Fall 2015

Metamorphic Java Engine

Sailee Choudhary
San Jose State University

Metamorphic Java Engine

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

Sailee Choudhary

December 2015

The Designated Project Committee Approves the Project Titled

Metamorphic Java Engine

by

Sailee Choudhary

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2015

Prof. Thomas Austin	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Prof. Younghee Park	Department of Computer Engineering

ABSTRACT

Metamorphic Java Engine

by Sailee Choudhary

Malware is a software program outlined to damage or perform other unwanted actions to a computer system. Metamorphic malware is a category of malignant software programs that has the ability to change its code as it propagates. A hidden Markov model (HMM) is a statistical model where the system is assumed to be a Markov process with unseen states. An HMM is based on the use of statistics to detect patterns, and hence in metamorphic virus detection. Previous work has been done in order to create morphing engines using LLVM-bytecode format.

This project includes the creation of a morphing engine for Java bytecode, using different code obfuscation techniques. The next aspect is to focus on detection techniques, specific HMM for validation of the created engine. The results presented show that HMM fail to detect the presence of morphing, provided specific set of rules have been followed while creation of metamorphic engine.

ACKNOWLEDGMENTS

I would like to thank Dr. Thomas Austin, for trusting me with the idea of this project and providing invaluable guidance throughout the project, helping me to find a way through all the hurdles. I would also like to express my special thanks and words of appreciation Dr. Chris Pollett and Dr. Younghee Park, for agreeing to be on my committee and providing me with their valuable feedback.

I would like to thank my parents for their constant support and motivation throughout the project. Last but not the least, I am immensely happy to express my gratitude towards all of my friends and seniors who braced and encouraged me to strive towards my goals.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

In today's world, we prefer to get things handily with no effort. The majority of transactions, forms, and other sensitive information is accessed from computers. It is pivotal to consider information security as an issue of supreme relevance. In today's world, computer systems are under threat from a variety of sources ranging from viruses, worms, hackers and phone freaks. Viruses are becoming difficult for detection and elimination [1].

Malware can be defined as a software designed by authors with an aim of damaging or abducting hosts, data, or network [2]. These malicious software can further be classified as viruses and worms. They self-replicate and spread copies of themselves.

Metamorphic malware is rewritten every time, so that the next version of the code is different. Metamorphic viruses use different manipulation techniques to modify the code structure. These techniques include subroutine permutation, insertion of jumps, equivalent instruction replacement, dead-code insertion, and transposition. There are certain bytecode manipulation libraries that can be used to obfuscate code using instruction substitution, dead-code insertion, etc. Some of these libraries are ASM, Javassist, BCEL, CGLib. These libraries allow us to create classes on the fly or modify existing classes directly in the binary form.

The detection techniques are also getting very sophisticated. The code revision makes it challenging for signature-based software programs to detect metamorphic viruses. The antivirus programs do not recognize that distinct iterations are the same program. To be specific, the paper [28] provides an evidence that metamorphic viruses

can evade signature-based detection if it has been built using specified guidelines. Hidden Markov models (HMMs) have been reliable for statistical analysis [26]. These models use machine learning techniques, and probability for each file can be determined against the trained model, to check for its classification.

Previously techniques have been included to achieve obfuscation in Java files which mainly include renaming of methods, fields and variables within a class file, with the intent to avoid reverse engineering [31]. Along with these mentioned techniques, some other techniques have been used which include replacing method body, addition of new fields, etc [22]. The purpose of this project is to develop a metamorphic engine using other code obfuscation techniques which include subroutine permutation, instruction permutation and dead-code insertion. The obfuscation is performed on Java class files using ASM bytecode manipulation library for achieving obfuscation. We test the effectiveness of obfuscation using HMMs. The goal of this project is to evade the HMM-based detection, so that any changes made to the files are undetectable.

This paper is arranged as follows:

- Chapter 2 consists of malware and its classification. It also describes the different methods used in detection of malware.
- Chapter 3 gives a detailed explanation at metamorphic malware and, different code obfuscation techniques.
- Chapter 4 provides accurate details about bytecode and bytecode manipulation library - ASM.
- Chapter 5 introduces Hidden Markov Models in-depth.
- Chapter 6 focuses on a detailed look at the design and implementation of this

project.

- Chapter 7 contains the experiments and the results of the experiments.
- Chapter 8 draws the conclusion based on our studies and explains the possible future scope.

CHAPTER 2

Malware and Malware Detection

2.1 Malware

As mentioned in Chapter 1, malware is a malicious piece of code. Malware can be further subdivided into more categories. Malware families [3] include worms, viruses, backdoor or trapdoor, Trojans, rabbit, spyware, adware, etc. There are various reasons for which malware can be written. Some may write it as pranks, others with the intent of affecting or stealing personal, confidential or business information. We will narrow our discussion to viruses in this section.

2.1.1 Virus

A virus relies on external entities to propagate itself [3]. The properties of a virus are [1]:

- It should clone itself.
- A program that can act as a transporter is needed.
- It is triggered by some external activity.
- Its cloning is restricted to the (virtual) system.

The most important issue faced by the malware producers, is to stretch the life of the malware. The main generations in virus development are:

2.1.1.1 Encryption

For hiding the functionality of the program, the easiest and the first method used by virus writers was encryption. It consists of two main parts [3]:

- the decryptor
- the encrypted body of the virus

The reasons for using encryption are as follows [3]:

1. **Prevention of static code analysis:** It involves disassembling the code to examine it for skeptical content such as blocks of the code. The use of encryption can cover skeptical instructions and so the use of static analysis can fail in encrypted viruses.
2. **Prolonging the dissection process:** Even though encryption makes the analysis of the code more difficult, it only adds a few extra minutes to the time required for analysis [3].
3. **To prevent tampering:** The process of modification or creation of new variants of the virus becomes complex if the virus is encrypted.
4. **Evading detection:** The previously encrypted viruses used an identical decryptor for all files infected by it, making it easier for detection [3]. However, more sophisticated viruses use self-changing encryption making detection impossible.

DOS virus Cascade used encryption [6]. The encryption method of cascade consists of XOR-ing every byte twice with variable values. The length of the program is one factor of determining variable value. The decryptor of the Cascade virus is as follows [6]:

```
ea      si, Start   ; position to decrypt (dynamically set)
mov     sp, 0682    ; length of encrypted body (1666 bytes)

Decrypt:
xor     [si],si     ; decryption key/counter 1
xor     [si],sp     ; decryption key/counter 2
inc     si          ; increment one counter
dec     sp          ; decrement the other
jnz     Decrypt     ; loop until all bytes are decrypted

Start:              ; Encrypted/Decrypted Virus Body
```

Figure 1: Cascade Decryptor [6]

2.1.1.2 Oligomorphism

Oligomorphic viruses are like encrypted viruses, but they differ in that, each new generation changes their decryptor [6].

Win95/Memorial could build 96 different decryptor patterns. This made detection based on the decryptor practically impossible. The oligomorphic properties were first used by Memorial. The code below is one instance for decryptor of the Memorial virus published in [6]:

```
mov     ebp,00405000h       ; select base
mov     ecx,0550h           ; this many bytes
lea     esi,[ebp+0000002E]  ; offset of "Start"
add     ecx,[ebp+00000029]  ; plus this many bytes
mov     al,[ebp+0000002D]   ; pick the first key

Decrypt:
nop                 ; junk
nop                 ; junk
xor     [esi],al    ; decrypt a byte
inc     esi         ; next byte
nop                 ; junk
inc     al          ; slide the key
dec     ecx         ; are there any more bytes to decrypt?
jnz     Decrypt     ; until all bytes are decrypted
jmp     Start       ; decryption done, execute body

; Data area

Start:
; encrypted/decrypted virus body
```

Figure 2: Memorial Decryptor [6]

The instructions can be rearranged to some extent and the decryptor can use diverse instructions for looping [6]. A slightly different version of the decryptor for the same virus is as follows [6] :

```
mov     ecx,0550h           ; this many bytes
mov     ebp,013BC000h       ; select base
lea     esi,[ebp+0000002E]  ; offset of "Start"
add     ecx,[ebp+00000029]  ; plus this many bytes
mov     al,[ebp+0000002D]   ; pick the first key

Decrypt:
nop                         ; junk
nop                         ; junk
xor     [esi],al            ; decrypt a byte
inc     esi                 ; next byte
nop                         ; junk
inc     al                  ; slide the key
loop    Decrypt             ; until all bytes are decrypted
jmp     Start               ; Decryption done, execute body

; Data area

Start:
; Encrypted/decrypted virus body
```

Figure 3: Different version of Memorial Decryptor [6]

2.1.1.3 Polymorphism

A polymorphic virus comprises of a decryptor and a virus body. The decryption routine takes command of the computer, and then decrypts the virus body. In addition to these, it consists of a third component, which is a mutation engine that develops arbitrary decryption routines [10]. The mutation engine as well as the virus body is encrypted. On execution of an infected program, the decryptor takes command of the computer and then decrypts the virus body as well as the mutation engine.

