, we extracted the opcode sequence from generated morphed versions of the given malware JavaScript. Then the frequency of each opcode is counted. All opcodes are arranged in the decreasing order of their count. Consider we get the

opcode order as

MOV, ADD, LOAD, SUB, JMP, CMP

This will be called as standard opcode order for the given malware. Then digram frequency graph is constructed by using the same opcode sequence. In this graph, each entry will represent frequency count of a digram. Adjacency matrix is used to represent this graph. This adjacency matrix will be analogous to E matrix in Jakobsen’s algorithm [13]. The opcode sequence is then extracted from the file to be tested. Frequency of each opcode is counted and arranged in decreasing order of their frequency count. Suppose we get the opcode sequence as

ADD, SUB, CMP, LOAD, MOV, MUL, JMP

This opcode order is mapped to previously calculated standard opcode order. This mapping will give us the putative key. Initial putative key is shown in Table 4. After

Table 4: Opcode Mapping (Putative Key)

Standard Opcode Order	MOV	ADD	LOAD	SUB	JMP	CMP	OTHER
Test Opcode Order	ADD	SUB	CMP	LOAD	MOV	MUL	JMP

doing one to one mapping, if testing file contains more unique opcodes, then they will be mapped to dummy opcode **OTHER**. Then all opcodes in testing file are replaced with corresponding opcode in the putative key. A digram matrix is created from the replaced opcode sequence. This matrix is analogous to D matrix in Jakobsen’s algorithm. A score is calculated by taking Manhattan distance between the D and E matrices.

$$\text{score} = \sum_{i,j} |d_{i,j} - e_{i,j}|$$

Key swapping is then done in the putative key by swapping the adjacent elements. For example, we have the putative key as shown in Table 5. First swapping occurs

between ADD and SUB to give a new putative key as shown in Table 6.

Table 5: Key Before Swapping

Standard Opcode Order	MOV	ADD	LOAD	SUB	JMP	CMP	OTHER
Test Opcode Order	ADD	SUB	CMP	LOAD	MOV	MUL	JMP

Table 6: Key After Swapping

Standard Opcode Order	MOV	ADD	LOAD	SUB	JMP	CMP	OTHER
Test Opcode Order	SUB	ADD	CMP	LOAD	MOV	MUL	JMP

By using this new key, we calculate scores for the given test file again. To reflect the changes made in the key, we need to construct a new D matrix. The new D matrix will be constructed by swapping the rows and columns in the previously constructed D matrix. Suppose we have a D matrix as shown in the Table 7.

Table 7: D Matrix before Swapping

	MOV	ADD	LOAD	SUB	JMP	CMP	OTHER
MOV	9	7	8	6	3	2	1
ADD	9	7	5	8	5	8	3
LOAD	5	8	6	4	3	7	1
SUB	4	8	4	7	4	8	3
JMP	5	3	8	5	3	8	6
CMP	3	2	5	6	3	7	3
OTHER	9	6	0	2	5	9	7

As we swap entries corresponding to MOV and ADD in the putative key, to achieve the same effect on D matrix, we need to swap the rows and columns corresponding to MOV and ADD. After swapping, we will get a new D matrix as shown in Table 8.

Thereafter, score is calculated by taking the Manhattan distance between the newly created D matrix and E matrix. If the new score is greater than the previous score, then we discard the swapping and move towards next swapping. Here in this

Table 8: D Matrix after Swapping

	MOV	ADD	LOAD	SUB	JMP	CMP	OTHER
MOV	7	9	5	8	5	8	3
ADD	7	9	8	6	3	2	1
LOAD	8	5	6	4	3	7	1
SUB	8	4	4	7	4	8	3
JMP	3	5	8	5	3	8	6
CMP	2	3	5	6	3	7	3
OTHER	6	9	0	2	5	9	7

example, next swapping will be done between MOV and LOAD. If the new score is less than the previously calculated score, then we will retain the swapping and continue algorithm from beginning, keeping swapped key as the putative key.

In the first round, swapping will be done between adjacent elements in the key. In next round, swapping will be done between elements on the i^{th} and $i + 2^{th}$ position in the key. In successive rounds, distance between elements to be swapped is increased by one. This swapping scheme can be visualized as

$$\begin{array}{lllllll}
 \text{round 1:} & k_1|k_2 & k_2|k_3 & k_3|k_4 & \dots & k_{n-3}|k_{n-2} & k_{n-2}|k_{n-1} & k_{n-1}|k_n \\
 \text{round 2:} & k_1|k_3 & k_2|k_4 & k_3|k_5 & \dots & k_{n-3}|k_{n-2} & k_{n-2}|k_n & \\
 \text{round 3:} & k_1|k_4 & k_2|k_5 & k_3|k_6 & \dots & k_{n-3}|k_n & & \\
 & \vdots & & \vdots & & & \ddots & \\
 \text{round n-2:} & k_1|k_{n-2} & k_2|k_{n-1} & k_3|k_n & & & & \\
 \text{round n-1:} & k_1|k_{n-1} & k_2|k_n & & & & & \\
 \text{round n:} & k_1|k_n & & & & & &
 \end{array}$$

The smallest score achieved in this process is the score of similarity. To classify a file, this score is compared against a predefined threshold. If the score is less than the threshold value, then file under consideration belongs to the given malware family.

