Spring 5-22-2017

Transcriptase–Light: A Polymorphic Virus Construction Kit

Saurabh Borwankar
San Jose State University

Recommended Citation
Borwankar, Saurabh, "Transcriptase–Light: A Polymorphic Virus Construction Kit" (2017). *Master's Projects*. 513.
DOI: https://doi.org/10.31979/etd.um68-q8av
https://scholarworks.sjsu.edu/etd_projects/513

Transcriptase–Light: A Polymorphic Virus Construction Kit

A Project

Presented to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Saurabh Borwankar

May 2017

The Designated Project Committee Approves the Project Titled

Transcriptase–Light: A Polymorphic Virus Construction Kit

by

Saurabh Borwankar

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2017

ABSTRACT

Transcriptase–Light: A Polymorphic Virus Construction Kit

by Saurabh Borwankar

Many websites use JavaScript to display dynamic and interactive content. Hence, attackers are developing JavaScript–based malware. In this paper, we focus on Transcriptase JavaScript malware.

The high–level and dynamic nature of the JavaScript language helps malware writers to create polymorphic and metamorphic malware using obfuscation techniques. These types of malware change their internal structure on each infection, making them difficult to detect with traditional methods. These types of malware can be detected using machine learning methods.

This project creates Transcriptase–Light, a new polymorphic construction kit. We perform an experiment with the Transcriptase–Light against a hidden Markov model. Our experiment shows that the HMM based detector failed in detecting Transcriptase–Light. After observing the results, we try to detect malware using the decryption part of Transcriptase–Light. To avoid detection, we generate the polymorphic version of the decryption part.

ACKNOWLEDGMENTS

I am very thankful to Dr. Thomas Austin for his support and guidance through out the project journey, and also for believing in me. Also, I would like to thank the committee members Dr. Mark Stamp and Dr. Robert Chun for their valuable time. I would like to thank Mr. Fabio Di Troia for sharing his expertise on the project topic. Finally, I am very thankful to my family and my friends for their support!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

The growth of the internet has raised security concerns among businesses and their consumers. Computer networks transfer data from one point to another. This network is very susceptible to attack. Cyber security is a branch of information technology which deals with protection of the data through the web. Many times, transactions happening over the internet contains sensitive data such as health records, bank detail, and other personal information. Attackers like to exploit and tap such sensitive information.

Websites and web pages are designed and implemented using web technologies such as HTML, CSS, and JavaScript, which is widely used in displaying dynamic content on web pages. JavaScript is also used to display interactive content on web pages. JavaScript is a high-level, dynamic, and untyped programming language [2]. These properties of JavaScript can be used to create malicious content and malware. Malware is any software program or piece of code that is designed to harm normal computer operations and computer users [3].

Ransomware is one example of JavaScript malware. Ransomware is a type of malware which disables the functionality of a user's computer activity in some way [4]. Ransomware malware displays a message in the web browser and demands payment in order to restore the functionality of the user's computer. Another example of a malicious act is to steal the user's sensitive information. Many people use e-commerce systems for trading. Some websites need to enter user sensitive information such as financial and medical details. This private information can be sent to a non-trusted source without the consent of the user [3].

An infection happens through the injection of malicious JavaScript code into legitimate websites [5]. When the user visits this compromised site, a malicious

JavaScript file gets downloaded on the user system [6]. Once the malicious file is downloaded on the system, it performs various functionalities such as redirecting the network traffic, capturing key logs, or downloading more malicious files without user authorization. Another source to carry out a malicious act is third-party add-ons and plugins for the web browser. When the user installs any untrusted plugin, malicious files gets downloaded on the user system [3].

In this project, we implement a polymorphic JavaScript virus construction kit (VCK) inspired by Transcriptase [7] and analyze different methods to detect this malware. This polymorphic malware takes advantage of JavaScript Obfuscation techniques to hide from detection mechanisms. We design an experiment to detect this malware using Hidden Markov Models(HMM).

This paper is organized as follows: Chapter 2 provides information on background of malware, malware detection technique, and JavaScript obfuscation techniques; Chapter 3 provides insight on the implementation of JavaScript malware. Chapter 4 focuses on experiment and results of Transcriptase–Light detection. Chapter 5 gives details on implementing a polymorphic JavaScript malware; Chapter 6 concludes and discusses future work.

CHAPTER 2

Background

Malicious software is often referred as malware. This software is designed to intentionally harm or disturb the normal functioning of other computer programs. It performs malicious activities like extracting information or exploiting resources without the consent of the user. Such malicious programs either stop the execution of a benign program or execute along with the benign programs to carry out a malicious act. These acts might be stealing sensitive user information or destroying other files by deleting them or infecting them. Malware is primarily classified based on the method of infection and the internal structure.

2.1 Encrypted Malware

One of the most common methods used for malware detection is signature based detection. It looks for a particular pattern in a file. Details of the signature based detection technique is explained in the later section. Encrypted malware hides the actual content of the malware. The decryption code is also attached to the body of the malware. When this malware gets executed, decryption code runs first and decrypts the encrypted body of the malware. Consider the code snippet shown in code 2.1:

```
// ExecuteCommand.java
public class ExecuteCommand {
        public void runProcess(){
                Process proc = Runtime.getRuntime.exec("whoami");
                proc.waitFor();
                BufferedReader ip = new BufferedReader(new
                   InputStreamReader(proc.getInputStream()));
                String user=ip.readLine();
                changeUser(user);
        }
}
```

Code 2.1: Malicious Code Example,

Let's assume a signature of this malicious code is the "`Runtime.getRuntime.exec("whoami")`" string. Now we can use encryption such as the simple substitution cipher to hide this signature. After using a simple substitution cipher with the key `phqgiumeaylnofdxjkrcvstzwb`, the code snippet shown in code 2.1 gets transformed into the code shown in code 2.2.

```
// ExecuteCommand.java
xvhnaq qnprr iziqvciqdoopfg {
        xvhnaq sdag kvfxkdqirr(){
                xkdqirr xkdq = kvfcaoi.mickvfcaoi.iziq("tedpoa");
                xkdq.tpacudk();
                hvuuikigkipgik ax = fit hvuuikigkipgik(fit
                   afxvcrckipokipgik(xkdq.micafxvcrckipo()));
                rckafm vrik=ax.kipgnafi();
                qepfmivrik(vrik);
        }
```

Code 2.2: Encrypted Malicious Code

In this newly derived code, the signature detection method will not be able to find the original signature and the malware file will remain undetected. Figure 1 shows

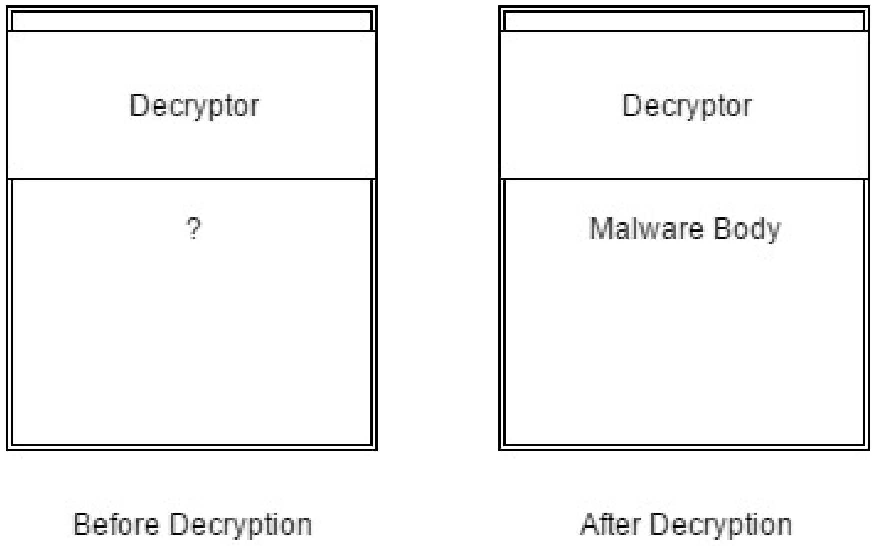


Figure 1: Encrypted Malware

the structure of encrypted malware.

2.2 Polymorphic Malware

In encrypted malware, the decryption body is the same for all malware files from the same family of encrypted malware. That is why encrypted malware is vulnerable to detection based on their decryption body. Polymorphic malware addresses this issue by generating a different version of the decryption body [8]. Polymorphic malware makes it difficult for anti-virus detection programs to detect an infected file by creating a different signature for each infected file from the same family [9].

2.3 Metamorphic Malware

In metamorphic malware, the internal body structure of the malware is changed on each infection by retaining the original functionality of the malware [10]. Let us discuss few approaches that are used to achieve morphed versions of the malware.

2.3.1 Instruction Substitution

In this approach, an instruction is replaced with other instructions that perform a functionally equivalent operation as the original instruction. For example:

```
var a = 100;
var b = a - a;
console.log(b);
> 0
```

Code 2.3: Instruction Substitution 1

```
var a = 100;
var c = a & 0x000;
console.log(c);
> 0
```

Code 2.4: Instruction Substitution 2

Here, the value of `b` is calculated by subtracting the value of `a` from itself. We can substitute this instruction with an AND operation of the value of zero. This new instruction will give us the same result as the subtraction operation.

2.3.2 Garbage Code Insertion

In this method, a variable amount of garbage code is inserted in between two instructions. This garbage code gets executed, but it does not affect the original functionality of the code. For example:

```
var a = 100;
var z = 6;        //garbage code
var d = z + 6;    //garbage code
z = z * 4;        //garbage code
d = z--;          //garbage code
var b = a + 1;
```

Code 2.5: Garbage Code Insertion

Here, the main goal is to find the value of variable `b`. However, multiple instructions are inserted in between the assignment to `a` and the assignment to `b` to generate multiple copies of the code.

2.3.3 Subroutine Permutation

Malware writers make use of subroutines to generate morphed versions of malware. They will reorder all function definitions in the code for each version of the malware. This will give each malware a new structure by maintaining the original functionality. A malware writer can create $n!$ morphed versions of malware if the original malware has n subroutines. For an example:

```
function A(userName) {
 var greeting = "Good Morning";
 console.log(greeting+userName);
}
function B{userID) {
 var message = "is busy!";
 console.log(userID+message);
}
main() {
 A("Jon");
 B(2204);
}
```

Code 2.6: Subroutine Permutation Code 1

```
function B(userID) {
 var message = " is busy!";
 console.log(userID + message);
}
function A(userName) {
 var greeting = "Good Morning ";
 console.log(greeting + userName);
}
main() {
 A("Jon");
 B(2204);
}
```

Code 2.7: Subroutine Permutation Code 2

Here, both the code snippets shown code 2.6 and code 2.7 perform the same operation, but they maintain different internal structure.

2.3.4 Independent Instruction Recording

In this method, instructions that do not depend on previous instructions are permuted to generate morphed versions of malware. It is a similar approach to subroutine permutation. If we have n independent instructions, then we can generate $n!$ morphed versions of the malware. This gives us a large number of morphed versions of the same malware [9]. For example:

```
1  var a = "Hello";
2  var b = "World";
3  var c = a + b;
```

Code 2.8: Independent Instruction Recording

Here, instruction 1 and 2 are independent of each other. Hence, we can alter an ordering of these two instructions.