The first known polymorphic virus written was 1260 [6]. The virus inserts junk instructions in its decryptor and uses two keys to decrypt its body. The junk instructions are used for altering the appearance of the code. The extraction of simple

search strings from the code became difficult. The decryptor can change its size based on the junk instructions inserted. Example of an instance taken from [6] is as follows:

```
; Group 1  Prolog Instructions
inc     si        ; optional, variable junk
mov     ax,0E9B   ; set key 1
clc               ; optional, variable junk
mov     di,012A   ; offset of Start
nop               ; optional, variable junk
mov     cx,0571   ; this many bytes - key 2

; Group 2  Decryption Instructions
Decrypt:
xor     [di],cx   ; decrypt first word with key 2
sub     bx,dx     ; optional, variable junk
xor     bx,cx     ; optional, variable junk
sub     bx,ax     ; optional, variable junk
sub     bx,cx     ; optional, variable junk
nop               ; non-optional junk
xor     dx,cx     ; optional, variable junk
xor     [di],ax   ; decrypt first word with key 1

; Group 3  Decryption Instructions
inc     di        ; next byte
nop               ; non-optional junk
clc               ; optional, variable junk
inc     ax        ; slide key 1
; loop
loop    Decrypt   ; until all bytes are decrypted  slide key 2
; random padding up to 39 bytes
```

Figure 4: Instance of 1260 [6]

Each group contains five non-repeated junk instructions. Two nop instructions always appear. 1260 produces a high range of decryptors.

2.1.1.4 Metamorphism

Metamorphic viruses are acknowledged as body-polymorphic. These viruses do not use encryption, and hence the decryptor is not needed. It is similar to polymorphic viruses in a way that it uses a mutation engine for replicating itself. The behavior of the virus does not change, even if the structure, properties and code sequence changes with each iteration [11]. Metamorphic computer viruses avoid generating instances similar to the shape of their parent [6].

There are various techniques that can be used to create metamorphic viruses like dead code insertion, subroutine permutation, etc. We will focus more on metamorphic viruses and its detection evasion techniques in the Chapter 3.

2.2 Malware Detection

The approaches that can be used to detect viruses as described in [4] are:

2.2.1 Signature Detection

Signature based virus detection is the most common technique employed in antivirus software for identifying viruses. The signature of the file is computed according to the contents of the file and this signature is compared with the database of the signatures that are already present in the antivirus software. If a signature match is found, the file is considered as infected and needs to be repaired or deleted from the system.

For example, according to [6], the signature used for the W32/Beast virus is **83EB 0274 EBOE 740A 81EB 0301 0000**. After searching in the system for this signature in all files, we cannot be sure that it is infected, since a benign file can also

contain this signature. So, even if a matching signature exists, more proofs may be required to be assured, it is the W32/Beast virus [4].

Advantage of Signature based virus detection [4]:

- This type of malware detection technique is effective for known malware and whose signature can be easily extracted.
- It requires less work from users and administrator end as only signature files need to be updated.

Disadvantage of Signature based virus detection [4]:

- The signature files can become too lengthy and thus could result in slower scanning.
- Signature detection works for only known signatures. A file with a minute variation can cause a miss in the detection of an infected file.

2.2.2 Change Detection

If we find a change in the system, it can be an indication of infection. This is known as change detection. One method of implementing change detection is by using hash functions. For using this method, we first compute the hashes of the files and store them in the system securely. At regular intervals, we re-compute the hash values of the files and compare them with these stored hashes. If any changes have been found in the hash values, we can check them for infection.

Advantages of change detection [4]:

- There are no false negatives, as changes in hash values will always be detected.

- Previously unknown malware can be detected.

Disadvantages of change detection [4]:

- Many false positives can be encountered as many files on the system often change. This may be a burden for users and administrators.

2.2.3 Anomaly Detection

The purpose of anomaly detection is to search any unusual behavior, which could potentially be malicious [4]. The elementary challenge in anomaly detection is to decide what is normal and what is abnormal. Another challenge is that the definition of normal changes with the requirements of the system. So, this detection technique can give false positives.

Advantages of anomaly detection system [4]:

- Previously unknown malware can be detected.

Disadvantages of anomaly detection system [4]:

- It cannot work as a standalone system and hence is often combined with signature detection.

Apart from these general approaches that are used for virus detection, various machine learning techniques such as Hidden Markov Models (HMM) can be used for virus detection as mentioned in Chapter 1. Chapter 5 provides detailed understanding of HMM and its working.

CHAPTER 3

Metamorphic Malware

Metamorphism aims at changing the appearance of the virus, without changing its functionality. These viruses alter their code during propagation in the system. These viruses evade the risk of signature based detection. Metamorphic viruses use various metamorphic techniques like code permutation, dead-code insertion, etc.

Metamorphic virus as described in [6], does not have a decryptor and a consistent body as a polymorphic virus. Nonetheless, they are capable of creating a generation that looks different every time.

The units for model of the anatomy of the metamorphic engine published in [7] are as follows:

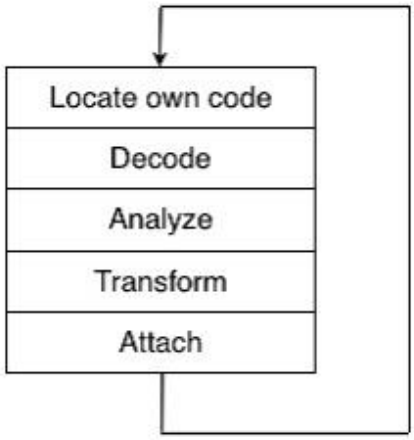

Figure 5: Units in metamorphic model [7]

Locate own code: It is important that the metamorphic viruses are able to discover their own code in different generations.

Decode: The information that is necessary for morphing must be decoded by

the engine. The engine must have some depiction of itself in order to know how to make transformations to it.

Analyze: In order to achieve metamorphosis precisely, some information must be accessible. The register liveliness information is needed for the performance of some transformations. If it is not available, the engine must be able to construct such information by itself.

Transform: Metamorphic code transformation without changing the functionality occurs at this point. Instruction blocks are replaced with equivalent blocks at this stage.

Attach: The last step is to adhere a recently transformed copy of the virus to a file. The execution of the units of the metamorphic engine might not be in order as in the figure. They can be randomly executed.

The book of "Art of Computer Virus Research and Defense" [1] denotes metamorphic malware as:

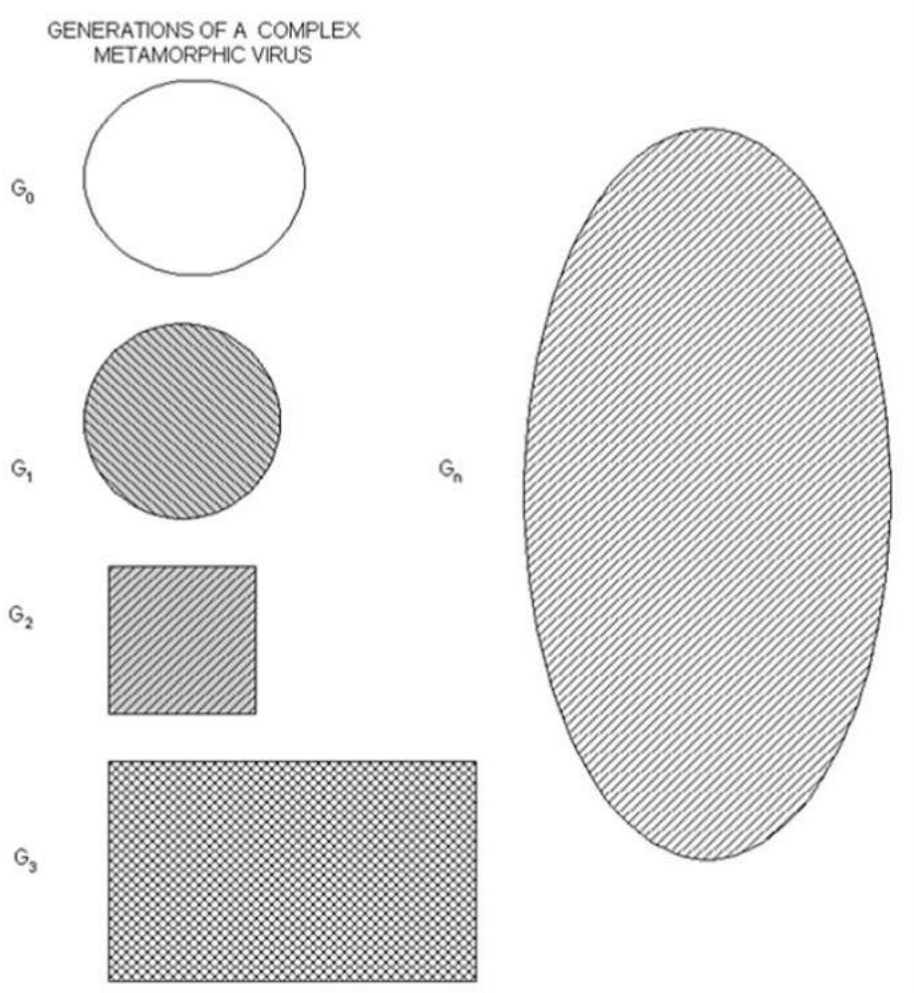

Figure 6: Metamorphic malware [1]

In the figure above, each subsequent generation G ($G_1, G_2, G_3....G_n$) changes its byte patterns. This makes it possible to change their appearance, but functionally remains the same. This will evade the signature based detection of the virus.

3.1 Techniques for generation of metamorphic malware

The techniques used to create metamorphic virus are as follows:

3.1.1 Garbage code insertion

Garbage code insertion is a methodology used by virus variants to evolve their code. By doing so, the appearance of their code changes, making the extraction of a usable hexadecimal string impractical [3]. The functionality of the code is not affected by garbage code insertion.

The Win95/Evol virus used dead code insertion. Two different versions of this virus appear distinct, even though their functions are same. The function of both of them is to transfer two double words into the memory address stated by esi. This makes it impossible to find a common sequence of bytes that can be used as a signature string. The following code taken from [3] shows dead-code insertion used in Win95/Evol virus:

```
a. An early generation:

C7060F000055        mov     dword ptr [esi],5500000Fh
C746048BEC5151      mov     dword ptr [esi+0004],5151EC8Bh

b. And one of its later generations:

BF0F000055          mov     edi,5500000Fh
893E                mov     [esi],edi
5F                  pop     edi
52                  push    edx
B640                mov     dh,40
BA8BEC5151          mov     edx,5151EC8Bh
53                  push    ebx
8BDA                mov     ebx,edx
895E04              mov     [esi+0004],ebx
```

Figure 7: Win95/Evol dead-code insertion [3]

3.1.2 Register swap

Register usage exchange relies on the use of different registers in each variant. Different variants use the same code, but the registers used are different. Win95/Reswap virus used this method for morphing. The two different generations of Win95/Regswap virus as published in [3] are as follows:

```
a.)
5A                      pop   edx
BF04000000              mov   edi,0004h
8BF5                    mov   esi,ebp
B80C000000              mov   eax,000Ch
81C288000000            add   edx,0088h
8B1A                    mov   ebx,[edx]
899C8618110000          mov   [esi+eax*4+00001118],ebx

b.)
58                      pop   eax
BB04000000              mov   ebx,0004h
8BD5                    mov   edx,ebp
BF0C000000              mov   edi,000Ch
81C088000000            add   eax,0088h
8B30                    mov   esi,[eax]
89B4BA18110000          mov   [edx+edi*4+00001118],esi
```

Figure 8: Win95/Evol register swap [3]

It can be seen that 'move edi, 004h' is substituted by the 'move ebx, 004h'. Similarly, 'move eax, 000Ch' is substituted by 'move edi, 000Ch'.

3.1.3 Subroutine Permutation

In subroutine permutation technique, the virus code remains consistent. The code is divided into frames, which are placed randomly. The branch instructions are used for connecting them and preserving the process flow. The flow of control always remains the same irrespective of the branching complexity.

TheWin32/Ghost and the Win95/Zperm were among the first viruses that used permutation techniques. The Win32/Ghost virus, re-positioned its own subroutines with each new generation. If there are n subroutines, the different possible generations of the virus is n!. The Win32/Ghost possessed 10 subroutines, thus the total number

of virus generations possible is 3628800. BadBoy is another example of a virus that uses this technique. It has 8 subroutines and hence the total number of generations possible are 40320. The BadBoy virus uses subroutine permutation. The subroutines on the left are the original subroutines whereas those on the right are the permuted ones. The figure referred from [6] shows subroutines of BadBoy:

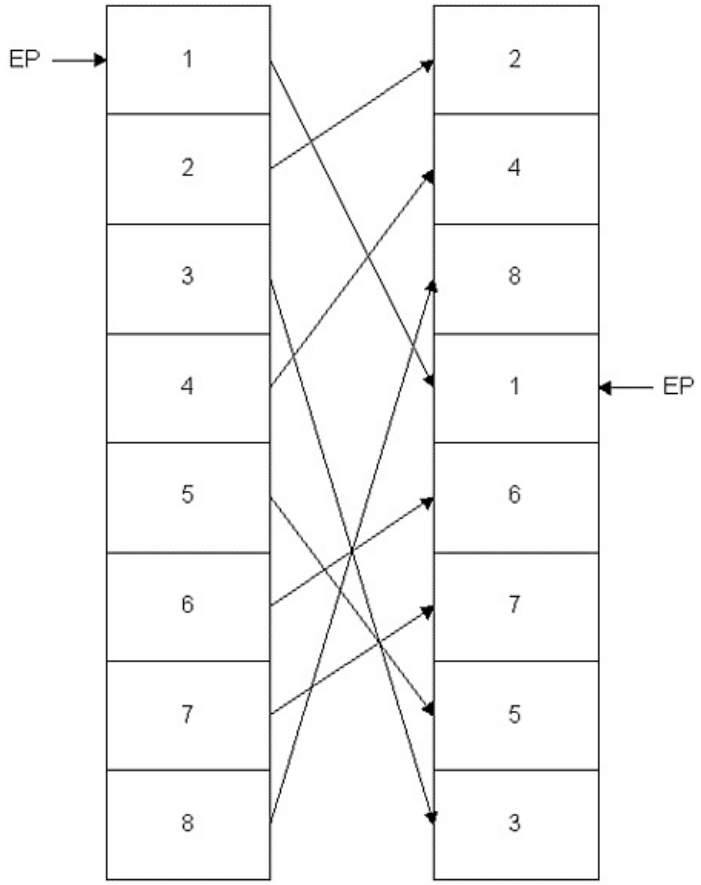

Figure 9: Badboy subroutines [6]

3.1.4 Random jump instructions

Metamorphism can be achieved by introducing jumps randomly in the code. The randomness of jumps has no effect on the execution of the virus. The Win95/Zperm virus inserts and deletes jump instructions in its own code. Every jump instruction points to an another instruction of the virus body. Zperm never generates a consistent body, to avoid the detection of the virus using a search string. For this, it also does not generate virus body, even in memory. The following figure referred from [6] shows the ZPerm jump instruction insertion:

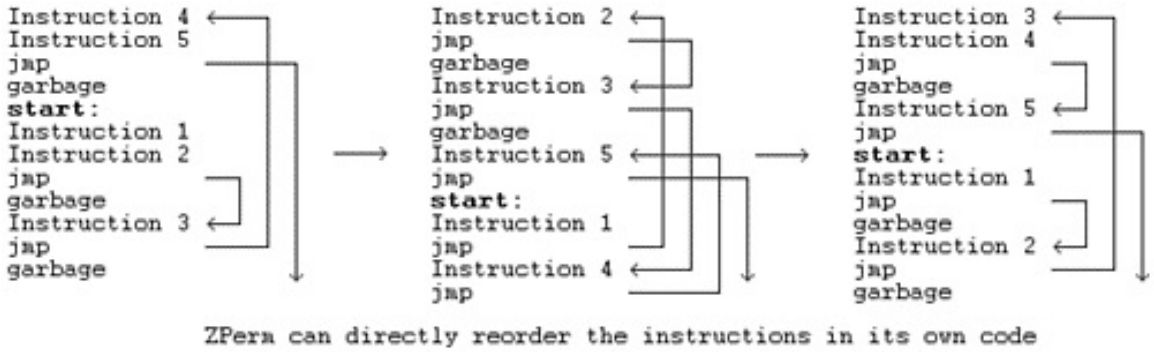

Figure 10: Zperm random jump instructions [6]

3.1.5 Equivalent code substitution

Some viruses are capable of replacing some instructions with other equivalent instructions. Win95/Zperm has the capability to perform instruction substitution. For example, "sub eax, eax" is replaced by its equivalent instruction, "xor eax, eax". Both these instructions are functionally the same, but have different opcodes.