5.4 Singular Value Decomposition

Singular Value Decomposition is a technique used in image detection which can also be used to detect malware files. Deshpande [9] describes a method based on Singular Value Decomposition to detect metamorphic malware. The scores are calculated by taking euclidean distance between weights of known malware sample and unknown malware sample. Weights are calculated by projecting column vector of a file on a space enclosed by eigenvectors of a known malware [14]. This enclosed space is called as eigenspace.

Eigenspace is calculated by projecting eigenvectors of covariance matrix. Covariance matrix is calculated as AA^T . Singular Value Decomposition of a matrix A is represented as

$$A = USV^T$$

where, U is covariance matrix. S is a diagonal matrix with square root of eigenvalues common to both matrices U and V .

5.4.1 Algorithm

The algorithm works in two phases—Training Phase and Testing Phase. In Training Phase, weights of known malware files are calculated by projecting them on to the eigenspace. In Testing Phase, we project files to be tested on to the same eigenspace and calculate their corresponding weights. Thereafter, euclidean distance between previously calculated and newly calculated weights are taken to compare them against the threshold value. Following sections will describe these two phases stepwise.

5.4.1.1 Training Phase

- In training phase, raw bytes from text or code section of training dataset are extracted. These raw bytes are converted into decimal values to construct a column vector.
- Then covariance matrix is constructed. Covariance matrix is a product of matrix A and its transpose. Matrix A is constructed by appending all column vectors. Suppose we have $\phi_1, \phi_2, \phi_3, \dots, \phi_M$ column vectors, then matrix A can be constructed as

$$A = [\phi_1 \ \phi_2 \ \phi_3 \ \dots \ \phi_M]$$

- Now, Eigenvectors for covariance matrix are calculated. These eigenvectors determine eigenspace. Eigenvectors with large eigenvalues represents prominent characteristics of a malware. So, we ignore eigenvectors with small eigenvalues.
- All eigenvectors are arranged in the descending order of their eigenvalues. For our experiment, we consider top M' eigen vectors from M eigenvectors (where $M' < M$). We project these eigenvectors onto the space. Space bounded by these eigenvectors is called eigenspace.
- Original malware can be constructed by using these M' eigenvectors by adding them with their weights. Suppose for a malware V we have $e_1, e_2, e_3, \dots, e_{M'}$ eigenvectors with corresponding weights $w_1, w_2, w_3, \dots, w_{M'}$. Then malware V can be reconstructed as

$$V = e_1 \times w_1 + e_2 \times w_2 + \dots + e_{M'} \times w_{M'}$$

$$V = \sum_{i=1}^{M'} w_i \times e_i$$

Therefore, weights for each sample can be obtained by

$$w_i = \sum_{i=0}^{M'} e_i^T \times V$$

The obtained weights for a particular virus V_i can be represented as

$$\Omega_i^T = [w_i, w_2, w_3, \dots, w_{M'}]$$

- Obtained weights for all malware samples can be represented as Δ

$$\Delta = [\Omega_1, \Omega_2, \Omega_3, \dots, \Omega_M]$$

5.4.1.2 Testing Phase

- In this phase, we first construct a column vector for each file which is to be tested. If the size of the file is less than N bytes, then column vector is appended with zeros. And if size of the file is more than N bytes, then column vector is constructed by taking first N bytes from file.
- This column vector is projected on to the previously calculated eigenspace. Weights for file T which is to be tested can be calculated by

$$\omega_i = \sum_{i=0}^{M'} e_i^T \times V_n$$

$$\Omega_n^T = [\omega_1, \omega_2, \omega_3, \dots, \omega_{M'}]$$

- Euclidean distance between Ω_n vector and weight vector generated in training phase can be calculated by

$$\text{Distance} = \sqrt{(\omega_1^2 - w_1^2) + (\omega_2^2 - w_2^2) + \dots + (\omega_{M'}^2 - w_{M'}^2)}$$

where, ω represents weight of test file and w represents weight of training file.

- A threshold is calculated by testing few known malware files which belong to the same malware family.

Figure 7: SVD Process

5.4.2 Implementation

To implement above described algorithm to detect given JavaScript malware family, we compiled JavaScript into Java byte code. To every byte code, there is an equivalent hex value associated. We converted those hex values to decimal values. Suppose we have M files in the training dataset, each file having different number of bytes. The largest file has N bytes, using which an $N \times M$ matrix is created. Zeros are appended to smaller files and first N bytes are chopped from the larger files. This matrix is called a covariance matrix.