2.4 Detection Techniques

We can divide malware detection techniques into two broad categories: static detection and dynamic detection methods. In static detection, we try to detect malware without actually executing it. Static detection systems focus on static properties of malware such as the structure of the malware. In dynamic detection, we run malware in a controlled environment. Dynamic detection systems measure the change in the system while malware is running. Some of the commonly used malware detection techniques are listed below:

2.4.1 Signature Based Detection

Signature based detection methods work in 2 phases. First, they extract the signature of a given file. The signature of a file is a particular sequence of bytes that uniquely identifies that file. In the second phase, it compares the extracted signature with the database of signatures constructed from known malware [11]. Based on the comparison result, it identifies the given file as benign or malicious. Anti-virus software companies maintain databases of signatures of malware, which they update whenever new malware is found.

It is a fast method to detect malware, and it is effective on most common malware. However, signature detection faces many challenges. The major drawback of this method is in detecting previously unknown malware. It cannot detect malware if the signature is not known previously. Most polymorphic and metamorphic malware is able to evade signature based detection due to its morphing ability [12].

2.4.2 Change Detection

All malware lives somewhere in the system. If we can detect a change in the system, we can detect malware [13]. If a file gets changed, then it might have been infected with malware. This can be achieved by computing and storing the hash values of all files in a system. We can periodically recompute the hashes to check whether a file is infected or not. The advantage of this method is there will be virtually no false negatives. Also, we can detect previously unknown malware with this approach.

The disadvantages of this method is that many files change frequently. This will result in too many false positives, which will eventually become a heavy burden on an administrator.

2.4.3 Anomaly Detection

In this method, a system is monitored for ‘‘unusual’’ or ‘‘potential malicious’’ activities [13]. Potential activities can be listed as:

- File changes in an unexpected way
- System misbehaves in some way
- Unexpected network activity
- Unexpected file access activity
- Resource consumption changes or requests for new resources

However, to achieve this, we need to define a normal behavior for the system. Also, we have to consider that normal behavior can change over a period. The major benefit of this system is that it can detect previously unknown malware. Disadvantages of this method are

1. Need to define the normal behavior for a system

2. The intruder or malware writer can make unexpected behavior look normal

2.4.4 Machine Learning Based Detection

Machine learning-based detection techniques are gaining popularity. In this approach, a model is trained on the features of malware. This trained model is later used for evaluating a score for a new file. This score is used to identify a benign or malicious file. The Hidden Markov Model (HMM) algorithm is widely used for malware detection [14]. In this method, an HMM is trained on the statistical features of malware such as the opcodes. The new file is scored against this model. If the score crosses a certain threshold value, then we identify the new file as a malicious file.

2.5 JavaScript Obfuscation Techniques

JavaScript obfuscation techniques make JavaScript code difficult to understand. This approach is used in commercial systems to prevent plagiarism of code. For example, yahoo.com uses obfuscated JavaScript code to prevent plagiarism[15]. But obfuscation has some side effects too. Malware writers use this technique to hide the malicious code from anti-virus software. Xu, Zhang, and Zhu in [15] examine different obfuscation techniques that are used for malicious purposes. This section summarizes the obfuscation techniques mentioned in [15].

2.5.1 Randomization Obfuscation

In this obfuscation technique, malware writers insert random whitespaces and comments in the code to change the structure of the code. The JavaScript interpreter ignores all whitespace and comments, so insertion of random whitespace and comments does not change the functionality. Also, malware writers change variable names and function names to hide the meaning. This method preserves the program semantics, but with a different static pattern.

```
function myFunction(uname) {
 console.log("Hello" + uname)
     ;
}
var user = "Jon Doe";
myFunction(user);
```

Code 2.9: Randomization Obfuscation Code 1

```
function
 Jb6IekyE12
(_0x2898x1)
{
//2skFUtW1Ff


 console.log("Hello" +
    _0x2898x1);}


var IIj77fc5P0 = "Jon Doe
   ";//7Xk08CztgT
//3ybP3EB5q2


Jb6IekyE12(IIj77fc5P0);
```

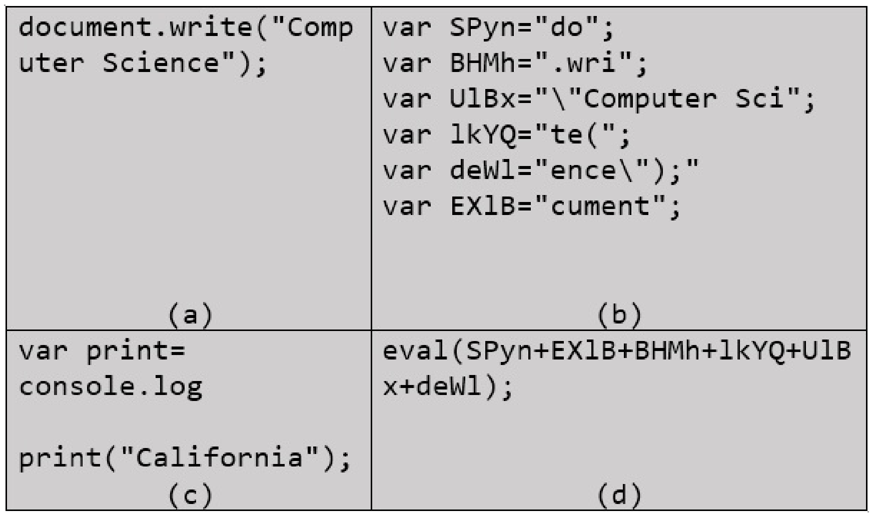
Code 2.10: Randomization Obfuscation Code 2

Code 2.10 is an examples of randomization obfuscation for code 2.9. In this example, the function name `myFunction` is changed to a string of random characters `Jb6IekyE12`. The function parameter `uname` is changed to `_0x2898x1`. The variable name `user` is changed to `IIj77fc5P0`. Also random comments are inserted to decrease the readability of the code.

2.5.2 Data Obfuscation

This technique focuses on converting variables or constants into the computation of multiple variables. A string is chopped into multiple chunks of smaller strings to change the structure of the code. Keyword substitution is also used to make code obfuscated. This splitting of the variable makes code harder to understand [3].

Figure 2 shows an example of data obfuscation. Figure 2(a) shows the original code.



```
document.write("Comp
uter Science");
```

(a)

```
var SPyn="do";
var BHMh=".wri";
var UlBx="\"Computer Sci";
var lkYQ="te(";
var deWl="ence\");"
var EXlB="cument";
```

(b)

```
var print=
console.log

print("California");
```

(c)

```
eval(SPyn+EXlB+BHMh+lkYQ+UlB
x+deWl);
```

(d)

Figure 2: Data Obfuscation

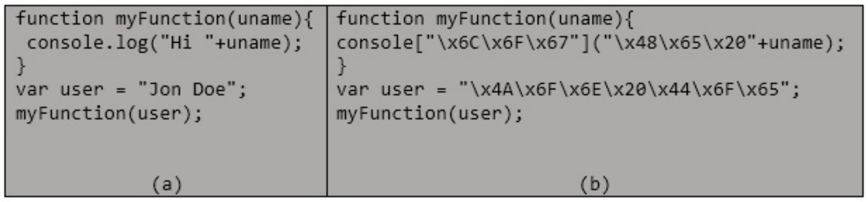
Figure 2(b) represents data obfuscation by splitting the original code into multiple string variables. Figure 2(d) shows the execution of obfuscated code by using eval function. Figure 2(c) shows keyword substitution. ‘‘console.log’’ is substituted with the ‘‘print’’ keyword.

2.5.3 Encoding Obfuscation

Malware writers use various methods to encode malicious code. Three common methods for encoding are:

- Convert code into escaped ASCII/unicode/hex characters
- Standard encryption and decryption functions
- Customized encoding functions

These encodings are used with the `document.write()` and `eval()` functions, which execute the code. In Figure 3, strings are converted into unicode characters. This encoding makes it difficult to search for a particular string. Also, it changes internal structure of the code by maintaining the semantics of the original code.



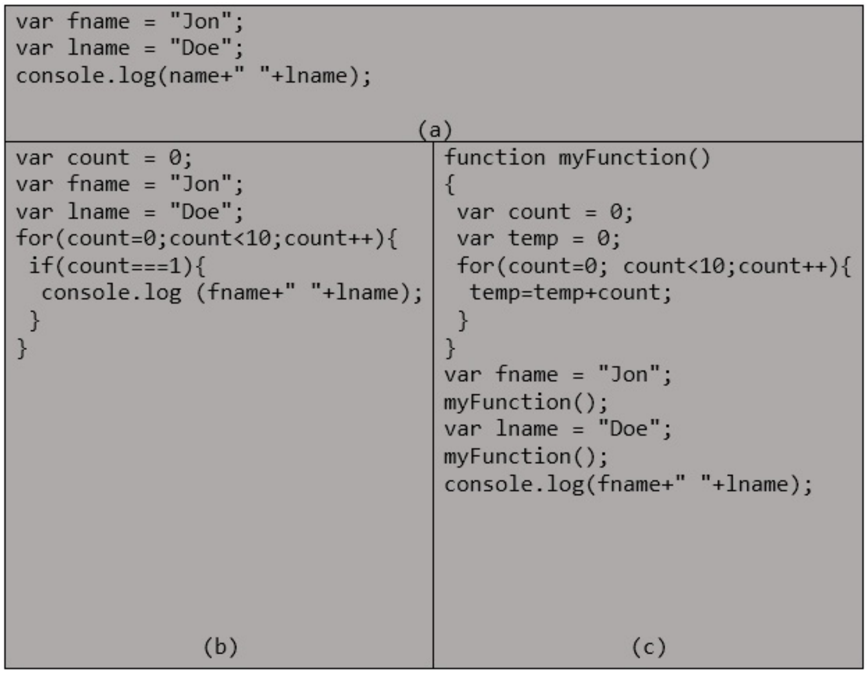
```
function myFunction(uname){
 console.log("Hi "+uname);
}
var user = "Jon Doe";
myFunction(user);
```

```
function myFunction(uname){
console["\x6C\x6F\x67"]("\x48\x65\x20"+uname);
}
var user = "\x4A\x6F\x6E\x20\x44\x6F\x65";
myFunction(user);
```

Figure 3: Encoding Obfuscation

2.5.4 Logic Structure Obfuscation

In this technique, malware writers manipulate the execution paths of the code by reconstructing the logical structure without affecting the original functionality. This is done by adding and changing conditional branches. Also, malware writers add dead code. This dead code has independent functionality, and it preserves the original functionality of the malware. In Figure 4, part (a) is the original code. Part (b) is an



```
var fname = "Jon";
var lname = "Doe";
console.log(name+" "+lname);
```

```
var count = 0;
var fname = "Jon";
var lname = "Doe";
for(count=0;count<10;count++){
 if(count===1){
  console.log (fname+" "+lname);
 }
}
```

```
function myFunction()
{
 var count = 0;
 var temp = 0;
 for(count=0; count<10;count++){
  temp=temp+count;
 }
}
var fname = "Jon";
myFunction();
var lname = "Doe";
myFunction();
console.log(fname+" "+lname);
```

Figure 4: Logic Structure Obfuscation

obfuscated version of the original code created by adding conditional branches in the code. Part (c) inserts dead code in the original code to make it obfuscated. Both part

(b) and part (c) maintains the semantics of the original code.