Another example of a virus that uses equivalent code substitution is Win95/Zmist. The types of replacements implemented by this virus, as described in [8] are:

- branch conditions are reversed

- push/pop operations are used as a replacement for register moves
- alternative opcode encrypting

Win95/Bistro performs identical replacements. The code for Win95/Bistro as in [6] is as follows:

```
Before:
55          push    ebp
8BEC        mov     ebp, esp
8B7608      mov     esi, dword ptr [ebp + 08]
85F6        test    esi, esi
743B        je      401045
8B7E0C      mov     edi, dword ptr [ebp + 0c]
09FF        or      edi, edi
7434        je      401045
31D2        xor     edx, edx

After :
55          push    ebp
54          push    esp             register move replaced by push/pop
5D          pop     ebp             register move replaced by push/pop
8B7608      mov     esi, dword ptr [ebp + 08]
09F6        or      esi, esi     test/or interchange
743B        je      401045
8B7E0C      mov     edi, dword ptr [ebp + 0c]
85FF        test    edi, edi     test/or interchange
7434        je      401045
28D2        sub     edx, edx             xor/sub interchange
```

Figure 11: Win95/Bistro instruction replacement [6]

CHAPTER 4

Bytecode Manipulation

4.1 Java Classes

4.1.1 Structure of Class

A compiled Java class retains its structural information and symbols from the source code. As described in [21], the compiled class consists of:

- The section that describes the name, the modifiers, the annotations, the interfaces and the super class of the class.
- A section for each field, which describes the name, modifiers, type and annotations of the field.
- Each constructor as well as a method that is declared within the class have a dedicated section. This section comprises of the return type, modifiers, the annotations, etc. for the constructor or method. Along with this, it also includes the compiled code of the method as a sequence of bytecode instructions.

A compiled class consists of a constant pool segment, which is an array of the numeric, type and string constants that exist within a class [21]. They are defined once, and are referenced in all the other sections using their index. The figure below from [21] shows, the overall structure of a compiled class:

Modifiers, name, super class, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute*
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code

Figure 12: Compiled class file format [21]

Java types are represented in a different way in the compiled class than the source class. The exceptions that are raised by a method cannot be primitive types. They can only be interface or class types and hence are defined by internal names. For example, java/lang/String is the internal name for String.

Type descriptors are used for representing field types. The following figure referred from [21] represents type descriptors for some Java types:

Java type	Type descriptor
boolean	Z
char	C
byte	B
short	S
int	I
float	F
long	J
double	D
Object	Ljava/lang/Object;
int[]	[I
Object[][]	[[Ljava/lang/Object;

Figure 13: Type descriptors [21]

The descriptors of primitive types, class type and an array are shown in the figure above.

A method descriptor consists of a list of type descriptors which include return type and parameters of a method. Some sample method descriptors as represented in [21] are:

Method declaration in source file	Method descriptor
void m(int i, float f)	(IF)V
int m(Object o)	(Ljava/lang/Object;)I
int[] m(int i, String s)	(ILjava/lang/String;)[I
Object m(int[] i)	([I)Ljava/lang/Object;

Figure 14: Sample method descriptors [21]

4.1.2 Structure of Method

A method code is stored as a series of bytecode instructions inside a compiled class. Bytecode is the transitional representation of Java programs [9]. The figure below shows the overview of Java program execution, and where bytecode resides in the entire process [29]:

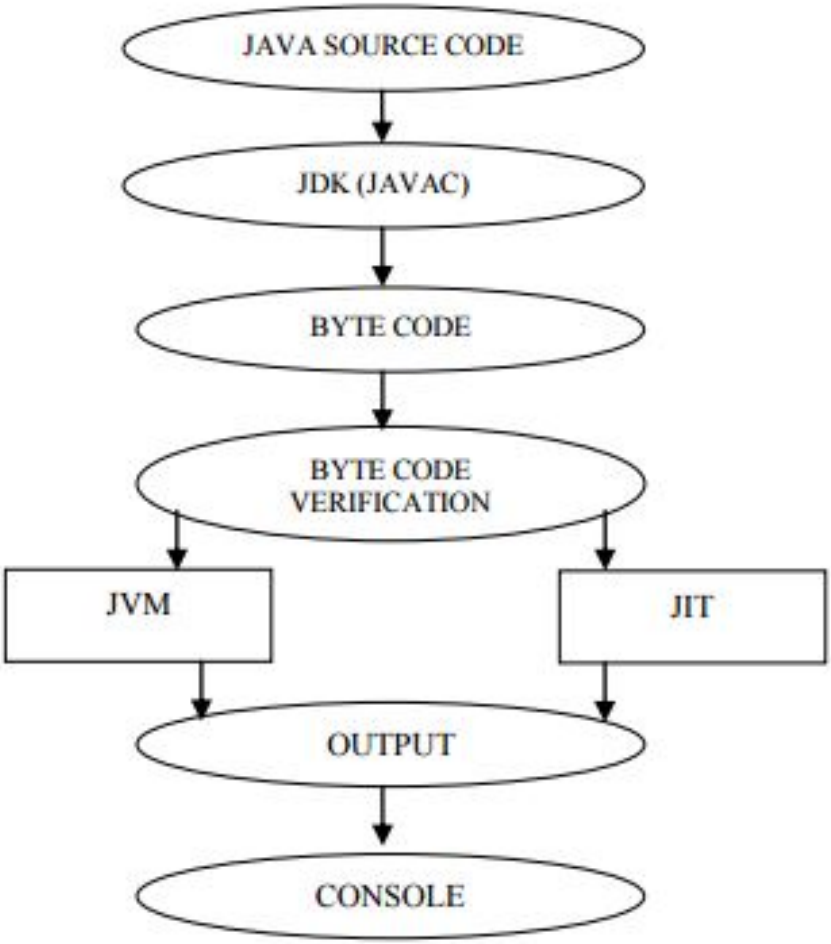

Figure 15: Java program execution [29]

To understand the bytecode thoroughly, we must understand the internal execution process of Java Virtual Machine (JVM). A JVM is a stack-based engine [9]. Every thread retains a JVM stack that stores frames. Each method invocation generates a new frame. A frame consists of an operand stack, which is an array of local variables. It also consists of reference to the run-time constant pool of the current method. The figure referred from [9] below shows how the JVM works:

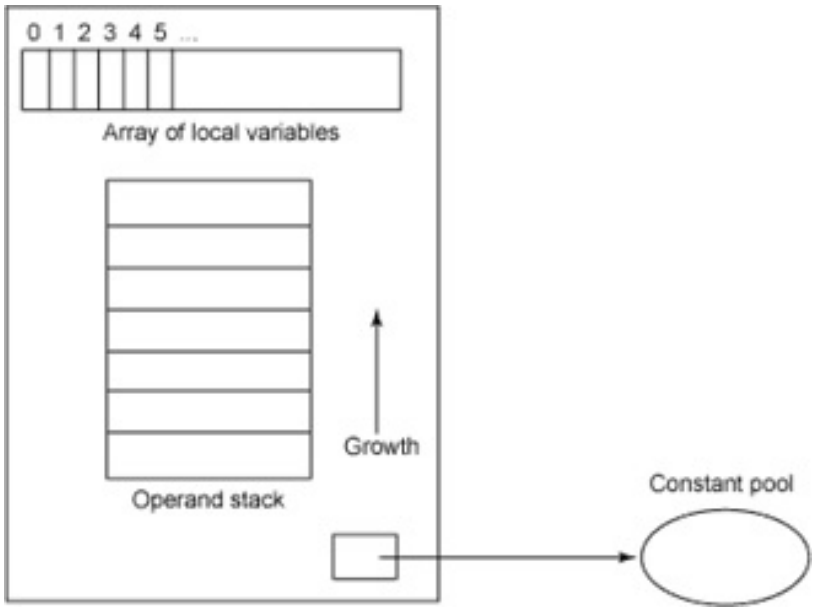

Figure 16: JVM bytecode execution [9]

The local variable table or the array of local variables consists of the method parameters and is used for holding the values of the local variables. Beginning at index 0, the parameters are stored first. Location 0 holds the reference if the frame is, for instance, method or constructor. It is followed by other locations which contain the next formal parameters. In case of a static method, location 0 is used for storing the first formal method parameter, and so on [9].