We pass this matrix to JAMA API which is developed in Java to calculate eigenvectors and eigenvalues. These obtained eigenvectors form a eigenspace which is used to calculate the weights. Weights for files which we want to test are calculated by projecting their column vectors on the generated eigenspace. This whole process can be visualized as shown in Figure 7. Once we get the weights for both the training set and the files to be tested, then scores are calculated by taking the euclidean distance between weights of the training files and the testing files. Experimental analysis of given JavaScript malware is explained in Section 6.

CHAPTER 6

Experiments

This chapter discusses about the experimental setup, files used in the experiment and results obtained from these experiments. Experiment involves implementation and testing of the following methods:

- Hidden Markov Model Method
- Opcode Graph Similarity Method
- Simple Substitution Distance Method
- Singular Value Decomposition Method

All these methods work on statistical properties of opcodes. Given malware files are written in JavaScript. To convert JavaScript code into opcode sequence, we used the Rhino JavaScript Engine. We modified Rhino in such a way that it gives opcode sequence at compile time as discussed in Section 4. We performed all these experiments on a machine with the following configuration.

Table 9: Machine Specifications

Model	Lenovo ThinkPad T530
Processor	3rd Generation Intel Core i7-3110M @2.80GHz
RAM	8.00GB
System type	64-bit OS
Java Compiler	Java 6
Operating System	Ubuntu 12.10

The given JavaScript malware carries a morphing engine with itself. After every infection, the internal structure of malware gets changed. For the experiments, we

generated 100 morphed versions of the given malware. These files are used in training phase and testing phase.

The experiments undertaken need benign files to test the proposed detection method. We collected benign files from different open source JavaScript libraries [5, 7, 12, 15, 36]. Effectiveness of each method is judged by plotting Receiver Operating Characteristics (ROC) curve [4]. Area under the ROC curve will give us the degree of correctness for every individual method.

Table 10: Source of Benign Files

Source	Number of files
Cassandra Project	5
DataTables Library	10
Flanagan’s book	5
jQuery	5
YUI Library	15
Total	40

6.1 Receiver Operating Characteristics

Receiver Operating Characteristics (ROC) curve is a graphical representation for correctness of classification system [4]. It is created by plotting true positives out of the total actual positives (called as $TPR = \text{true positive rate}$) vs. the fraction of false positives out of the total actual negatives (called as $FPR = \text{false positive rate}$), at various threshold values. TPR and FPR are also known as sensitivity and fall out of classification system. For plotting ROC curve, we first calculate probabilities for true positive and false positive. Then a curve is generated by plotting the cumulative distribution function of the true positive probabilities on the y-axis vs the cumulative distribution function of false positive probabilities on the x-axis [11]. The Area Under the Curve (AUC) of ROC gives a measure of correctness of classification. Following

Figure 8: HMM Score Analysis (N=2)

sections will discuss results of above mentioned method by plotting ROC curves for each method.

6.2 Hidden Markov Model

To test the correctness of HMM method, we did a number of experiments. It has been observed that number of states do not make any significant difference in the score calculation. So, we decided to continue with number of states equal to 2. Opcode sequence for benign files and malware files are same as discussed above. We randomly choose 20 opcode sequences from generated pool of malware samples to train HMM. Then we calculated scores for rest of the malware samples and benign files. To visualize the distribution of scores we plotted Scatter plot for the obtained scores. Scatter Plot shown in Figure 8 clearly depicts the distinction between score of malware and benign files.

The effectiveness of HMM can be judged using the ROC curve [4]. ROC curve for the above described experiment is shown in Figure 9. As area under the curve

Figure 9: ROC curve for HMM

(AUC) is 1, it shows that trained HMM model is correctly distinguishing between virus files and benign files.

6.3 Opcode Graph Similarity Method

For this experiment, we used same malware files and benign files as discussed above. Threshold is determined by the method described in Section 5.2.3. Purpose of this experiment is to verify whether the computed scores for benign files and malware files are distinguishable or not.

Scores are obtained in this experiment are plotted on scatter graph as shown in Figure 10. Red points in graph shows scores of malware file vs malware file and green points shows scores of malware file vs benign file. It can be observed from Table 10 that scores for malware file vs malware file are in order of 10^{-2} while scores for malware file vs benign file are in order of 10^{-1} . From this, we can clearly conclude that this method is distinguishing files correctly.

ROC curve for the experiment is shown in Figure 11. Area under the curve is 1, which shows perfect classification between malware files and benign files.