2.6 Transcriptase

The word “Transcriptase” is derived from biochemistry. During the transcription process, transcriptase is an enzyme that catalyzes the formation of ribonucleic acid (RNA) from a deoxyribonucleic acid (DNA) template. The term “Transcriptase” was selected by the original developer even though the link between transcription process and metamorphic JavaScript engine is fairly weak. This metamorphic malware infects the other JavaScript files in the same folder by appending the malicious code to a benign file.

An original version of Transcriptase was developed by Sperl [16] using a meta-language. This meta-language helps malware to evade signature based detection. The meta-language was designed to add to extra information required for morphing versions. A custom compiler is also attached to the body of the malware. The syntax for the meta-language is as follows [6]:

$$(Identifier|Restriction)Meta - Instruction$$

- Identifier: Unique identification number given to each statement in script
- Restriction: This set represents instructions that must be executed before current instruction
- Meta-Instruction: This instruction creates actual instruction in the script.

The combination of the identifier and the restriction in the meta-language gives malware flexibility to generate a permutation of the code. The meta-instruction is used to hide the actual malicious code. Code 2.11 shows an example of Transcriptase.

```
(100) var a = "#Hello#";;
(200) var b = "#World#";
(300) var count = #n1n#;
(400 | 100,200) console.log(a + b);
(500 | 300 ) var result = Math.sqr (count);
(600 | 500 ) console.log (result);
```

Code 2.11: Transcriptase Example

In code 2.11, instructions 100, 200 and 300 are independent instructions. Hence, the right side of ‘‘|’’ is empty. We can rearrange these instructions in 6! ways. Instruction 400 depends on instruction 100 and 200. Hence, the right side of ‘‘|’’ contains instructions identifier 100, 200 for instruction 400. Similarly, instruction 500 dependent on the value of count from instruction 300. Hence instruction 500 contains 300 as a restriction. Also, instruction 100,200 and 300 contains meta-characters. These symbols are introduced in the instruction as a part of the meta-language. The custom compiler takes care of the following operations:

- Permutation of instructions
- Parsing and Interpretation of meta-characters
- Code generation

An advanced version of Transcriptase is implemented using JavaScript obfuscation techniques and a homophonic cipher [7]. Various JavaScript obfuscation techniques help a malware writer to create metamorphic malware. Chapter 3 and chapter 5 explain the implementation details of ‘‘Transcriptase–Light’’, which is a polymorphic virus construction kit that we have designed as the part of this project. Transcriptase–Light is based on the encryption process described in [7]

CHAPTER 3

Transcriptase Light

As described in Chapter 2, Transcriptase is a metamorphic malware implemented using JavaScript. This section describes the method to implement Transcriptase–Light. Transcriptase–light uses similar techniques as Advanced Transcriptase [7] to generate a polymorphic JavaScript virus. Figure 5 shows the overall transformation process for Transcriptase–Light. Each layer of encryption in the process is discussed in subsequent sections. Transcriptase creation takes place in following three layers.

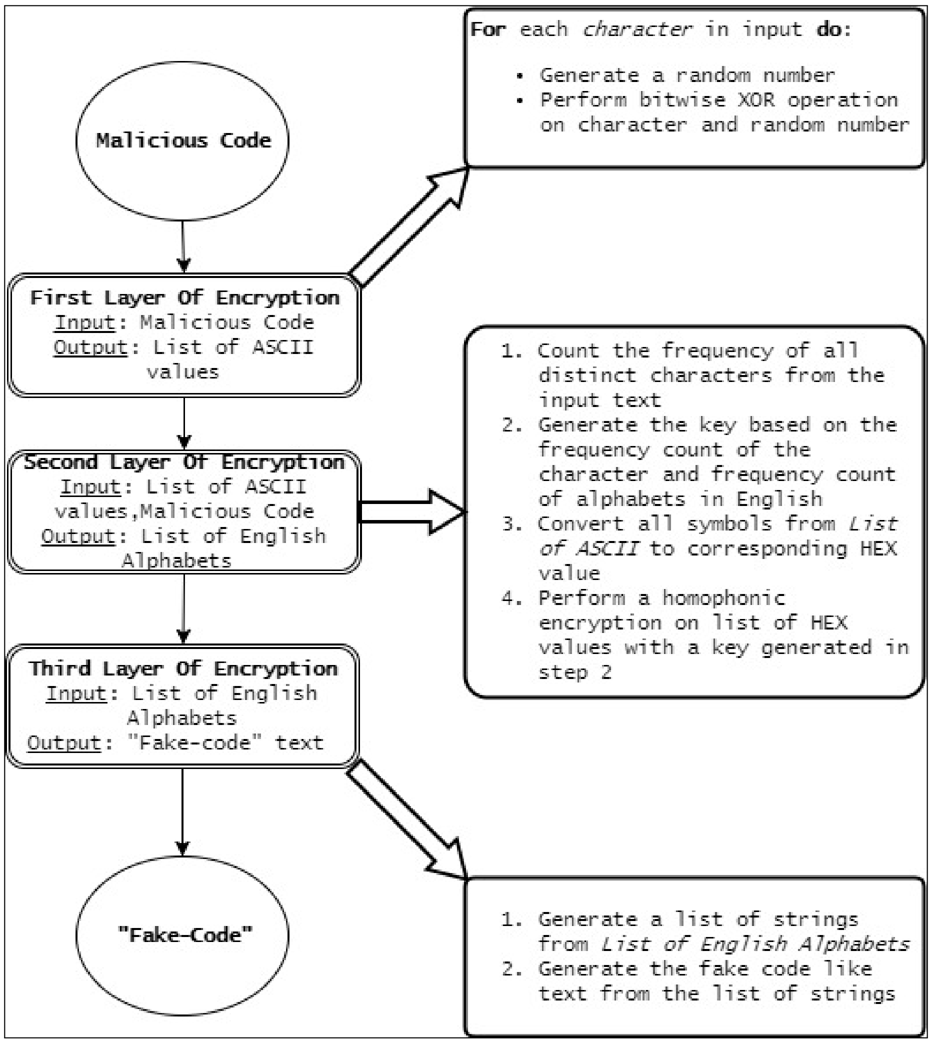

Malicious Code
First Layer Of Encryption
Input: Malicious Code
Output: List of ASCII values
Second Layer Of Encryption
Input: List of ASCII values,Malicious Code
Output: List of English Alphabets
Third Layer Of Encryption
Input: List of English Alphabets
Output: "Fake-code" text
"Fake-Code"
For each character in input do:
• Generate a random number
• Perform bitwise XOR operation on character and random number
1. Count the frequency of all distinct characters from the input text
2. Generate the key based on the frequency count of the character and frequency count of alphabets in English
3. Convert all symbols from List of ASCII to corresponding HEX value
4. Perform a homophonic encryption on list of HEX values with a key generated in step 2
1. Generate a list of strings from List of English Alphabets
2. Generate the fake code like text from the list of strings

Figure 5: Transcriptase–Light Overview

3.1 First Layer of Encryption

In this layer, we convert malicious code into random ascii symbols. We achieve this using bitwise a XOR operation. We generate a random key of length equal to the length of the malicious code. Then, we perform the bitwise operation on each symbol of the plaintext and each symbol of the key. We use this method to increase the entropy of a file. The entropy of a file is defined as randomness in a given set of data [17]. Increasing the entropy helps in the second layer of encryption. For an example, consider the following malicious code snippet shown in code 3.1:

```
var sys = require('sys')
var exec = require('child_process').exec;
var proc = exec("whoami", function (error, stdout, stderr) {
    sendServer(stdout);});
```

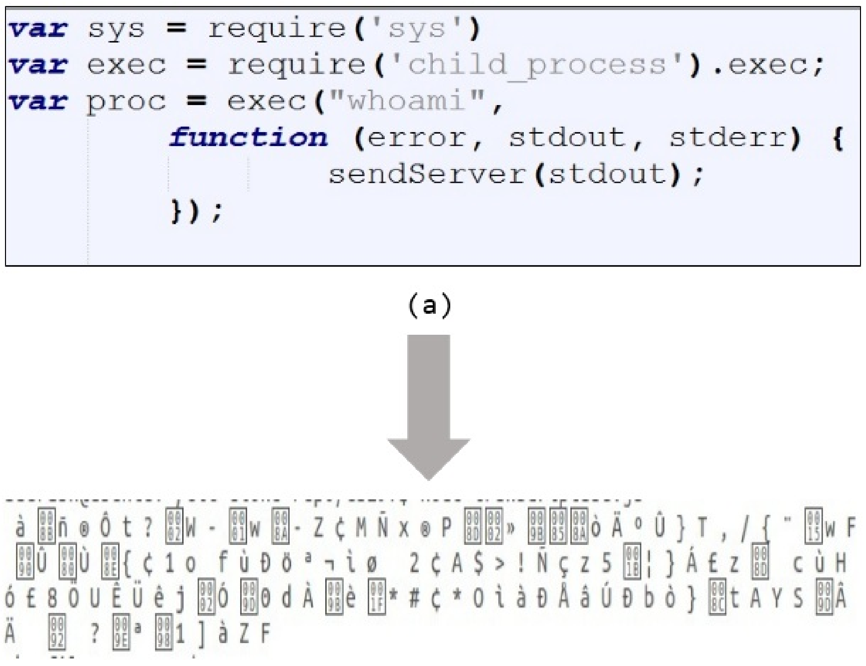
Code 3.1: Input Example

Figure 6 shows the initial transformation of the input code text. Each character of the input text is converted into a random ASCII value ranging from $0 - 255$.