The size of the local variable table is calculated at the time of compilation, depending on the attributes of local variables as well as formal method parameters. A LIFO stack is used for pushing and popping values. Some opcode instructions retrieve values from the stack for computation, while others push values on it. The return values from methods are also handled by the operand stack.

4.1.3 Bytecode Instructions

A bytecode instruction has two parts, opcode, and arguments.

- The opcode is an unsigned byte value [21].
- The arguments are static values that are used for defining the instruction behavior [21].

The bytecode instructions can be divided into mainly two categories: one that are used for the exchange of values from local variables to the operand stack and vice versa; the others only work on the operand stack. They are used for computation of results, popping and pushing values on the stack.

In this paragraph, we discuss the first kind of bytecode instructions that are used to load and store from local variables and operand stack. The ILOAD, ALOAD, FLOAD, LLOAD and DLOAD instructions are used for reading a local variable and

pushing the value on the stack [21]. The index i of the local variable is taken as the argument. FLOAD, LLOAD and DLOAD are used for loading float, long and double respectively. ILOAD is used for loading of boolean, char, byte, int, or short local variable. Non-primitive values, which are objects and array references are loaded using ALOAD. In a similar way, ISTORE, FSTORE, LSTORE, ASTORE and DSTORE are used for popping the values from the operand stack and saving them back to the local variable at index i.

The next set of instructions is those that function only on the operand stack. They are as follows [21]:

- Stack - All possible instructions that can be used for manipulation of values on stack come under this category: POP, SWAP, DUP, PUSH, etc.
- Constants - A constant value is pushed on the operand stack by these instructions. For example, ICONST_0 pushes int value 0, ACONST_NULL pushes null, LDC cst pushes the random long, float, double, int, class or String constant cst, etc.
- Arithmetic and Logic - They do not have any argument and are used for performing arithmetic and logical operations on values, by popping them from the operand stack. After performing the operations, the result is saved back on the operand stack. The operations include xADD, xSUB, xDIV, xREM and xMUL, where x is either I, L, D or F [21].
- Casts - These instructions are used for typecasting operations like I2F, L2D, F2D, etc., which are converted from one numeric type to another.
- Objects - These instructions are used for creation and locking of new objects. A

new object can be pushed on the stack using NEW *type*, where type is internal name.

- Fields - These instructions are used for reading or writing the values of the field. GETFIELD and PUTFIELD are used for this purpose.

- Methods - These instructions are used for invoking a constructor or a method. These instructions first pop the values same as the number of method arguments and one extra for pushing the result. INVOKEVIRTUAL, INVOKESTATIC, INVOKESPECIAL, INVOKEINTERFACE, etc. are used for invoking different kinds of methods and constructors.

- Arrays - Reading and writing of values in arrays is performed by these instructions. Some instructions are xASTORE, where x can be I, F, L, D, A, B, C, S.

- Jumps - Jumps are used to go to an arbitrary instruction on the occurrence of a certain condition. Examples of these instructions include IFNE, IFGE, TABLESWITCH, etc.

- Return - RETURN and xRETURN are used to return the result and terminate the method.

Let us consider an example given in [21], to get a clear understanding:

```
public void checkAndSetF(int f) {
    if (f >= 0) {
       this.f = f;
    } else {
       throw new IllegalArgumentException();
    }
}
```

The bytecode of this method is:

```
   ILOAD 1
   IFLT label
   ALOAD 0
   ILOAD 1
   PUTFIELD pkg/Bean f I
   GOTO end
label:
   NEW java/lang/IllegalArgumentException
   DUP
   INVOKESPECIAL java/lang/IllegalArgumentException <init> ()V
   ATHROW
end:
   RETURN
```

The local variable 1 is pushed on the operand stack using the first instruction. The next instruction compares the value popped from the operand stack to 0. If the value is lower than 0, it jumps to the instruction label *label*, otherwise the next instruction is executed. The next instruction pushes this on the stack, followed by pushing the local variable 1 on the stack. The PUTFIELD instruction, then stores the value i of the f field of the object, i.e. *this.f*. The GOTO instruction goes to the designated instruction addressed by the end label. The end label contains the RETURN instruction. The instructions after the label create and throw an exception: the NEW instruction is used for creating an exception object and pushing it on the

stack. The value is duplicated using the DUP instruction. The INVOKESPECIAL pops one copy, followed by the exception constructor invocation on it. Lastly, the ATHROW instruction is used to throw an exception by popping the remaining copy [21].

4.2 Bytecode Manipulation Library - ASM

Bytecode manipulation is an effective technique that can be used to alter existing classes or generate classes dynamically. There are various bytecode manipulation frameworks and libraries available. Some of them are ASM [11], BCEL [12], CGLib [13], Javassist [14], Serp [15], Cojen [16], Soot [17], etc.

After evaluating the existing frameworks used for bytecode manipulation, engineers developed a more efficient framework to boost performance and memory efficiency. ASM can be used for different purposes like analysis and manipulation of Java bytecode [11].

4.2.1 Objectives of ASM

The following are the objectives or motivations for ASM as described by Eric Bruneton in [21]:

1. In generating the source code dynamically, there is an overhead of compiling the source code. It not only adds to the time, but also to the size of the code. The objective of ASM was to use a small tool which is time as well as size efficient.

2. One main rule of optimizing the performance of any application is to optimize frequently used codes first. So, the second objective of ASM is to build a tool for the most frequent dynamic class manipulations.

3. The final objective of ASM was to build a general tool, which could be used for class manipulation operations.

4.2.2 Overview

The primary objective of ASM library is analyzing, transforming and generating Java classes. Byte arrays can be read, written and transformed by using higher level concepts, and not restricting to bytes by ASM [21]. ASM skips the process of class loading and is restricted to reading, writing, analyzing and transforming classes.

The ASM library provides two APIs for manipulation of compiled Java classes: one is the core API and other is the tree API.

Core API - An event-based depiction of classes is provided by this API. A class is denoted as a sequence of events, with every event denoting an element of the class. These elements may be headers, fields, instructions, method declarations, and others. The core/event-based API defines a set of all the possible events and prioritizes them in order of occurrence. It also provides a class parser and writer, which are used to generate one event per element and to generate compiled classes from these sequences of events respectively.

Tree API - It is also known as an object-based model. In this API, a class is denoted as a tree of objects, with each object denoting some part of a class, which are a field, a method, or the class itself. This API facilitates the conversion of events that represent a class to the object tree and, vice-versa. In other words, it can be said that the object-based API resides over the event-based API [21].

ASM provides both of these APIs as it cannot be said that one of them is best. Both of them have their advantages and disadvantages. The event-based API

requires less memory and is faster as compared to tree API, as it does not need to create and store objects representing classes [21]. However, it has a drawback that transformations are difficult as only one element of a class is available at one time, which is in contrast with the tree-based API. In tree based API, the whole class is available for transformation in memory [21].

4.2.3 Interfaces and Components

In this section, we would be focusing more on the use of the ASM Tree API for understanding its interaction with class files. The tree API is based on the ClassNode class as described in [21]:

```
public class ClassNode ... {
   public int version;
   public int access;
   public String name;
   public String signature;
   public String superName;
   public List<String> interfaces;
   public String sourceFile;
   public String sourceDebug;
   public String outerClass;
   public String outerMethod;
   public String outerMethodDesc;
   public List<AnnotationNode> visibleAnnotations;
   public List<AnnotationNode> invisibleAnnotations;
   public List<Attribute> attrs;
   public List<InnerClassNode> innerClasses;
   public List<FieldNode> fields;
   public List<MethodNode> methods;
}
```

Similar classes exist for FieldNode and MethodNode, which contain a structure similar to the contents of the subsections of class file structure.

As mentioned earlier, ASM allows generation of the classes. This can be achieved by simply generating a ClassNode object and initializing its values for the field. Adding class members or removing them can be accomplished by adding or deleting elements in the parameters of a ClassNode object [21].

In addition, the ClassNode class extends the ClassVisitor class [21]. The ClassVisitor class also provides an accept method, which is used to generate events depending on the field values of ClassNode.

- A ClassNode can be constructed from a byte array by constituting it with a ClassReader. The ClassNode component depletes the events that are generated by the ClassReader, which results in the field initialization.
- Similarly, a ClassNode can be transformed to its byte array format by using it with a ClassWriter. In this case, the accept method produces events that are absorbed by the ClassWriter.
- Transforming the classes can be achieved by combining ClassReader, ClassWriter and addition of transformation code.