Figure 10: Opcode Graph Similarity Analysis

Figure 11: ROC curve for Opcode Graph Similarity Method

6.4 Simple Substitution Distance Method

For this experiment, we have randomly selected 10 virus files from a pool of generated morphed copies of malware and 10 benign files. Their scores are listed below in Table 11. Looking at the score in Table 11, we can conclude that distribution of opcode frequency in malware samples is consistent. On the other hand, as the score for the benign file is large, it shows that the distribution of opcode frequency differs

Figure 12: Simple Substitution Method Scores Analysis

a lot.

Table 11: Simple Substitution Method Scores

File name	Score	File Name	Score
Ben_1	2.65007	Virus_1	1.82535
Ben_2	2.52474	Virus_2	1.82537
Ben_3	2.49072	Virus_3	1.82537
Ben_4	2.58470	Virus_4	1.82537
Ben_5	2.65446	Virus_5	1.82539
Ben_6	2.66617	Virus_6	1.82539
Ben_7	2.64091	Virus_7	1.82537
Ben_8	2.65737	Virus_8	1.82537
Ben_9	2.66809	Virus_9	1.82539
Ben_10	2.64965	Virus_10	1.82537

Scatter Plot in Figure 12 clearly shows the distinction between scores of malware and benign files. For the same experiment, we have drawn a ROC curve as shown in Figure 13. Area under the curve is 1, which indicates perfect classification of files.

Figure 13: ROC for Simple Substitution Method

6.5 Singular Value Decomposition Method

We randomly selected 20 samples of benign JavaScript and malware JavaScript files from above mentioned pool of benign files and malware samples. Opcodes from these files are then converted to decimal equivalent with the help of opcodes to hex value mapping. Eigenspace is created using training malware samples. Weights of files (files to be tested) are obtained by projecting their column vectors on previously generated eigenspace. Scatter plot of these values is shown in Figure 14.

To test the effectiveness of experiment, we plot ROC curve for the scores obtained in experiment. The ROC curve for the experiment is as shown in Figure 15. Area under the Curve is 1. AUC 1 shows that algorithm described in Section 5.4 perfectly distinguish between malware samples and benign samples.

Figure 14: SVD Scatter Plot Analysis

Figure 15: ROC curve for SVD

CHAPTER 7

Enhancement

Looking at the analysis of given malware using Hidden Markov Model, Simple Substitution Method, Opcode Graph Similarity method and Singular Value Decomposition method, it is clear that given malware can be detected by methods based on statistical properties of opcodes. To evade such detection we executed an experiment by inserting deadcode into given JavaScript malware.

To implement this, we have created a service which will run on central server. Whenever malware gets executed, it creates a connection with this central server. A random number `n` is passed to the service. This service will generate `n` number of functions and send back to malware. The malware then adds this deadcode to its body. By adding this deadcode, functionality will remain same but the statistical characteristic of opcode sequence will change. This will help to evade detection by above mentioned methods.

7.1 Hidden Markov Model

To evade the detection by Hidden Markov Model we modified the service which will insert deadcode in controlled amount. We chose 20 morphed versions of the given malware. Then we created different versions of chosen files by adding deadcode. Initially we started with 100 random functions as deadcode. Then we compiled this new version with the help of Rhino and got opcode sequence for chosen files. Scores are calculated against previously calculated models.

We repeated the process with increasing number of deadcode function each time

Figure 16: Hidden Markov Model AUC Analysis

and calculated scores for each case. After plotting ROC curve for these cases, it has been observed that HMM can clearly distinguish between benign files and malware files up to 2900 deadcode functions. Table 12 gives us an idea about decrease in Area Under the Curve (AUC) of ROC as we increase the deadcode. The trend of change in AUC can be visualized by the graph shown in Figure 16. Corresponding ROC curves are shown in Appendix A, Section A.1.

Table 12: AUC for Hidden Markov Model

Functions	AUC	Functions	AUC
100	1	6000	0.99517
200	1	7000	0.98028
300	1	8000	0.96922
400	1	9000	0.96528
500	1	10000	0.96294
600	1	11000	0.95439
700	1	12000	0.94709
800	1	13000	0.93387
900	1	14000	0.93307
1000	1	15000	0.92757
1500	1	16000	0.91845
2000	1	17000	0.90832
3000	1	18000	0.89851
4000	1	19000	0.88560
5000	1	20000	0.86933

7.2 Opcode Graph Similarity Method

To evade detection by using Graph Similarity Method, we used the same experimental setup. Graph Similarity Method is very sensitive towards deadcode. Scores change very rapidly as compared to HMM for small amount of deadcode. Therefore, for this experiment we started with adding 100 deadcode functions. In each iteration we increased number of deadcode functions by 100. Table 13 shows the AUC for each case. The trend of change in AUC can be visualized by graph shown in Figure 17. Corresponding ROC curves are shown in Appendix A, Section A.2.