3.2 Second Layer of Encryption

We have used the homophonic cipher to encrypt the values from the first layer of encryption. A homophonic cipher is a substitution cipher, where a given symbol may have multiple representations. It makes it easy to disguise the structural properties of the plain text. The following example shows the working of the homophonic cipher. Figure 7 represents the key used for encryption

In Figure 8, the letters of the plaintext are substituted with multiple ciphertext letters. For example, each occurrence of the letter "S" is substituted with different cipher letters such as "K", "Z", and "7" . Multiple substitutions of ciphertext letters make it hard to do cryptanalysis on the ciphertext.



```
var sys = require('sys')
var exec = require('child_process').exec;
var proc = exec("whoami",
        function (error, stdout, stderr) {
                sendServer(stdout);
        });
```

(a)

(b)

Figure 6: First Encryption Layer

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
M	U	E	N	Q	S	J	X	L	2	Q	5	W	F	9	T	B	0	K	A	V	6	C	I	D	1
P				R		3			O			4		Y				Z		8	G				
				H														7							

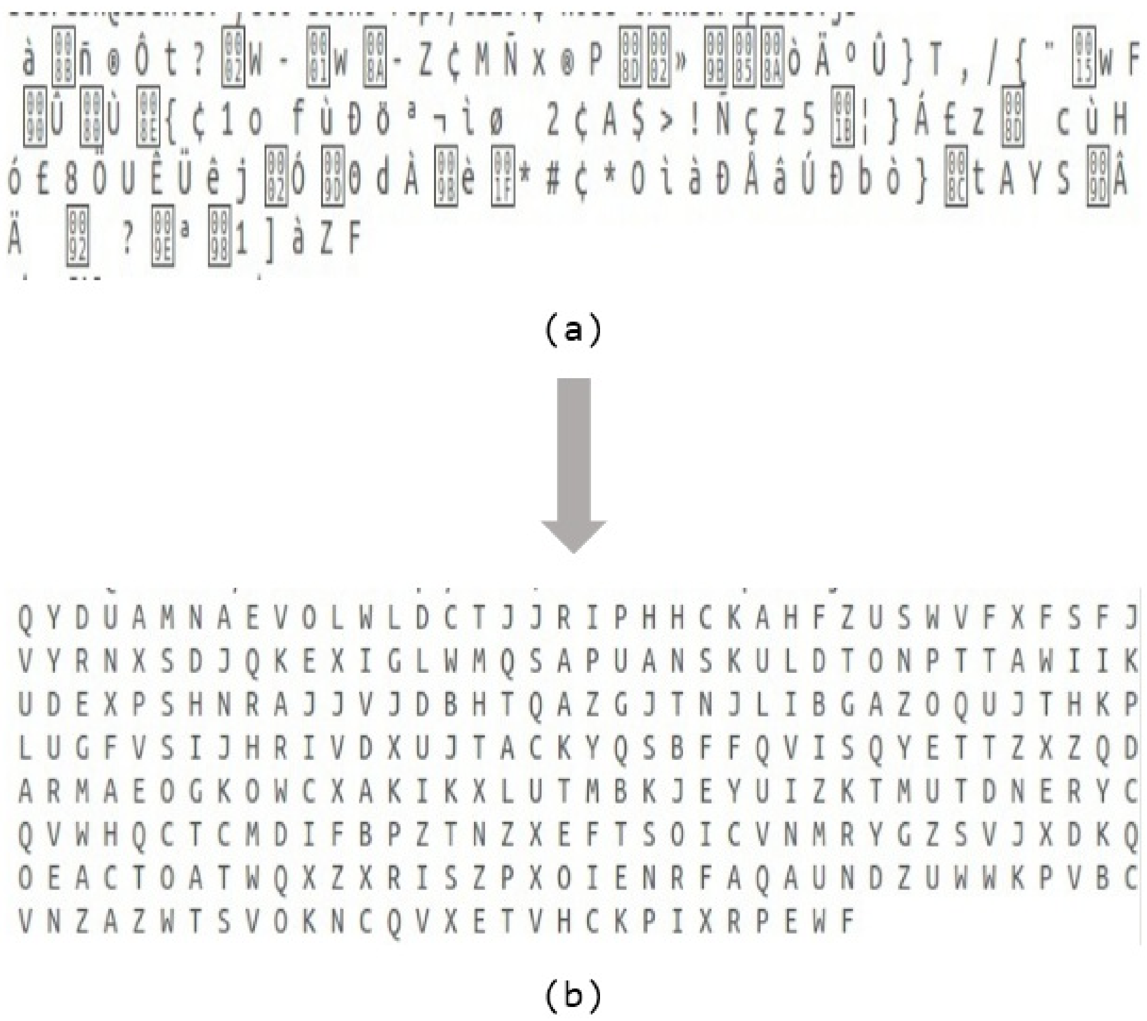
Figure 7: Homophonic Cipher Key

In Transcriptase–Light, a key for a homophonic cipher gets generated based on the frequency count of the plaintext symbol. The most frequent symbols in the

```
PLAINTEXT:   SAN JOSE STATE UNIVERSITY
CIPHERTEXT:  KPF 29ZQ 7AMAR VFL6H07LAD
```

Figure 8: Homophonic Cipher Encryption

plaintext get substituted by the most frequent letters in the English language. This method retains the statistical properties of generic English text. The plaintext for the cipher is generated from the hex values of each symbol. If the hex value for a given symbol is "C2", it gets converted to C and 2 as two individual characters. Each letter gets encrypted based on the previously generated homophonic cipher key. Figure 9



```
à 008B ñ ® Ô t ? 0082 W - 0081 w 008A - Z ¢ M Ñ x ® P 008D 0082 » 009B 0085 008A ò Ä º Û } T , / { " 0015 W F
0099 Û 0089 Ù 008E { ¢ 1 o  f ù Ð ö ª ¬ ì ø  2 ¢ A $ > ! Ñ ç z 5 001B ¦ } Á £ z 008D  c ù H
ó £ 8 Ö U Ê Ü ê j 0082 Ó 009D 0 d À 009B è 001F * # ¢ * O ì à Ð Å â Ú Ð b ò } 008C t A Y S 009D Â
Ä  0092  ? 009E ª 0098 1 ] à Z F
```

(a)

```
Q Y D U A M N A E V O L W L D C T J J R I P H H C K A H F Z U S W V F X F S F J
V Y R N X S D J Q K E X I G L W M Q S A P U A N S K U L D T O N P T T A W I I K
U D E X P S H N R A J J V J D B H T Q A Z G J T N J L I B G A Z O Q U J T H K P
L U G F V S I J H R I V D X U J T A C K Y Q S B F F Q V I S Q Y E T T Z X Z Q D
A R M A E O G K O W C X A K I K X L U T M B K J E Y U I Z K T M U T D N E R Y C
Q V W H Q C T C M D I F B P Z T N Z X E F T S O I C V N M R Y G Z S V J X D K Q
O E A C T O A T W Q X Z X R I S Z P X O I E N R F A Q A U N D Z U W W K P V B C
V N Z A Z W T S V O K N C Q V X E T V H C K P I X R P E W F
```

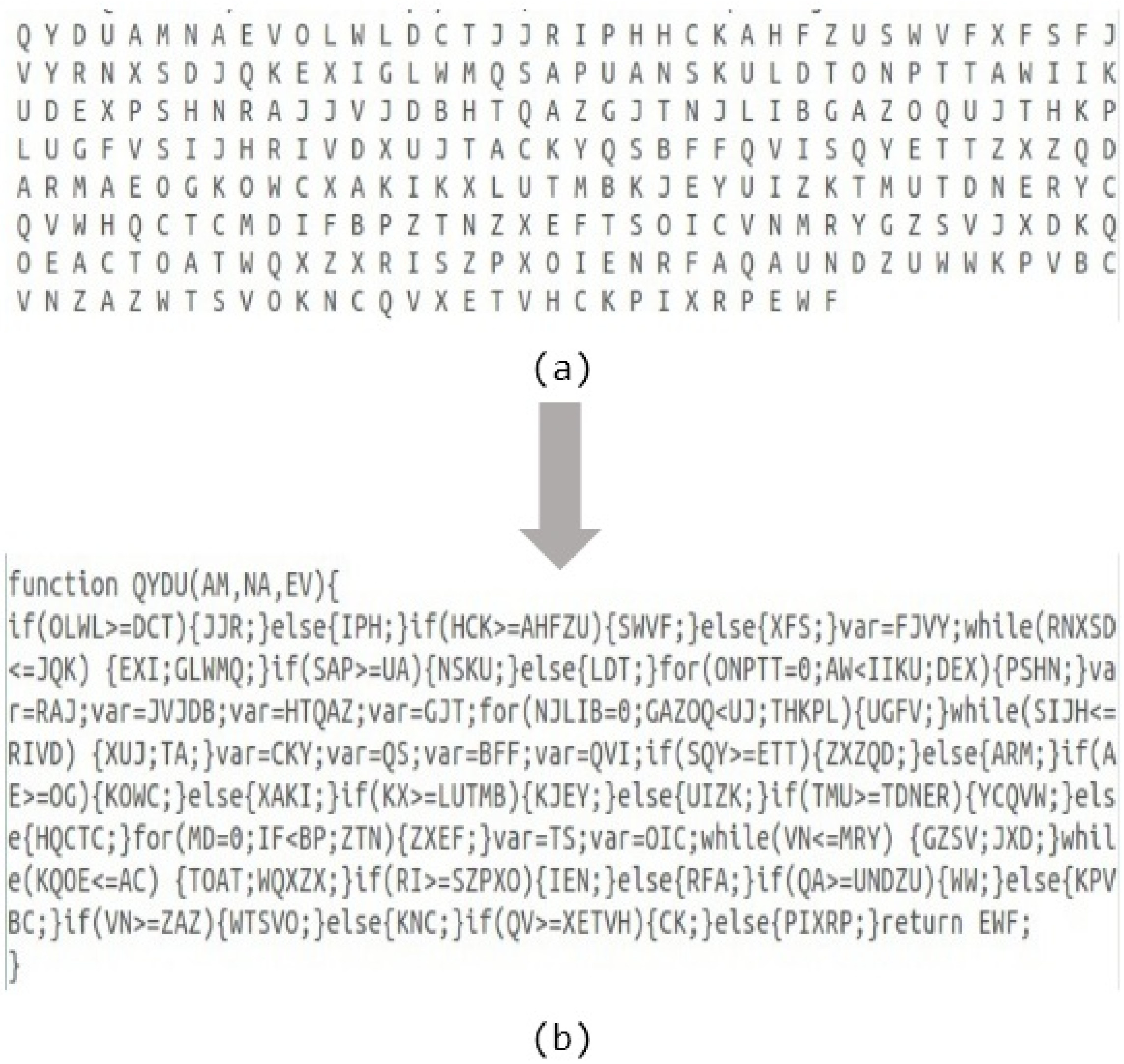
(b)

Figure 9: Second Encryption Layer: Cipher text

shows the output of the second encryption layer. The output of the first layer is used as the input to the second layer of encryption. After the second layer encryption, random ascii values get converted to English letters.