The details about how classes can be actually transformed are mentioned in Chapter 6.

CHAPTER 5

Hidden Markov Models

Hidden Markov Models are known to be used in various areas like speech recognition, biological sequence selection, piracy detection, and study of protein structure. In the past few years, they were known to be used for detecting the metamorphic malware. The HMMs are trained with the use of known malware opcodes. They can then be used for scoring the files [28].

5.1 Overview of HMMs

The HMMs (i.e. state machine based models) is useful to describe a set of observations developed by a stochastic process [24]. These processes can be demonstrated as state sequences, where the succeeding state depends entirely on the present state. Let us demonstrate the Hidden Markov Models using Dr. Stamp's paper [25].

In an example from this paper, the problem is to calculate the temperature at a specific location at the particular instance of the time. Also, let us consider that there was no accurate way to determine the temperature during the period of time under question. As there are no recorded past temperatures, we would consider only two annual temperature descriptions, one is 'hot 'and another is 'cold'. Suppose, the scientists have arrived at the probability of 0.6 for a cold year to be after a cold year, and '0.7 'for a hot year that is after a hot year. This information can be denoted in the form of matrix as follows:

$$\begin{array}{c} \\ H \\ C \end{array} \begin{array}{c} \begin{array}{cc} H & C \end{array} \\ \begin{bmatrix} 0.7 & 0.3 \\ 0.4 & 0.6 \end{bmatrix} \end{array}$$

Figure 17: Matrix for temperature probabilities [25]

where H denotes hot and C denotes cold.

Also, we assume that an interrelation exists among the temperature and tree growth ring sizes [25]. Let us consider three different rings, small, medium and large denoted by S, M, and L respectively. Let us assume the relation between annual temperature and the tree ring sizes to be as follows [25]:

$$\begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix}$$

For above example, the state is the average annual temperature —H or C. The next state depends only on the current state, and hence it is a Markov process. We cannot directly have access to temperatures of the past, and so the actual states are known to be 'hidden'. Although the temperature cannot be observed directly, the tree ring sizes are observable. Since, the states are hidden, the model is known as a Hidden Markov model.

The transition matrix will look as follows:

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

and

$$B = \begin{bmatrix} 0.10.40.5 \\ 0.70.20.1 \end{bmatrix}$$

Let us assume that the initial state distribution is: $\pi = [0.6\ 0.4]$

All the matrices above are row stochastic, that is, the summation of all elements in every row is 1 and each element is a probability. Let us consider a span of four years, for which the sequence of the rings is S, M, S, L. Let S be 0, M be 1 and L be 2. Therefore, the observation sequence O is (0, 1, 0, 2).

The most likely state series of the Markov process can be determined from given observations. For instance, if we want to determine the average annual temperature, it can be defined as the state sequence which will exploit the number of correct states. We can use HMMs to find this sequence. We will now have a look at the notations, which are the most challenging part of HMM. The notations as described in [25] are as follows:

Let

T = the observation sequence length

N = number of states

M = number of observation symbols

Q = $q_0, q_1, ..., q_{N^-1}$ = states of the Markov process

V = 0, 1, . . . , M—1 = set of possible observations

A = state transition probabilities

B = observation probability matrix

π = initial state distribution

O = $(O_0, O_1, \ldots, O_{T^-1})$ = observation sequence.

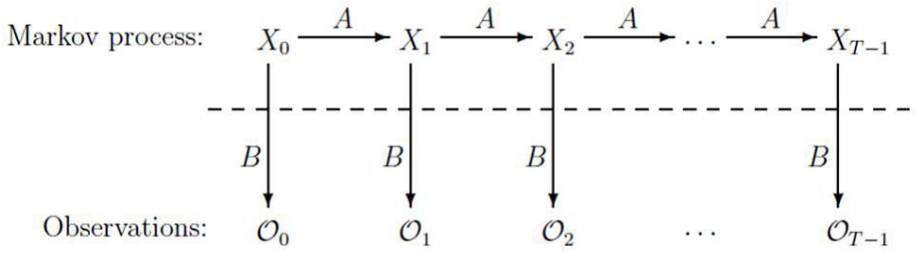

Figure 18: Markov process [23]

The above figure taken from [23] denotes a generic HMM. $X_0, X_1, \ldots, X_T$ represents the hidden state sequence. The observations corresponding to the hidden states are connected by B.

For the temperature example described above, we have T = 4, N = 2, M =3, Q = H,C, V =0, 1, 2 where 0, 1, 2 represent the tree ring sizes. The model is defined using A, B and π. The HMM can be denoted as λ = (A, B, π). The HMM is denoted as λ = (A, B, π). Let us assume that the length of a state sequence is four. X = (x_0, x_1, x_2, x_3) with corresponding observations O = (O_0, O_1, O_2, O_3)

Then, for starting in the state x_0 the probability would be πx_0. Also, bx_0 (O_0) is the probability of O_0. Similarly, ax_0, x_1 denotes the probability of transition from x_0 to x_1. Further, the probability of the state sequence X can be denoted by P(X) = πx_0 bx_0 (O_0) ax_0, x_1 $bx_1(O_1)$ ax_1, x_2 bx_2 $(O_2)ax_2$,$x_3bx_3(O_3)$.

With the example described above, we have the sequence O = (0, 1, 0, 2). The probability is computed as follows:

P(H H C C) = = 0.6(0.1)(0.7)(0.4)(0.3)(0.7)(0.6)(0.1) = 0.000212

In a similar way, the probability of each possible state can be calculated. To find the most probable sequence, we have to look at each position independently and find which of the two 'H' or 'C' have a higher probability for that particular position. We

then choose the state having the higher probability for that position. We can add the normalized probabilities of all sequences starting with either H or C. The one with the higher probability best fits that position. The calculated probabilities of each position are given in the table below [23]:

State	Probability	Normalized Probability
HHHH	0.000412	0.042787
HHHC	0.000035	0.003635
HHCH	0.000706	0.073320
HHCC	0.000212	0.022017
HCHH	0.000050	0.005193
HCHC	0.000004	0.000415
HCCH	0.000302	0.031364
HCCC	0.000091	0.009451
CHHH	0.001098	0.114031
CHHC	0.000094	0.009762
CHCH	0.001882	0.195451
CHCC	0.000564	0.058573
CCHH	0.000470	0.048811
CCHC	0.000040	0.004154
CCCH	0.002822	0.293073
CCCC	0.000847	0.087963

Table 1: Probabilities [23]

Using the table below, we can now pick up the state with the highest probability.

	element			
-	0	1	2	3
P(H)	0.188182	0.519576	0.228788	0.804029
P(C)	0.811818	0.480424	0.771212	0.195971

Table 2: State with highest probabilities [23]

The optimal sequence will be CHCH.

5.2 Threshold Approach

As described in Wong and Stamp's paper, threshold approach [26] can be used for detecting the malware. The HMMs after being trained are used for malware detection.

The working of the threshold model can be described as follows:

1. Malware file opcode sequences can be used to train an HMM.

2. This result can be observed by performing multiple runs to obtain a threshold value. Once, the threshold value is obtained, all the files scoring above are considered as malicious.

We will implement threshold approach for detection. The details for testing are included in Chapter 7.

CHAPTER 6

Design and Implementation

This chapter consists of all the design principles that were carried during this project. It also comprises of the machine configurations that were used during the experiments.

6.1 Machine Configurations and Programming Languages

6.1.1 Machine Configurations

Host Machine:

- Operating System: Windows 8
- Model: Lenovo Ideapad Z400
- Processor: Intel (R) Core (TM) i5-3230M CPU @ 2.60 GHz
- RAM: 6.00 GB
- System type: 64-bit Operating System, x64 based processor

Guest Machine:

- Operating System: Ubuntu 12.04 LTS
- Model: Lenovo Ideapad Z400
- Processor: Intel (R) Core (TM) i5-3230M CPU @ 2.60 GHz
- RAM: 1.00 GB
- System type: 64-bit Operating System, x64 based processor

6.1.2 Programming Languages - Java

- Java bytecode manipulation library ASM was used for code obfuscation.
- JAHMM, which is a Java library was used for building Hidden Markov Models.