Table 13: AUC for Opcode Graph Similarity Method

Functions	AUC	Functions	AUC
100	1	6000	1
200	1	7000	1
300	1	8000	1
400	1	9000	1
500	1	10000	1
600	1	11000	1
700	1	12000	1
800	1	13000	1
900	1	14000	0.99991
1000	1	15000	0.99974
1500	1	16000	0.99974
2000	1	17000	0.99947
3000	1	18000	0.99912
4000	1	19000	0.99886
5000	1	20000	0.99868

7.3 Simple Substitution Distance Method

To test the enhancement against Simple Substitution Distance method, we used the same setup. Now we have 25 different morphed versions of given malware with 100 to 20000 deadcode functions. After doing regression testing on all these cases, we found that Simple Substitution Distance Method can clearly distinguish between

Figure 17: Opcode Graph Similarity Method AUC Analysis

malware files and benign files upto 5000 deadcode functions. After that AUC starts decreasing slowly. Table 14 gives complete list of AUCs in this experiment and its trend can be observed in Figure 18. Corresponding ROC curves are shown in Appendix A, Section A.3.

Table 14: AUC for Simple Substitution Distance Method

Functions	AUC	Functions	AUC
100	1	6000	1
200	1	7000	1
300	1	8000	1
400	1	9000	1
500	1	10000	1
600	1	11000	0.96133
700	1	12000	0.92400
800	1	13000	0.78800
900	1	14000	0.19467
1000	1	15000	0.17067
1500	1	16000	0.11833
2000	1	17000	0.25833
3000	1	18000	0.33667
4000	1	19000	0.24000
5000	1	20000	0.15833

Figure 18: Simple Substitution Distance Method AUC Analysis

Figure 19: Singular Value Decomposition AUC Analysis

7.4 Singular Value Decomposition Method

To evade detection by the Singular Value Decomposition method, we used the same experimental setup. We started with 100 deadcode functions and went on till 10000 deadcode functions. It has been observed that with the increase in deadcode functions, AUC kept decreasing. At 9000 deadcode functions, AUC went below 0.5 which means results of the method are no better than tossing a coin. Change in AUC can be visualized by graph shown in Figure 19. Complete list of AUC is listed in Table 15. Corresponding ROC curves are shown in Appendix A, Section A.4.

Table 15: AUC for Singular Value Decomposition Method

Functions	AUC	Functions	AUC
100	0.89754	6000	0.56608
200	0.88719	7000	0.54653
300	0.89045	8000	0.52633
400	0.88477	9000	0.49789
500	0.87839	10000	0.47226
600	0.86709	11000	0.45442
700	0.84899	12000	0.43638
800	0.83663	13000	0.41377
900	0.83045	14000	0.39261
1000	0.83141	15000	0.38166
1500	0.81121	16000	0.37638
2000	0.77688	17000	0.36583
3000	0.70849	18000	0.35678
4000	0.67116	19000	0.34271
5000	0.60020	20000	0.32764

CHAPTER 8

Conclusion and Future Work

The aim of this experiment was to design and test a system to detect metamorphic JavaScript malware. We adopted a new approach of statistical analysis for JavaScript malware. In this approach, we compiled JavaScript into Java class files using Rhino—a JavaScript Engine. We decompiled these class files and extracted a sequence of Java bytecode. Using Hidden Markov Model, Opcode Graph Similarity, Simple Substitution Distance and Singular Value decomposition techniques, we compared similarity between morphed files and random benign files. It was found that the given malware Transcriptase is detectable using these techniques as morphed versions exhibits same statistical properties.

Detection by using these methods can be evaded by adding deadcode. During our experiments, it was observed that Hidden Markov Model, Opcode Graph Similarity, Simple Substitution Distance and Singular Value Decomposition starts misclassification between JavaScript malware and benign files at 3000, 100, 5500 and 100 deadcode functions respectively. This shows that the Opcode Graph Similarity and Singular Value Decomposition techniques are very sensitive to deadcode which concludes that these two techniques can classify files having highly similar opcode statistics.

The approach discussed in this paper can be used for all metamorphic JavaScript malware files. A generic system can thus be designed for classifying JavaScript files as the primary step of web security system. As Hidden Markov Model and Simple Substitution Distance techniques can classify files with considerable amount of deadcode. Use of these methods in web security system to classify JavaScript files will

give cluster of suspicious JavaScript files. This will help to reduce the huge set of malware under consideration into a small subset.

Future work for this experiment can include enhancing malware. In our experiments, we used enhanced malware which creates connection with some remote server to get deadcode functions. The malware script can be modified in such a way that it will gather function definitions from other benign JavaScript files in that folder and use them as deadcode at the time of infection. This will avoid connection with some static server for deadcode which could be exploited for detection of morphed copies of the malware.