3.3 Third Layer of Encryption

In the third layer of encryption, cipher text from the previous encryption layer is transformed into code–like text. This ‘‘code’’ text resembles JavaScript code. This fake–code never gets executed. Transforming the strings of cipher letters into fake–code helps evade statistical analysis. The strings of letters from the second encryption layer are split into multiple words of different length. These words are used to make fake–code. Multiple control structures are used to construct a definition of a function. The function name and function parameters are also selected from cipher text words. Figure 10 represents an output of the third encryption layer. The fake–code is defined using one function definition. The function body is constructed using multiple control structure such as ‘‘FOR’’,‘‘WHILE’’, and ‘‘IF–ELSE’’. The function contains the original malicious code. This fake–code is then inserted into a benign file to infect it.



```
Q Y D U A M N A E V O L W L D C T J J R I P H H C K A H F Z U S W V F X F S F J
V Y R N X S D J Q K E X I G L W M Q S A P U A N S K U L D T O N P T T A W I I K
U D E X P S H N R A J J V J D B H T Q A Z G J T N J L I B G A Z O Q U J T H K P
L U G F V S I J H R I V D X U J T A C K Y Q S B F F Q V I S Q Y E T T Z X Z Q D
A R M A E O G K O W C X A K I K X L U T M B K J E Y U I Z K T M U T D N E R Y C
Q V W H Q C T C M D I F B P Z T N Z X E F T S O I C V N M R Y G Z S V J X D K Q
O E A C T O A T W Q X Z X R I S Z P X O I E N R F A Q A U N D Z U W W K P V B C
V N Z A Z W T S V O K N C Q V X E T V H C K P I X R P E W F
```

(a)

```
function QYDU(AM,NA,EV){
if(OLWL>=DCT){JJR;}else{IPH;}if(HCK>=AHFZU){SWVF;}else{XFS;}var=FJVY;while(RNXSD
<=JQK) {EXI;GLWMQ;}if(SAP>=UA){NSKU;}else{LDT;}for(ONPTT=0;AW<IIKU;DEX){PSHN;}va
r=RAJ;var=JVJDB;var=HTQAZ;var=GJT;for(NJLIB=0;GAZOQ<UJ;THKPL){UGFV;}while(SIJH<=
RIVD) {XUJ;TA;}var=CKY;var=QS;var=BFF;var=QVI;if(SQY>=ETT){ZXZQD;}else{ARM;}if(A
E>=OG){KOWC;}else{XAKI;}if(KX>=LUTMB){KJEY;}else{UIZK;}if(TMU>=TDNER){YCQVW;}els
e{HQCTC;}for(MD=0;IF<BP;ZTN){ZXEF;}var=TS;var=OIC;while(VN<=MRY) {GZSV;JXD;}whil
e(KQOE<=AC) {TOAT;WQXZX;}if(RI>=SZPXO){IEN;}else{RFA;}if(QA>=UNDZU){WW;}else{KPV
BC;}if(VN>=ZAZ){WTSVO;}else{KNC;}if(QV>=XETVH){CK;}else{PIXRP;}return EWF;
}
```

(b)

Figure 10: Third Encryption Layer: Fake–code like text

CHAPTER 4

Statistical Analysis of Transcriptase–Light

Signature–based detection is commonly used for malware detection. However, these methods are ineffective for metamorphic malware. Metamorphic malware uses various techniques discussed in Chapter 2 to alter its internal code structure. However, the original functionality remains the same. To maintain the original functionality of the malware, instructions are replaced with a functionally equivalent set of instructions. This keeps the overall statistical distribution of the instructions the same. In the previous study [6, 18], this property is used to detect the original version of Transcriptase. In the case of the advanced version of Transcriptase, the original code is transformed in ‘‘code’’–like text to hide the malicious content. This hides the statistical distribution of instructions from the original code. Experiments conducted in [7] shows the failure of detection based on instruction distribution statistics.

We designed an experiment to detect Transcriptase–Light based on the statistical distribution of the control structures in the code. To implement this experiment, we used Hidden Markov Models. We conducted our experiment with two HMM models. We trained the first model on malicious JavaScript files, which are essentially Transcriptase–Light. We trained our second model on a set of benign JavaScript files collected from different standard JavaScript libraries such as `d3.js` and `peg.js`.

4.1 Hidden Markov Model Methodology

A Hidden Markov Model works on pattern recognition to detect metamorphic malware. A lot of research is being conducted to enhance this technique [19, 20, 21]. The method to train HMM against a sequence of opcodes is described by Sridhara [22]. This trained HMM model is then used to score the new sequence. If the score of a new sequence is greater than a certain threshold value, we can consider that new sequence as a malicious sequence. HMMs are based on Markov chains.

4.1.1 Markov Chain

Consider two symbols X and Y in a sequence. Each symbol has a transition probability. The transition probability is defined as the probability of transition from one symbol to another symbol. The transition probability of the current symbol only depends on the current symbol. It does not depend on the previous symbols. This is known as a "Markov chain" of the first order. In the n^{th} order Markov chain, the transition probability of the current symbol depends on previous $n-1$ symbols in the sequence. Each symbol in a sequence is referred as the state. Consider an example of a Markov chain shown in Figure 11.

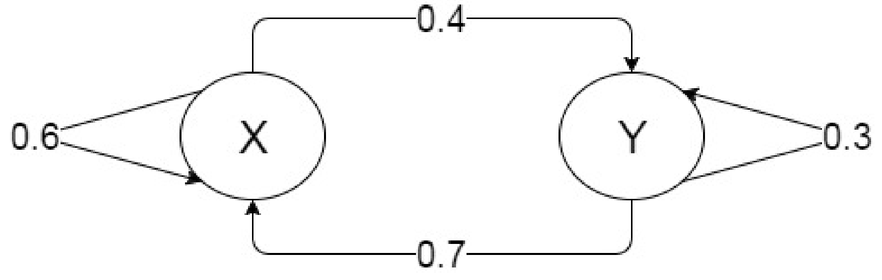


Figure 11: Markov Chain

In this example, there are two states "X" and "Y". The initial probability (i.e. a probability of a sequence starting in any symbol) is even (0.5). The transition probability from state X to state Y is 0.4, and state X to state X is 0.6. A Markov chain is useful in finding the likelihood of a sequence of symbols given the state transition probability matrix. For example, the probability of sequence (X, Y, Y, X, Y, X, X) is $0.5 * 0.4 * 0.3 * 0.7 * 0.4 * 0.7 * 0.6 = 0.007056$. All the rows in the transition probability matrix are row–stochastic[1].

[1]Row–stochastic: Summation of all the probabilities in the give row is 1

4.1.2 Hidden Markov Model

A Hidden Markov Model (HMM) is a statistical model in which the Markov process is “hidden”. That is, we cannot observe the underlying Markov process directly. We observe the probabilistic emissions related to the hidden Markov process. Based on these emissions, we can calculate the possible state path using a Viterbi algorithm. We can deduce the structure within the data using an HMM. In [1], notations used to define an HMM are as described in Table 1. To understand the

Table 1: HMM Notations [1]

notation		explanation
T	=	Length of the observation sequence
N	=	Number of states in the model
M	=	Number of observation symbols
Q	=	Distinct states of the Markov process $q_0, q_1, \ldots, q_{N-1}$
V	=	Possible observations, assumed to be $0, 1, \ldots, M-1$
A	=	State transition probabilities
B	=	Observation Probability Matrix
Π	=	Initial state distribution
O	=	Observation sequence $O_0, O_1, \ldots, O_{T-1}$

notations of the HMM and their use, consider an example shown in Figure 12. Here,

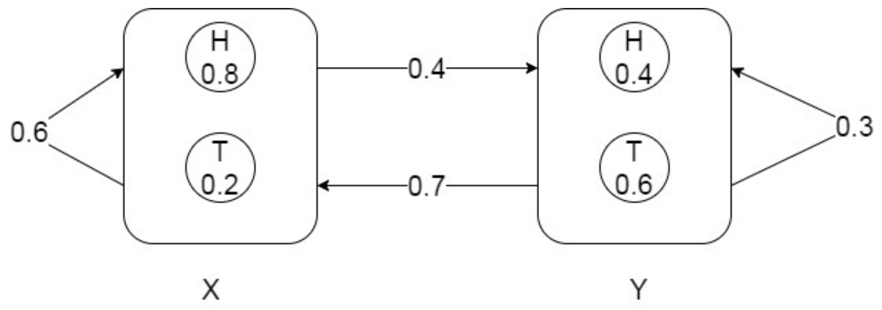


Figure 12: HMM Example

we have two coins X and Y i.e., $N = 2$. Each coin can emit either heads or tails with distinct probability. If the given observation sequence is $(H,\ T,\ T,\ H,\ T,\ H)$, then $T = 6$. We have two different symbols H and T. Hence, $M = 2$ and $V = H,\ T$. Matrix A is represented as:

$$A = \begin{bmatrix} 0.6 & 0.4 \\ 0.7 & 0.3 \end{bmatrix}$$

Where each row represents transition probabilities of the state, and each column represents next possible state. Each value $(i,\ j)$ in the matrix A represents transition probability from i^{th} state to j^{th} state. Matrix B defines the observation probability of the symbols for each state. Each row in matrix B corresponds to a particular state, and each column represents observation probability of a particular symbol in that state.

$$B = \begin{bmatrix} 0.8 & 0.2 \\ 0.4 & 0.6 \end{bmatrix}$$

If the possibility of starting a Markov process in either state is equal, matrix π is defined as

$$\pi = \begin{bmatrix} 0.5 & 0.5 \end{bmatrix}$$

All the rows in matrix A, B and π are row stochastic, i.e. all the probabilities in that row sums up to 1. Using these values, we can define our HMM as following:

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

Figure 13 represents a generic hidden Markov Model.

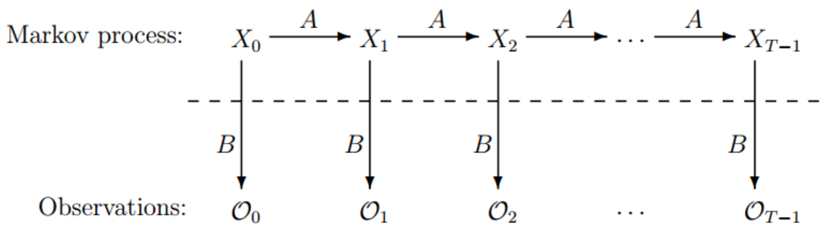


Figure 13: Hidden Markov Model[1]

4.2 Experiment

We designed an experiment to detect Transcript–Light using HMMs. In this experiment, we studied the statistical distribution of the keywords, assignment statements, and punctators in a JavaScript file. The notion behind designing this experiment was based on the construction of Transcriptase–Light. In Chapter 3, we learned how Transcriptase–Light transforms the malicious code into a ‘‘fake–code’’ like structure. This ‘‘fake–code’’ structure contains repetitive elements that follow the same pattern. This will not be the case with benign code or normal code. In the benign code, the logical structure of the code will differ. There will be a less repetitive elements than the Transcriptase–Light code. This gives the foundation for our experiment. In our experiment, we trained the following two different models to analyze the internal code structure of the file.