6.2 Design

The metamorphic engine is constructed using the tree API of ASM library. This section will give a brief and detailed explanation of the algorithm. The figure below explains the overview of the algorithm:

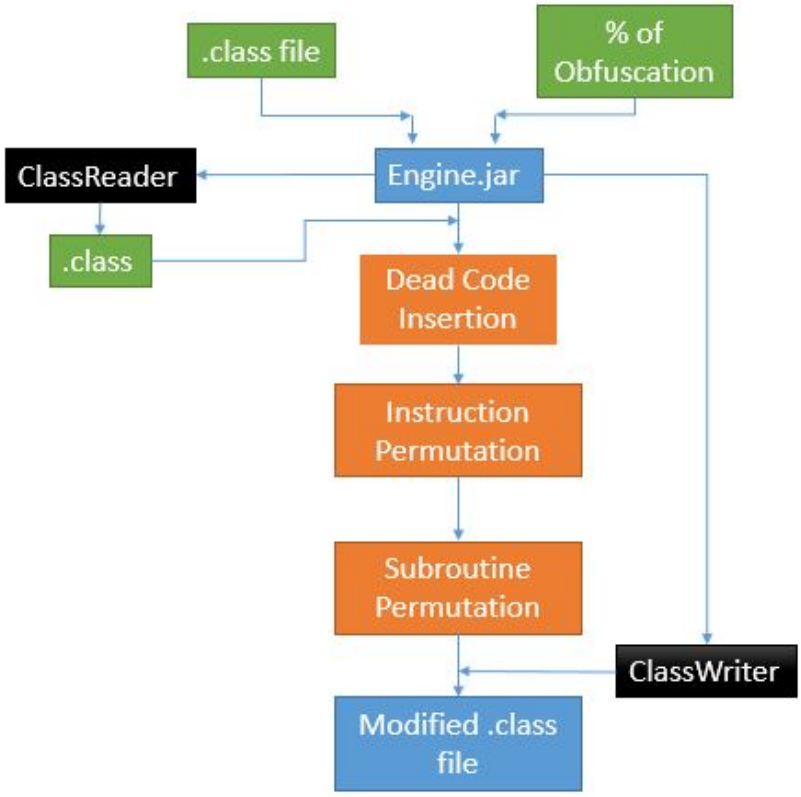

Figure 19: Metamorphic Engine Algorithm

6.2.1 Algorithm

The detailed algorithm that is used for the design of the morphing code is as follows:

1. A class MyClassAdapter that inherits ClassNode class in tree package is written. Also, ClassReader is used to read from a file, and ClassWriter writes the altered bytecode to the output file. The MyClassAdapter class takes two arguments: the class file name that needs to be modified and the percentage of modification that needs to be performed.

    - Engine.jar *sample.class 30*

2. The class MyClassAdapter reads from a file with the use of ClassReader, and makes a clone of that class.

3. The main transformation function is used to perform dead-code insertion, subroutine permutation and instruction permutation.

    (a) This function visits the methods in order, transforms it, and then add it to the new cloned class.

    (b) This method creates two objects, one for the insertion of dead-code, and the other for instruction permutation. Both these operations are performed in such a way that the overall meaning of the code remains unchanged.

    (c) The next step performed is subroutine permutation. According to the position of new added methods, the last position of methods is determined. *this.method.add(0, method)* means add new modified method to the front of the method list, so the methods are ordered randomly.

4. Once the transformation has been achieved, the last step is to write the class in bytecode format to the output file. The new modified class is written to the output file by the ClassWriter.

6.2.2 Overall Transformation Process

As mentioned in Chapter 4, the tree API relies on the use of the ClassNode class for generation and transformation of Java classes.

A class can be generated by creating a ClassNode object and then initializing field values. Class members can be added or removed by the addition or removal of members in the fields or methods of a ClassNode object [21].

The ClassNode class extends the ClassVisitor class [21]. The ClassVisitor class provides an accept method, which can be used to generate events depending on the field values of ClassNode. The ClassVisitor methods implement the opposite operation of setting the ClassNode fields depending on the received events. The following code snippet from [21] shows the accept method, along with class hierarchy:

- A ClassReader and a ClassNode can be used together to construct a ClassNode from a byte array. The events produced by the ClassReader are absorbed by the ClassNode component, which results in the field initialization. The following code snippet from [21] shows how ClassReader can be used:

```
public class ClassNode extends ClassVisitor {
  ...
    public void visit(int version, int access, String name,
       String signature, String superName, String[] interfaces[]) {
             this.version = version;
             this.access = access;
             this.name = name;
             this.signature = signature;
             ...
    }
  ...
    public void accept(ClassVisitor cv) {
             cv.visit(version, access, name, signature, ...);
             ...
    }
}
```

```
ClassNode cn = new ClassNode();
ClassReader cr = new ClassReader(...);
cr.accept(cn, 0);
```

- Similarly, a ClassNode can be changed to its byte array format by using it with a ClassWriter. In this case, the events produced by the accept method are consumed by the ClassWriter. The code below from [21] shows how ClassWriter is used for converting a ClassNode to its byte representation :

```
ClassWriter cw = new ClassWriter(0);
cn.accept(cw);
byte[] b = cw.toByteArray();
```

- Transformation of the classes can be achieved by integrating the ClassReader, the ClassWriter and the addition of transformation code. The transformation code from [21] is:

```
ClassNode cn = new ClassNode(ASM4);
ClassReader cr = new ClassReader(...);
cr.accept(cn, 0);
...
// here transform cn as you want
ClassWriter cw = new ClassWriter(0);
cn.accept(cw);
byte[] b = cw.toByteArray();
```

6.2.3 Code Transformation

Three methods have been used for code transformation, which include dead-code insertion, subroutine permutation and instruction permutation. Let us consider a small code snippet on which we would perform these techniques individually, and then combine all of them to get our end result.

Let us consider a simple program for addition of two numbers:

```
package General;

public class Add{

public static int add(int a, int b)
{
int c = a + b;
return c;
}

public static void main(String[] args) {
int temp = add(4,5);
System.out.println(temp);
}
}
```

The bytecode for this program is as follows:

```
Compiled from "Add.java"
public class General.Add {
  public General.Add();
    Code:
       0: aload_0
       1: invokespecial #8       // Method java/lang/Object."<init>":()V
       4: return

  public static int add(int, int);
    Code:
       0: iload_0
       1: iload_1
       2: iadd
       3: istore_2
       4: iload_2
       5: ireturn

  public static void main(java.lang.String[]);
    Code:
       0: iconst_4
       1: iconst_5
       2: invokestatic  #22       // Method add:(II)I
       5: istore_1
       6: getstatic     #24       // Field java/lang/System.out:Ljava/io
                                     /PrintStream;
       9: iload_1
      10: invokevirtual #30       // Method java/io/PrintStream.println:
                                     (I)V
      13: return
}
```

6.2.3.1 Dead Code Insertion

Dead code insertion is performed by inserting junk instructions in every method of the class file depending on the percentage of the obfuscation that needs to be performed. Each method is visited and depending on the instructions and the number of instructions present in the method, new junk instructions is inserted in the method body. Dead code is added to avoid over-fitting the model. If the instruction ends

with "LOAD", the size of the stack is increased by cloning, which returns a copy of the instruction and adding extra instructions before RETURN statement. Also, AbstractInsNode list that exists within a method is used for adding junk instructions to the InsnList, which is the instruction list for the method. An AbstractInsnNode denotes a bytecode instruction. One InsnList can contain an instruction at most once at a time. The dead-code insertion for the above program is as follows:

```
Compiled from "Add.java"
public class General.Add {
  public General.Add();
    Code:
       0: aload_0
       1: invokespecial #9        // Method java/lang/Object."<init>":()V
       4: aload_0
       5: return

  public static int add(int, int);
    Code:
       0: iload_0
       1: iload_1
       2: iadd
       3: istore_2
       4: iload_2
       5: iload_0
       6: iload_1
       7: iload_2
       8: ireturn

  public static void main(java.lang.String[]);
    Code:
       0: iconst_4
       1: iconst_5
       2: invokestatic  #21       // Method add:(II)I
       5: istore_1
       6: getstatic     #27       // Field java/lang/System.out:Ljava/io
                                     /PrintStream;
       9: iload_1
      10: invokevirtual #33       // Method java/io/PrintStream.println:
                                     (I)V
      13: iload_1
      14: return
}
```