The given malware uses custom designed meta-language for carrying its code. Presently, there is one to one static mapping between meta-language symbols and original JavaScript literals. This static mapping can be made dynamic, so that after each new infection meta-language code will also change. This enforcement will reduce the chances of getting detected because of static meta-language code. Mapping of symbols can be passed as configuration to meta-language compiler.

Future work can also include developing a browser plugin for meta-morphic JavaScript malware detection. As JavaScript malware executes in browsers and JavaScript Engine is a part of the browser, modified Rhino described in this paper can be used to develop a browser plugin which will take opcode sequence generated by modified Rhino and verify JavaScript at page load time. If JavaScript is found to be malicious, then it can restrict page load events or browser capabilities. This will provide dynamic protection from malware infecting through browser.

LIST OF REFERENCES

- [1] B. Anderson, et al. Graph-based Malware Detection using Dynamic Analysis. *Journal of Computer Virology*, Volume 7, Issue 4, pp. 247–258. (2011)
- [2] S. Attaluri, S. McGhee, and M. Stamp. Profile Hidden Markov Models and Metamorphic Virus Detection. *Journal in Computer Virology*, Volume 5, Issue 2, pp. 151–169. (2009)
- [3] T. Austin, E. Filiol, S. Josse, M. Stamp. Exploring Hidden Markov Models for Virus Analysis: A Semantic Approach. *Proceedings of 46th Hawaii International Conference on System Sciences*. (2013)
- [4] A. P. Bradley. The use of the Area Under the ROC Curve in the Evaluation of Machine Learning Algorithms. *Journal Pattern Recognition*, Volume 30, Issue 7, pp. 1145–1159. (1997)
- [5] Cassandra Library. <http://cassandra.apache.org/>
- [6] E. Daoud, I. Jebril. Computer Virus Strategies and Detection Methods. *International Journal of Open Problems in Computer Science and Mathematics*, Volume 1, Issue 2. [http://www.emis.de/journals/IJOPCM/files/IJOPCM\(vol.1.2.3.S.08\).pdf](http://www.emis.de/journals/IJOPCM/files/IJOPCM(vol.1.2.3.S.08).pdf) (2008)
- [7] DataTables Library. <https://github.com/DataTables/DataTables>
- [8] P. Desai. Towards an Undetectable Computer Virus. Master’s Projects. Paper 90. http://scholarworks.sjsu.edu/etd_projects/90. (2008)
- [9] S. Deshpande, Y. Park, and M. Stamp. Eigenvalue Analysis for Metamorphic Detection.
- [10] Journal of Computer Virology and Hacking Techniques.(2012)
- [11] T. Fawcett. An Introduction to ROC Analysis. <http://people.inf.elte.hu/kiss/13dwhdm/roc.pdf>
- [12] D. Flanagan. *JavaScript: The Definitive Guide*, 6th Edition.
- [13] T. Jakobsen. A Fast Method for the Cryptanalysis of Substitution Ciphers. *Cryptologia*, Volume 19, pp. 265–274. (1995)
- [14] R. Jidigam, et al. Metamorphic Detection Using Singular Value Decomposition, Master’s Project. Paper 330. http://scholarworks.sjsu.edu/etd_projects/330. (2013)

- [15] JQuery Library. <http://jquery.com/>
- [16] D. Lin, M. Stamp. Hunting for Undetectable Metamorphic Viruses. *Journal in Computer Virology*, Volume 7, Issue 3, pp. 201–214. (2011)
- [17] Metamorphism and Self-Compilation in JavaScript. <http://spth.virii.lu/MSCJS.txt>
- [18] D. W. Mount. Bioinformatics: Sequence and Genome Analysis. *Cold Spring Harbor Laboratory*. (2004)
- [19] N. Provos, P. Mavrommatis, M. Rajab, F. Monroe. All Your iFRAMEs Point to Us. *Proceedings of the 17th Conference on Security Symposium*. (2008)
- [20] B. Rad, M. Masrom, S. Ibrahim. Camouflage in Malware: from Encryption to Metamorphism. *International Journal of Computer Science and Network Security*, Volume 12, Issue 8, pp. 74. (2012)
- [21] Rhino Documentation. https://developer.mozilla.org/en-US/docs/Rhino_documentation
- [22] N. Runwal, R. M. Low, and M. Stamp. Opcode Graph Similarity and Metamorphic Detection. *Journal in Computer Virology*, Volume 8, Issue 1-2, pp. 37–52. (2012)
- [23] G. Shanmugam, R. Low, M. Stamp. Simple Substitution Distance and Metamorphic Detection. *Journal of Computer Virology and Hacking Techniques*, Volume 9, Issue 3, pp. 159–170. (2013)
- [24] SOPHOS Securitiy Threat Report. <http://www.sophos.com/en-us/medialibrary/PDFs/other/sophossecuritythreatreport2013.pdf>. (2013)
- [25] Source of Transcriptase JavaScript Malware. <http://spth.virii.lu/Transcriptase.rar>
- [26] S. Sridhara. Metamorphic Worm that Carries its Own Morphing Engine. *Journal of Computer Virology and Hacking Techniques*, Volume 9, Issue 2, pp. 49–58. (2013)
- [27] M. Stamp. A Revealing Introduction to Hidden Markov Models. <http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf>. (2012)
- [28] P. Szor. The Art of Computer Virus Research and Defense. Addison-Wesley Professional. (2005)
- [29] P. Szor, P. Ferrie. Hunting for Metamorphic, Symantec Security Response. <http://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf>