- Model trained against malicious JavaScript files
- Model trained against benign JavaScript files

We collected our benign data set using standard JavaScript files listed in Table 2. Since each JavaScript file mentioned in Table 2 is composed of multiple functions, we

Table 2: Benign Dataset

Library
buttons.bootstrap.js
d3.js
dataTables.responsive.js
dataTables.scoller.js
intro.js
jquery-3.2.0.min
moment.js
p5.js
peg.js
progressbar.js
responsive.bootstrap.js
responsive.foundation.js
svg.js
swipe.js
typed.js

split the file into multiple files containing a subset of the functions from the original file. We created 80 benign JavaScript files in total. We generated 100 malicious files using the technique discussed in Chapter 3. We trained our malicious HMM on 75 files and tested it against 25 files. For our benign HMM, we trained it on 60 files and tested it against 20 files. We used `Esprima` for tokenizing a JavaScript file. `Esprima` is a JavaScript parser written in JavaScript. `Esprima` performs lexical and syntactic analysis of a JavaScript program [23]. After tokenizing the program, we extracted the `type` of each statement to create a token sequence. Code 4.1 shows the example usage of `esprima.js`.

```
> var esprima = require('esprima');
> var program = 'function myFunction (uname){ console.log("User="+uname);}';
> esprima.tokenize(program)
[ { type: 'Keyword', value: 'function' },
  { type: 'Identifier', value: 'myFunction' },
  { type: 'Punctuator', value: '(' },
  { type: 'Identifier', value: 'uname' },
  { type: 'Punctuator', value: ')' },
  { type: 'Punctuator', value: '{' },
  { type: 'Identifier', value: 'console' },
  { type: 'Punctuator', value: '.' },
  { type: 'Identifier', value: 'log' },
  { type: 'Punctuator', value: '(' },
  { type: 'String', value: '"User="' },
  { type: 'Punctuator', value: '+' },
  { type: 'Identifier', value: 'uname' },
  { type: 'Punctuator', value: ')' },
  { type: 'Punctuator', value: ';' },
  { type: 'Punctuator', value: '}' } ]
>
```

Code 4.1: Esprima Demo

The results of the experiments are explained in the next section. We plotted Receiver Operating Characteristics (ROC) curve for each experiment to find the effectiveness of each classifier. The Area under the curve (AUC) is calculated to find the accuracy of each classifier.

4.3 Result

The ROC represents the correctness of the classifier. ROC is generated by plotting true positive (TPR) values on Y–axis and false positive rate (FPR) values on X–axis. For the malware detection, we define a true positive (TP), true negative (TN), false positive (FP), false negative (FN) as the following:

- TP: Malware file is correctly identified as a malware
- TN: Benign file is correctly identified as a benign file
- FP: Benign file is incorrectly identified as a malware
- FN: Malware file is incorrectly identified scored as a benign file

Based on these definitions, TPR and FPR are calculated as:

- $TPR = \frac{TP}{TP+FN}$
- $FPR = \frac{FP}{FP+TN}$

For our classifier, we build a ROC curve by plotting TPR and FPR values for all possible threshold values. The area under the curve (AUC) is calculated from the ROC curve to estimate the accuracy of the classifier [24]. The higher the value of the AUC, the better the performance of the classifier.

4.3.1 Malicious Model

We trained our HMM model on the token sequence extracted from the 75 malicious files. We generated four different models with a number of states (N) ranging from 2–5. We measure the performance of each model by plotting the ROC and calculating the AUC. In Figure 14, there is no clear separation between scores of benign files and malicious files. Therefore, we cannot identify the malware with any confidence.

Similarly, Figure 15, Figure 16, and Figure 17 scores for all benign files and malicious files are mixed together. There is no clear line of separation between the scores of benign samples and malicious samples. Hence, it is difficult to classify a

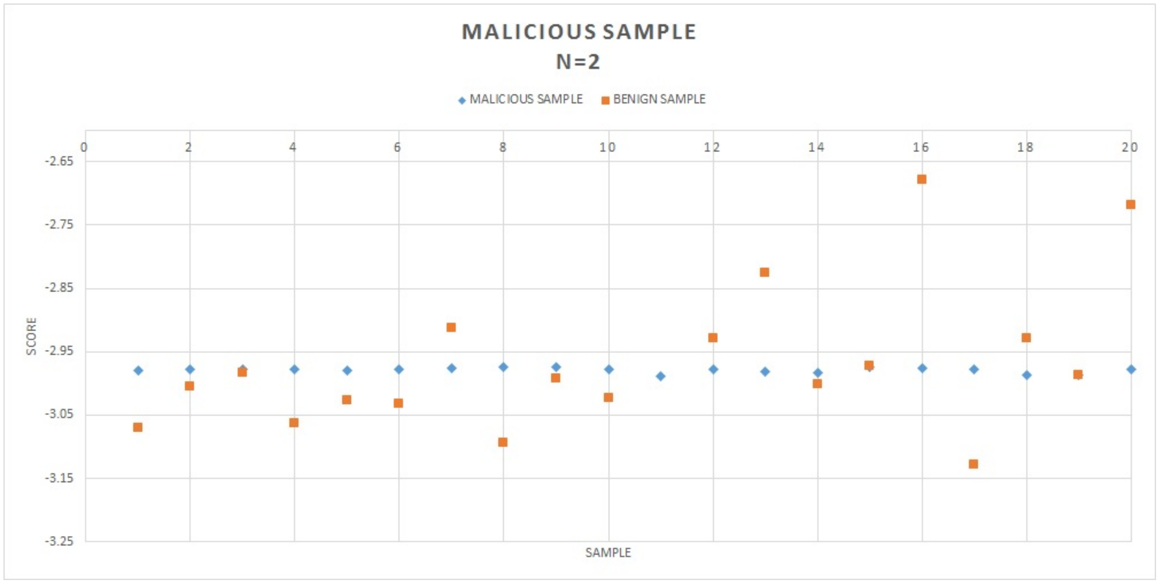


Figure 14: Scatter Plot for Malicious Model with N = 2

file as a malicious file. We observed that the performance of the classifier does not improve with an increase in the number of states in the HMM. Table 3 represents the AUC values for the corresponding HMM. The AUC value for a ROC curve should lie between the range 0.5–1.0. When the AUC value drops below 0.5, we can infer that when certain classifier is trained to identify type A entity, it is good at identifying type B entity. Hence, we can invert AUC scores i.e. subtracting the current AUC score from 1. The inverted score table 4 represents the $1 - AUC$ values from table 3. The inverted scores for the AUC in table 3 are given in table 4

Table 3: Area Under Curve(AUC) for Malicious Model

Number of States	Score to detect malicious file
2	0.634
3	0.295
4	0.346
5	0.522

We get the best performance result with two states in the HMM.

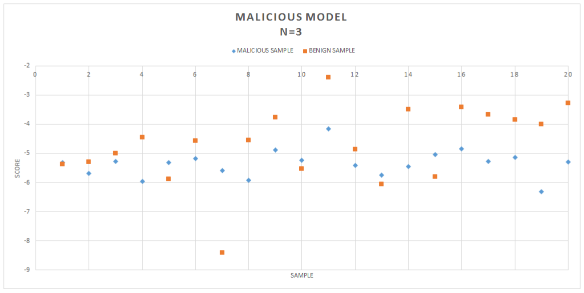


Figure 15: Scatter Plot for Malicious Model with N = 3

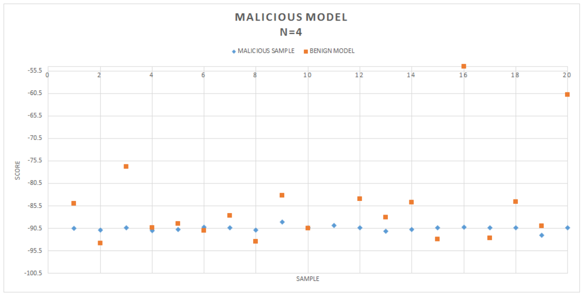


Figure 16: Scatter Plot for Malicious Model with N = 4

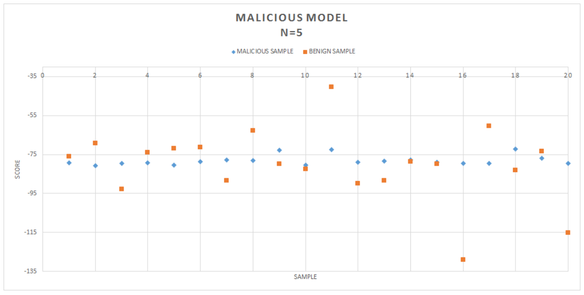


Figure 17: Scatter Plot for Malicious Model with N = 5

Table 4: Inverted Area Under Curve(1-AUC) for Malicious Model

Number of States	Score to detect benign file
2	0.366
3	0.706
4	0.654
5	0.478

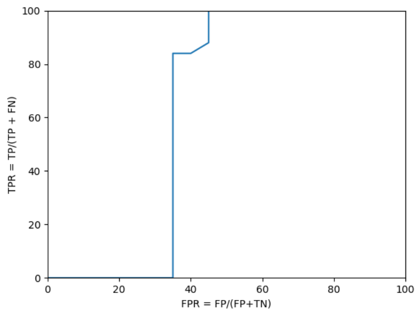


Figure 18: ROC curve for Malicious Model with N = 2

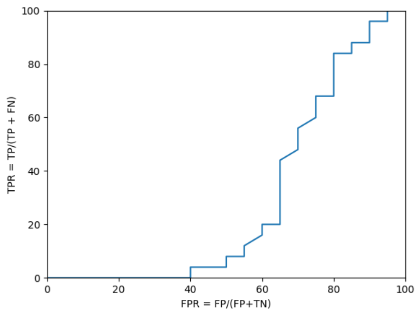


Figure 19: ROC curve for Malicious Model with N = 3

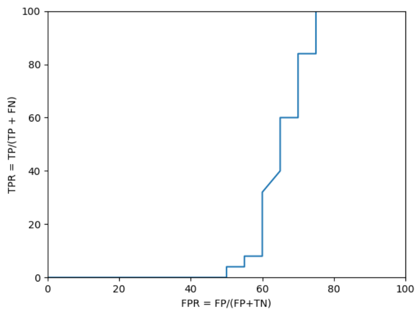


Figure 20: ROC curve for Malicious Model with N = 4

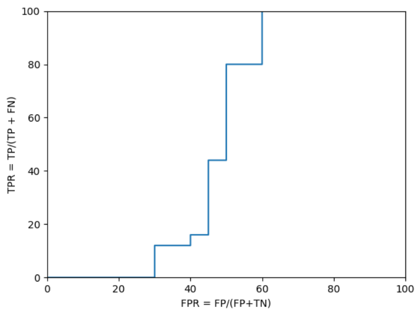


Figure 21: ROC curve for Malicious Model with N = 5

4.3.2 Benign Model

We trained our HMM model on the token sequence extracted from the 60 benign files. We generated four different models with number of states (N) ranging from 2–5. We measured the performance of each model by plotting the ROC and AUC. In this experiment, we tried to detect a benign file instead of malicious file from a given set of files. As the results from Table 5 indicate, our classifier performed poorly in correctly identifying the benign file. Table 5 shows the AUC values for models with multiple number of states. The classifier with two states yields better results than others. When the AUC value drops below 0.5, we can infer that when certain classifier is trained to identify type A entity, it is good at identifying type B entity. The inverted score table 6 represents the $1 - AUC$ values from table 5. The inverted

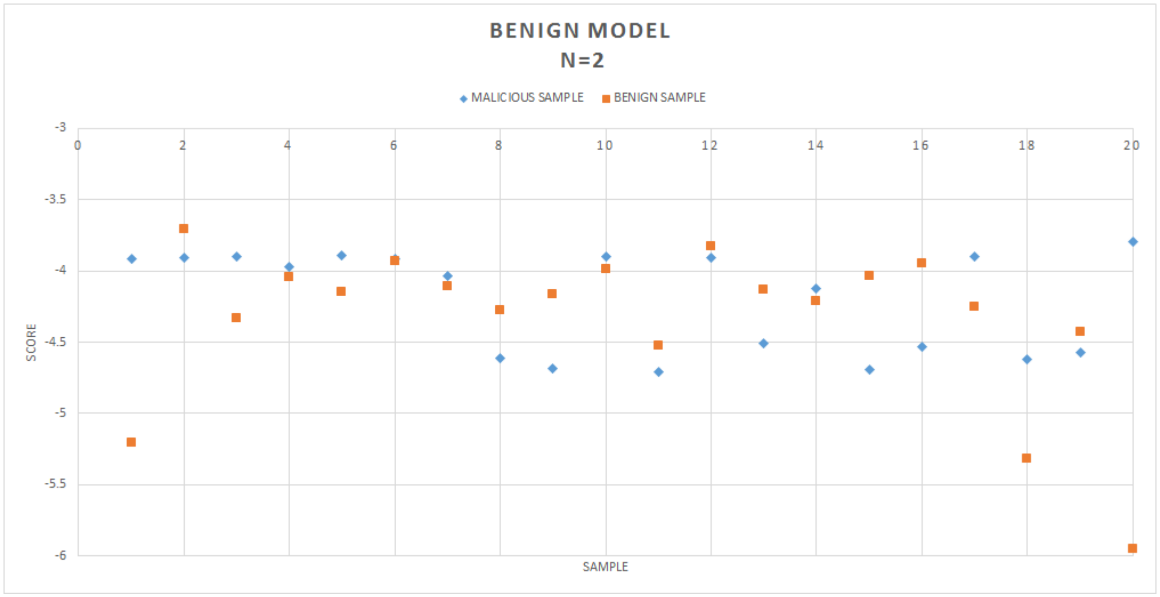


Figure 22: Scatter Plot for Benign Model with N = 2

scores for the AUC in table 5 are given in table 6

Table 5: Area Under Curve(AUC) for Benign Model

Number of States	Score to detect benign file
2	0.416
3	0.294
4	0.162
5	0.209

Table 6: Inverted Area Under Curve(1-AUC) for Benign Model

Number of States	Score to detect malicious file
2	0.584
3	0.706
4	0.830
5	0.791

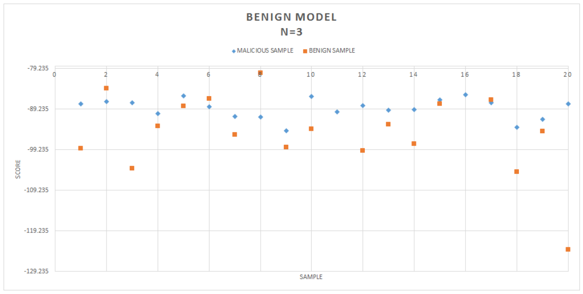


Figure 23: Scatter Plot for Benign Model with N = 3

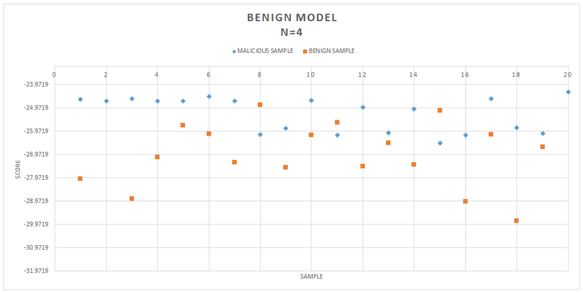


Figure 24: Scatter Plot for Benign Model with N = 5

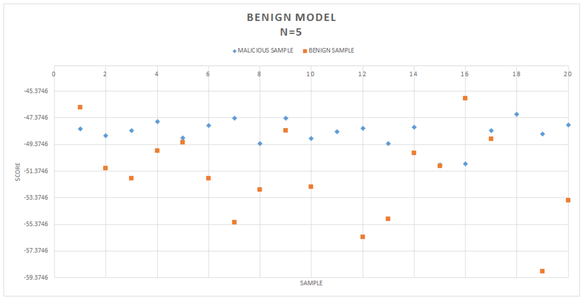


Figure 25: Scatter Plot for Benign Model with N = 5

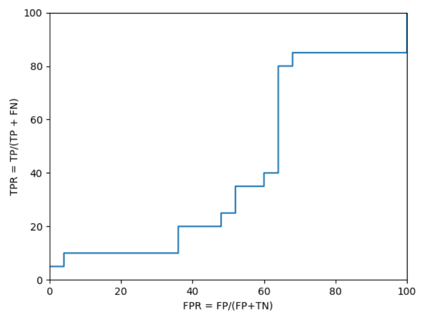


Figure 26: ROC curve for Benign Model with N = 2

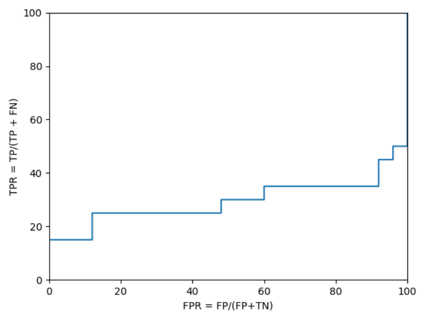


Figure 27: ROC curve for Benign Model with N = 3

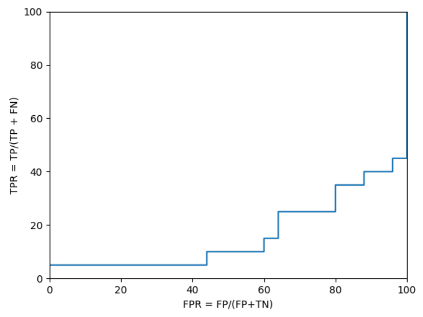


Figure 28: ROC curve for Benign Model with N = 4

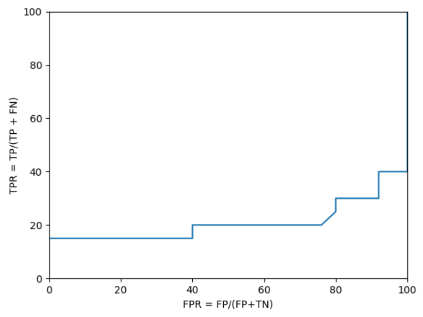


Figure 29: ROC curve for Benign Model with N = 5

CHAPTER 5

Polymorphic Decryption Code for Transcriptase–Light

eval is evil. Avoid it. eval has aliases. Don't use them

- Douglas Crockford [25]

In Chapter 4, we observed that a detector based on hidden Markov models failed in detecting Transcriptase–Light. The reason behind the failure of the detection was the three–layer encryption process of Transcriptase–Light. In Chapter 2, we introduced the concept of encrypted malware. Figure 1 shows the generic structure of the malware. In this type of malware, only the body of the malware is encrypted, and code for decryption gets appended to the encrypted body. When the encrypted malware gets executed, decryption code runs first and decrypts the encrypted body, and then the malicious code gets executed.

5.1 Dynamic Execution

In the case of JavaScript malware, the malware writer takes advantage of dynamic nature of the JavaScript. The dynamic characteristics of the JavaScript make it easy to turn text into the code at runtime. However, it has also raised security concerns. The `eval()` function of the JavaScript takes a string as an argument, parses it down to the source code and executes it immediately [25]. The `eval()` function has a wide range of usage. Some of the capabilities of the `eval()` are listed below:

1. Add or remove functionalities to the existing code
2. Add, remove, or modify the fields of existing objects
3. Install new libraries

Consider the examples shows in code 5.1 and code 5.2. Both codes shows in code 5.1 and code 5.2 are executed in the `node.js` terminal.

```
> eval("4+2");
6
> var a= 4;
> var b =2;
> eval(a+b);
6
```

Code 5.1: `eval()` Example 1

In the code 5.1, addition instruction for two integers is passed as a string to the `eval` function.

```
> var ip=[2,5,1,10,6,3]
> eval("function mySort(arr){var n= arr.length;var temp;
                   for(var i=0;i<n-1;i++){for(var j=0;j<n-i-1;j++)
                   {if(arr[j]>arr[j+1]){temp=arr[j]; arr[j]=arr[j
                       +1];
                   arr[j+1]=temp;}}}return arr;}var res=mySort(ip);
                   console.log(res);"
                  );
[ 1, 2, 3, 5, 6, 10 ]
> ip
[ 1, 2, 3, 5, 6, 10 ]
```

Code 5.2: `eval()` Example 2

In the example shown in code 5.2, the function definition for the function `mySort` is passed as an argument to the `eval` function. Also, in the first instruction, an array of random variables is assigned to the `ip` variable. The string passed to the `eval` method is accessing the variable declared outside the scope of the string and modifying it. After an execution of the `eval` function, the value of the variable `ip` is changed. Another way to achieve the same result as the `eval()` is with the use of `document.write()`. We can add `<script>` nodes to the `DOM` tree of the page using `document.write()`.

5.2 Transcriptase–Light Decryption

The decryption process for Transcriptase–Light is the reverse of the encryption process. The decryption process takes place in following 3 steps:

- Recover the all the string of random characters from the Transcriptase–Light function
- Decrypt the homophonic cipher
- Perform the `XOR` operation to recover the original malicious code

This decryption code is then converted into a long string and used as an argument for the `eval()` function as discussed in previous section.

The drawback of this method is the unencrypted decryption code. Instead of searching and detecting Transcriptase–Light, anti-virus software can use signature detection to identify the decryption code. If the decryption code is removed, Transcriptase–Light will never get execute. To tackle this problem, we introduce the JavaScript Obfuscation technique.

5.3 Decryption–Code Polymorphism

An easy way to avoid signature detection is to generate new signatures while maintaining the original functionality. This can be achieved by various techniques described in the Chapter 2. We used the following two methods to generate a polymorphic version of the Transcriptase–Light.

- Garbage Code Insertion
- Encoding Obfuscation

5.3.1 Garbage Code Insertion

We injected garbage code into the original decryption code. This method changed the structure of the original decryption code. However, the functionality remained the same. We inserted multiple methods whose job was to fetch the results from the google search engine based on some query and sort the results.

5.3.2 Encoding Obfuscation

We obfuscated the decryption code with the randomly inserted garbage code with the help of encoding obfuscation technique. In this technique, we used a hexadecimal representation of the character to obfuscate the original code.

CHAPTER 6

Conclusion and Future Work

In this project, our aim was to create the Transcriptase–Light polymorphic virus construction kit from scratch. We transformed the malicious code into ‘‘fake–code’’ like text to hide the malicious content from anti-virus detection techniques. We processed the original malicious code in three layers of encryption. In the first layer of encryption, we perform the bitwise `XOR` operation with each character of malicious code and a randomly generated number. In the second layer of encryption, we used the homophonic cipher to encrypt the output of the first layer. We created the list of English alphabets as an output of the second layer. In the last layer of transformation, we generated strings of random length from the list of alphabets and used them as variable names for the logical constructs such as `function-definition`, `IF-ELSE`, `FOR-loop`, and `WHILE-loop`.