- [30] S. Venkatachalam. Detecting Undetectable Computer Viruses. Master's Projects. Paper 156. http://scholarworks.sjsu.edu/etd_projects/156. (2010)
- [31] A. Venkatesan. Code Obfuscation and Virus Detection. Master's Projects. Paper 116. http://scholarworks.sjsu.edu/etd_projects/116. (2008)
- [32] Walenstein, R. Mathur, M. Chouchane, R. Chouchane, and A. Lakhotia. The Design Space of Metamorphic Malware. In *Proceedings of the 2nd International Conference on Information Warfare*. (2007)
- [33] Working of Rhino. <http://www.quora.com/JavaScript-programming-language/How-does-a-JavaScript-engine-work>
- [34] W. Wong. Analysis and Detection of Metamorphic Computer Viruses. Master's Projects. Paper 153. http://scholarworks.sjsu.edu/etd_projects/153. (2006)
- [35] W. Xu, F. Zhang, S. Zhu. The Power of Obfuscation Techniques in Malicious JavaScript Code: A Measurement Study. *Malicious and Unwanted Software (MALWARE)*, 7th International Conference, Volume 7, pp. 9–16. (202)
- [36] YUI Library. <http://yuilibrary.com/>
- [37] P. Zbitskiy. Code Mutation Techniques by means of Formal Grammars and Automaton. *Journal in Computer Virology*, Volume 5, Issue 3, pp 199–207. (2009)

APPENDIX

ROC curves

A.1 Hidden Markov Model

Figure A.20: ROC for 2500 Functions for HMM

Figure A.21: ROC for 3000 Functions for HMM

Figure A.22: ROC for 3500 Functions for HMM

Figure A.23: ROC for 4000 Functions for HMM

Figure A.24: ROC for 4500 Functions for HMM

Figure A.25: ROC for 5000 Functions for HMM

Figure A.26: ROC for 5500 Functions for HMM

Figure A.27: ROC for 6000 Functions for HMM

Figure A.28: ROC for 6500 Functions for HMM

Figure A.29: ROC for 7000 Functions for HMM

Figure A.30: ROC for 7500 Functions for HMM

Figure A.31: ROC for 8000 Functions for HMM

Figure A.32: ROC for 8500 Functions for HMM

Figure A.33: ROC for 9000 Functions for HMM

Figure A.34: ROC for 9500 Functions for HMM

Figure A.35: ROC for 10000 Functions for HMM

Figure A.36: ROC for 10500 Functions for HMM

Figure A.37: ROC for 11000 Functions for HMM

Figure A.38: ROC for 11500 Functions for HMM

Figure A.39: ROC for 12000 Functions for HMM

Figure A.40: ROC for 12500 Functions for HMM

Figure A.41: ROC for 13000 Functions for HMM

Figure A.42: ROC for 13500 Functions for HMM

Figure A.43: ROC for 14000 Functions for HMM

Figure A.44: ROC for 14500 Functions for HMM

Figure A.45: ROC for 15000 Functions for HMM

A.2 Opcode Graph Similarity Method

Figure A.46: ROC for 100 Functions for OGS

Figure A.47: ROC for 200 Functions for OGS

Figure A.48: ROC for 300 Functions for OGS

Figure A.49: ROC for 400 Functions for OGS

Figure A.50: ROC for 500 Functions for OGS

Figure A.51: ROC for 600 Functions for OGS

Figure A.52: ROC for 700 Functions for OGS

Figure A.53: ROC for 800 Functions for OGS

Figure A.54: ROC for 900 Functions for OGS

Figure A.55: ROC for 1000 Functions for OGS

Figure A.56: ROC for 2000 Functions for OGS

Figure A.57: ROC for 3000 Functions for OGS

Figure A.58: ROC for 4000 Functions for OGS

Figure A.59: ROC for 5000 Functions for OGS

Figure A.60: ROC for 6000 Functions for OGS

Figure A.61: ROC for 7000 Functions for OGS

Figure A.62: ROC for 8000 Functions for OGS

Figure A.63: ROC for 9000 Functions for OGS

Figure A.64: ROC for 10000 Functions for OGS

Figure A.65: ROC for 11000 Functions for OGS

Figure A.66: ROC for 12000 Functions for OGS

Figure A.67: ROC for 13000 Functions for OGS

Figure A.68: ROC for 14000 Functions for OGS

Figure A.69: ROC for 15000 Functions for OGS

A.3 Simple Substitution Distance Method

Figure A.70: ROC for 5000 Functions for SSM

Figure A.71: ROC for 5500 Functions for SSM

Figure A.72: ROC for 6000 Functions for SSM

Figure A.73: ROC for 6500 Functions for SSM

Figure A.74: ROC for 7000 Functions for SSM

Figure A.75: ROC for 7500 Functions for SSM

Figure A.76: ROC for 8000 Functions for SSM

Figure A.77: ROC for 8500 Functions for SSM

Figure A.78: ROC for 9000 Functions for SSM

Figure A.79: ROC for 9500 Functions for SSM

Figure A.80: ROC for 10000 Functions for SSM

Figure A.81: ROC for 10500 Functions for SSM

Figure A.82: ROC for 11000 Functions for SSM

Figure A.83: ROC for 11500 Functions for SSM

Figure A.84: ROC for 12000 Functions for SSM

Figure A.85: ROC for 12500 Functions for SSM

Figure A.86: ROC for 13000 Functions for SSM

Figure A.87: ROC for 13500 Functions for SSM

Figure A.88: ROC for 14000 Functions for SSM

Figure A.89: ROC for 14500 Functions for SSM

Figure A.90: ROC for 15000 Functions for SSM

Figure A.91: ROC for 15500 Functions for SSM

A.4 Singular Value Decomposition Method

Figure A.92: ROC for 100 Functions for SVD

Figure A.93: ROC for 200 Functions for SVD

Figure A.94: ROC for 300 Functions for SVD

Figure A.95: ROC for 400 Functions for SVD

Figure A.96: ROC for 500 Functions for SVD

Figure A.97: ROC for 600 Functions for SVD

Figure A.98: ROC for 700 Functions for SVD

Figure A.99: ROC for 800 Functions for SVD

Figure A.100: ROC for 900 Functions for SVD

Figure A.101: ROC for 1000 Functions for SVD

Figure A.102: ROC for 1100 Functions for SVD

Figure A.103: ROC for 1200 Functions for SVD

Figure A.104: ROC for 1300 Functions for SVD

Figure A.105: ROC for 1400 Functions for SVD

Figure A.106: ROC for 1500 Functions for SVD

Figure A.107: ROC for 1600 Functions for SVD

Figure A.108: ROC for 1700 Functions for SVD

Figure A.109: ROC for 1800 Functions for SVD

Figure A.110: ROC for 1900 Functions for SVD

Figure A.111: ROC for 2000 Functions for SVD

Figure A.112: ROC for 2100 Functions for SVD

Figure A.113: ROC for 2200 Functions for SVD

Figure A.114: ROC for 2300 Functions for SVD

Figure A.115: ROC for 2400 Functions for SVD

Figure A.116: ROC for 2300 Functions for SVD

Figure A.117: ROC for 2400 Functions for SVD

Figure A.118: ROC for 2300 Functions for SVD

Figure A.119: ROC for 2400 Functions for SVD

Figure A.120: ROC for 2500 Functions for SVD

Figure A.121: ROC for 2600 Functions for SVD

Figure A.122: ROC for 2700 Functions for SVD

Figure A.123: ROC for 2800 Functions for SVD

Figure A.124: ROC for 2900 Functions for SVD

Figure A.125: ROC for 3000 Functions for SVD

Figure A.126: ROC for 3500 Functions for SVD

Figure A.127: ROC for 4000 Functions for SVD

Figure A.128: ROC for 4500 Functions for SVD

Figure A.129: ROC for 5000 Functions for SVD

Figure A.130: ROC for 5500 Functions for SVD

Figure A.131: ROC for 6000 Functions for SVD

Figure A.132: ROC for 6500 Functions for SVD

Figure A.133: ROC for 7000 Functions for SVD

Figure A.134: ROC for 7500 Functions for SVD

Figure A.135: ROC for 8000 Functions for SVD

Figure A.136: ROC for 8500 Functions for SVD

Figure A.137: ROC for 9000 Functions for SVD

Figure A.138: ROC for 9500 Functions for SVD

Figure A.139: ROC for 10000 Functions for SVD

Figure A.140: ROC for 10500 Functions for SVD

Figure A.141: ROC for 11000 Functions for SVD

Figure A.142: ROC for 12500 Functions for SVD

Figure A.143: ROC for 12000 Functions for SVD

Figure A.144: ROC for 12500 Functions for SVD

Figure A.145: ROC for 13000 Functions for SVD

Figure A.146: ROC for 13500 Functions for SVD

Figure A.147: ROC for 14000 Functions for SVD

Figure A.148: ROC for 14500 Functions for SVD

Figure A.149: ROC for 15000 Functions for SVD