We designed an experiment to do a statistical analysis of the malware. In this experiment, we analyzed the internal code structure of the malware. We tokenized the given the JavaScript file into the sequence of tokens with the help of `esprima.js`. We trained our Hidden Markov Models against the extracted sequences. We scored the set of malicious file and benign files with HMM models. Our experiments results showed that the Transcriptase–Light cannot be detected with the Hidden Markov Model technique. The reason behind the failure of the model was the third layer of encryption in Transcriptase–Light generation process.

After the careful evaluation of the Transcriptase–Light, we discovered that it is vulnerable to the signature detection if the method is applied to the decryption code of the malware. To avoid this pitfall, we generated the polymorphic versions of the decryption code. To achieve the polymorphism, we introduced a various amount of dead code in the decryption code. After inserting the dead code, we used JavaScript encoding method to obfuscated the decryption code. We used the dynamic nature of JavaScript to convert a text into source code with the `eval` function.

Future work for this project can be categorized into two tracks. The first track includes the future work on generation of Transcriptase–Light. We can measure the performance of the malware with the Vigenere cipher instead of the homophonic cipher in the second layer of the encryption process. The Vigenere cipher is also a form of polyalphabetic substitution cipher. In the third encryption layer, we can add more complicated and nested logical constructs. Another way to disguise the anti-virus detection system is by creating multiple templates for "fake–code" text. We can generate these templates by extracting the logical structure of functions from benign files. After extracting the template from the benign file, we can use that template with the list of strings (containing the malicious code information) to form a new "fake–code" like text.

The second track for the future work includes the improvement on detection technique. We can try to detect the malware by combining the two or more methods together. For an example, we can train the HMM model on token sequence as mentioned in this paper and opcode sequence as described in [6]. We can measure the performance of this method to detect a JavaScript malware.

We can build the call–graph for the given JavaScript to detect a potential malware. The call–graph represents the relationship between subroutines in a program. We can use the call–to find the procedures that never gets called[compiler construction]. Previous study in [26] has shown the possible method to detect a malware based on the call–graph similarity between two binary files. We can extend that method to detect the Transcriptast–Light. We can design and implement a browser plugin with functionality to construct call–graph and identify the procedure that never gets called.

LIST OF REFERENCES

[1] M. Stamp, ‘‘A revealing introduction to hidden Markov models,’’ 2015.

[2] M. Haverbeke, *Eloquent JavaScript: a modern introduction to programming.* No Starch Press, 2015.

[3] S. S. Borwankar, ‘‘CS200W Literature Review,’’ Fall 2016.

[4] G. O’Gorman and G. McDonald, ‘‘Ransomware: A growing menace,’’ Tech. Rep. [Online]. Available: http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/ransomware-a-growing-menace.pdf

[5] ‘‘Securtiy Threat Report 2013,’’ Tech. Rep. [Online]. Available: http://www.sophos.com/en-us/medialibrary/PDFs/other/sophossecuritythreatreport2013.pdf

[6] M. Musale, T. H. Austin, and M. Stamp, ‘‘Hunting for metamorphic JavaScript malware,’’ *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 2, pp. 89--102, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11416-014-0225-8

[7] F. Di Troia and C. A. Visaggio, T. H. Austin, and M. Stamp, ‘‘Advanced transcriptase for javascript malware,’’ in *2016 11th International Conference on Malicious and Unwanted Software (MALWARE)*, Oct 2016, pp. 1--8.

[8] N. S. Selamat, F. H. M. Ali, and N. A. A. Othman, ‘‘Polymorphic malware detection,’’ in *2016 6th International Conference on IT Convergence and Security (ICITCS)*, Sept 2016, pp. 1--5.

[9] S. I. Babak Bashari Rad, Maslin Masrom, ‘‘Camouflage in malware: from encryption to metamorphism,’’ *IJCSNS International Journal of Computer Science and Network Security*, vol. 12, no. 8, pp. 74--83, aug 2012.

[10] D. Lin and M. Stamp, ‘‘Hunting for undetectable metamorphic viruses,’’ *Journal in Computer Virology*, vol. 7, no. 3, pp. 201--214, 2011. [Online]. Available: http://dx.doi.org/10.1007/s11416-010-0148-y

[11] Z. Bazrafshan, H. Hashemi, S. M. H. Fard, and A. Hamzeh, ‘‘A survey on heuristic malware detection techniques,’’ in *The 5th Conference on Information and Knowledge Technology*, May 2013, pp. 113--120.

[12] G. Canfora, A. D. Sorbo, F. Mercaldo, and C. A. Visaggio, ‘‘Obfuscation techniques against signature-based detection: A case study,’’ in *2015 Mobile Systems Technologies Workshop (MST)*, May 2015, pp. 21--26.

[13] M. Stamp, ‘‘CS266, Fall 2015,’’ lecture Slides.

[14] M. Stamp, ‘‘CS266, Fall 2015,’’ Machine Learning with Application in Information Security.

[15] W. Xu, F. Zhang, and S. Zhu, ‘‘The power of obfuscation techniques in malicious JavaScript code: A measurement study,’’ in *2012 7th International Conference on Malicious and Unwanted Software*, Oct 2012, pp. 9--16.

[16] Sperl, ‘‘Original Transcriptase malware source ,’’ http://spth.virii.lu/Transcriptase.rar.

[17] A. Shamir, , A. Shamir, and N. V. Someren, ‘‘Playing hide and seek with stored keys,’’ in *Lecture Notes in Computer Science*, 1998, pp. 118--124.

[18] S. K. R. Javaji, ‘‘Firefox Add-On For Metamorphic JavaScript Malware Detection,’’ Master’s thesis, San Jose State University, San Jose, California, 2015.

[19] C. Annachhatre, T. H. Austin, and M. Stamp, ‘‘Hidden Markov Models for malware classification,’’ *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 2, pp. 59--73, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11416-014-0215-x

[20] T. H. Austin, E. Filiol, S. Josse, and M. Stamp, ‘‘Exploring hidden markov models for virus analysis: A semantic approach,’’ in *Proceedings of the 2013 46th Hawaii International Conference on System Sciences*, ser. HICSS ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 5039--5048. [Online]. Available: http://dx.doi.org/10.1109/HICSS.2013.217

[21] S. Attaluri, S. McGhee, and M. Stamp, ‘‘Profile hidden markov models and metamorphic virus detection,’’ *Journal in Computer Virology*, vol. 5, no. 2, pp. 151--169, 2009. [Online]. Available: http://dx.doi.org/10.1007/s11416-008-0105-1

[22] S. Madenur Sridhara and M. Stamp, ‘‘Metamorphic worm that carries its own morphing engine,’’ *J. Comput. Virol.*, vol. 9, no. 2, pp. 49--58, May 2013. [Online]. Available: http://dx.doi.org/10.1007/s11416-012-0174-z

[23] ‘‘Documentation on using Esprima,’’ 2017.

[24] A. P. Bradley, ‘‘The use of the area under the roc curve in the evaluation of machine learning algorithms,’’ *Pattern Recogn.*, vol. 30, no. 7, pp. 1145--1159, July 1997. [Online]. Available: http://dx.doi.org/10.1016/S0031-3203(96)00142-2

[25] G. Richards, C. Hammer, B. Burg, and J. Vitek, *The Eval That Men Do.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 52--78. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22655-7_4

[26] S. Shang, N. Zheng, J. Xu, M. Xu, and H. Zhang, ‘‘Detecting malware variants via function-call graph similarity,’’ in *2010 5th International Conference on Malicious and Unwanted Software*, Oct 2010, pp. 113--120.

APPENDIX

Dual Hidden Markov Model Experiment

A.1 Experiment

In this experiment, we try to categorize the given file into the malicious or the benign file. To achieve this, we trained two different HMMs: one based on the benign token sequence, and another based on the malicious token sequence. To train these model, we used 60 benign files and 80 malicious files. We made our testing set from 10 benign files and 10 malicious files. For each file in the testing set, we calculated the score for malicious HMM model and benign HMM model.

A.2 Results

Table A.7 shows two score values (i.e. benign model score and malicious model score) for the file in the testing dataset. Figure A.30 shows the distribution of the scores of testing files. Since we were not able to draw any solid conclusion from the results, we decided not to further pursue this approach.

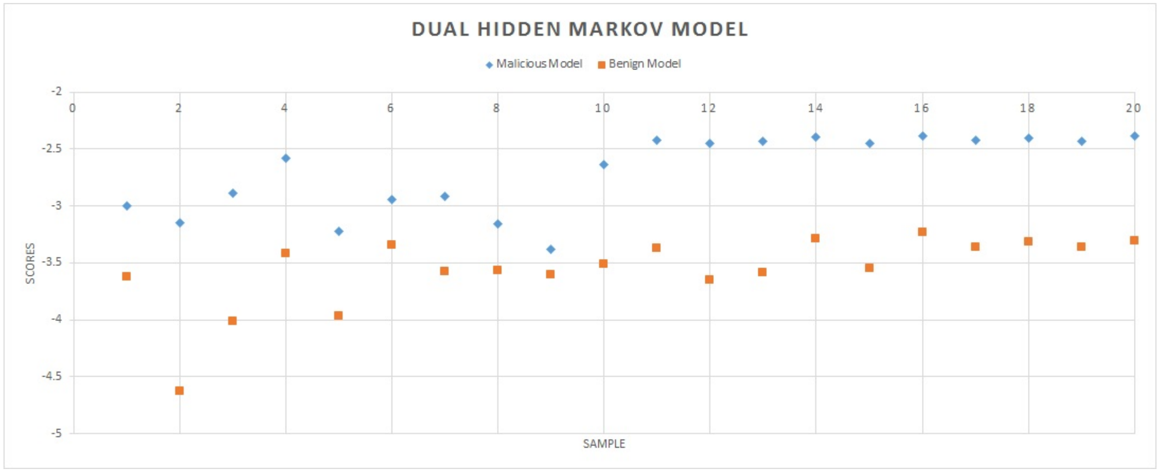


Figure A.30: Dual Markov Model Score Distribution

Table A.7: Dual Markov Model

No.	Test File Type	Malicious Model Score	Benign Model Score
1	Benign	-3.002050079	-3.624371133
2	Benign	-3.143231626	-4.623246804
3	Benign	-2.88843832	-4.011045738
4	Benign	-2.579372881	-3.420358922
5	Benign	-3.221981703	-3.966016104
6	Benign	-2.937954042	-3.343129034
7	Benign	-2.915655633	-3.579517625
8	Benign	-3.151845989	-3.561194521
9	Benign	-3.383500437	-3.606214313
10	Benign	-2.639056432	-3.512560091
11	Malicious	-2.420003093	-3.371775871
12	Malicious	-2.451921313	-3.652604043
13	Malicious	-2.4276303	-3.580825532
14	Malicious	-2.395147611	-3.284059666
15	Malicious	-2.447484536	-3.544918521
16	Malicious	-2.387315718	-3.234949182
17	Malicious	-2.421519549	-3.364419235
18	Malicious	-2.405848299	-3.314671518
19	Malicious	-2.433030139	-3.362563944
20	Malicious	-2.388197498	-3.304143124