6.2.3.2 Subroutine Permutation

All the methods are stored in a list and transformation is performed by iterating on each method. According to the position of modified methods, the last position of methods is determined. *this.method.add(0, method)* adds the newly modified method to the front of the method list, so the new output class that is written by the ClassWriter has its methods in random order. The subroutine permutation for the program given above is:

```
Compiled from "Add.java"
public class General.Add {
  public static void main(java.lang.String[]);
    Code:
       0: iconst_4
       1: iconst_5
       2: invokestatic  #11       // Method add:(II)I
       5: istore_1
       6: getstatic     #17       // Field java/lang/System.out:Ljava/io
                                     /PrintStream;
       9: iload_1
      10: invokevirtual #23       // Method java/io/PrintStream.println:
                                     (I)V
      13: return

  public static int add(int, int);
    Code:
       0: iload_0
       1: iload_1
       2: iadd
       3: istore_2
       4: iload_2
       5: ireturn

  public General.Add();
    Code:
       0: aload_0
       1: invokespecial #34       // Method java/lang/Object."<init>":
                                     ()V
       4: return
}
```

6.2.3.3 Instruction Permutation

Instruction permutation modifies the method body, by re-ordering the flow of instructions which are not dependent on each other. This is achieved by using jump instructions, to change the flow of the order in which instructions are executed. The meaning of the code still remains the same. The following code shows instruction permutation for add function:

```
public static int add(int, int);
  Code:
     0: iload_0
     1: iload_1
     2: iadd
     3: istore_2
     4: goto          28
     7: iload_1
     8: iflt          19
    11: aload_0
    12: iload_1
    13: putfield      #15        // Field pkg/Bean.f:I
    16: goto          27
    19: new           #17        // class java/lang/
                                    IllegalArgumentException
    22: dup
    23: invokespecial #18        // Method java/lang/
                                    IllegalArgumentException."<init>":()V
    26: athrow
    27: return
    28: iload_2
    29: ireturn
```

The combination of all three techniques performed on the above code produces the following result:

```
Compiled from "Add.java"
public class General.Add {
  public static void main(java.lang.String[]);
    Code:
       0: iconst_4
       1: iconst_5
       2: invokestatic  #11        // Method add:(II)I
       5: istore_1
       6: getstatic     #17        // Field java/lang/System.out:Ljava/io
                                     /PrintStream;
       9: goto          33
      12: iload_1
      13: iflt          24
      16: aload_0
      17: iload_1
      18: putfield      #23        // Field pkg/Bean.f:I
      21: goto          32
      24: new           #25        // class java/lang/
                                     IllegalArgumentException
      27: dup
      28: invokespecial #29        // Method java/lang/
                                     IllegalArgumentException."<init>":()V
      31: athrow
      32: return
      33: iload_1
      34: invokevirtual #35        // Method java/io/PrintStream.println:
                                     (I)V
      37: iload_1
      38: return

  public static int add(int, int);
    Code:
       0: iload_0
       1: iload_1
       2: iadd
       3: istore_2
       4: goto          28
       7: iload_1
       8: iflt          19
      11: aload_0
      12: iload_1
      13: putfield      #23        // Field pkg/Bean.f:I
```

```
      16: goto          27
      19: new           #25        // class java/lang/
                                      IllegalArgumentException
      22: dup
      23: invokespecial #29        // Method java/lang/
                                      IllegalArgumentException."<init>":()V
      26: athrow
      27: return
      28: iload_2
      29: iload_0
      30: iload_1
      31: iload_2
      32: ireturn

  public General.Add();
    Code:
       0: aload_0
       1: invokespecial #42        // Method java/lang/Object."<init>":()V
       4: goto          28
       7: iload_1
       8: iflt          19
      11: aload_0
      12: iload_1
      13: putfield      #23        // Field pkg/Bean.f:I
      16: goto          27
      19: new           #25        // class java/lang/
                                      IllegalArgumentException
      22: dup
      23: invokespecial #29        // Method java/lang/
                                     IllegalArgumentException."<init>":()V
      26: athrow
      27: return
      28: aload_0
      29: return
}
```

6.3 Generation of metamorphic variants

This section will include the details for the generation of metamorphic variants based on the algorithm described in the Design section of this chapter.

6.3.1 Original files

A hundred copies of a single Java class file were generated each having different amounts of dead-code inserted between 20% to 30% as the original files, which were modified using different techniques specified in the next section. These 100 files are used as original files.

6.3.2 Modified files

The copies of the base file are modified with increments of 20% of code obfuscation. In order to test the efficiency of each obfuscation technique, we test each technique separately, as well as in combination:

- Original files are obfuscated using only dead-code insertion.
- Original files are obfuscated using only subroutine permutation.
- Original files are obfuscated using only instruction permutation.
- Original files are obfuscated using an aggregation of subroutine permutation and dead-code insertion.
- Original files are obfuscated using an aggregation of instruction permutation and dead-code insertion.
- Original files are obfuscated using a combination of subroutine permutation and instruction permutation.
- Original files are obfuscated using a combination of all three techniques: subroutine permutation, dead-code insertion and instruction permutation.

Thus, we have 100 files for each of the obfuscation performed. These gives us a huge data-set, as obfuscation is performed in multiples of 20%. The testing of generated files is explained in the next chapter.

CHAPTER 7

Experiments and Results

For testing the validity of generated variants of the base file, we will Hidden Markov Models(HMM). We focus on the use of threshold approach to test the obfuscated files. In this approach, we used k-fold cross validation. In our case, we used five-fold cross validation.

The goal of the project was to create a morphing engine which could evade HMM based detection. If some of the modified files that we generated score higher than the normal Java class files, the modified files will not be detected by the HMM. So, we use additional 100 Java class files, from simple programs that we write daily. Some of these programs were written by me in the past for some coding assignments. Others were gathered online from [30]. We then score these normal files, as well as the modified files against our model. We first divide the original files in five sets of 20 files each. In threshold approach, HMM is trained using four sets of the original code and tested using a fifth subset of obfuscated files as well as normal files. To understand the efficiency of each technique, we would morph the base files individually, increasing the percentage of obfuscation in multiples of 20%, until we are unable to distinguish between normal and modified files.

It was also observed that the number of states have no significant effect on the results. Therefore, we use only 4-state model for testing purposes.

7.1 Dead-code Insertion

The original files were modified by inserting only dead-code in them, starting from 20% and above. It was observed that inserting 40% of dead-code in the original files was enough to evade the HMM-based detection. The figure below shows the graph that represents the overlap between the normal files and the modified file, when 40% of dead-code is inserted:

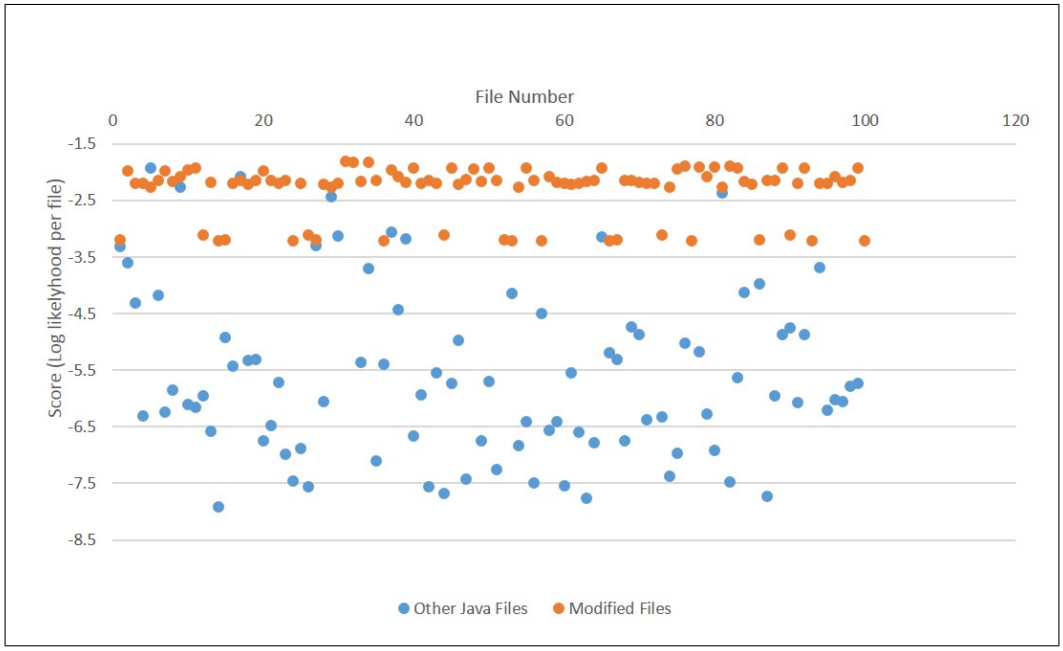

Figure 20: HMM result with dead-code insertion

7.2 Subroutine Permutation

The original files were modified by only using subroutine permutation. It was observed that, there is very less overlap between the modified files and the normal files. The detection might be still possible in this case. The graph below shows the results of modifications performed on the original files versus the normal files:

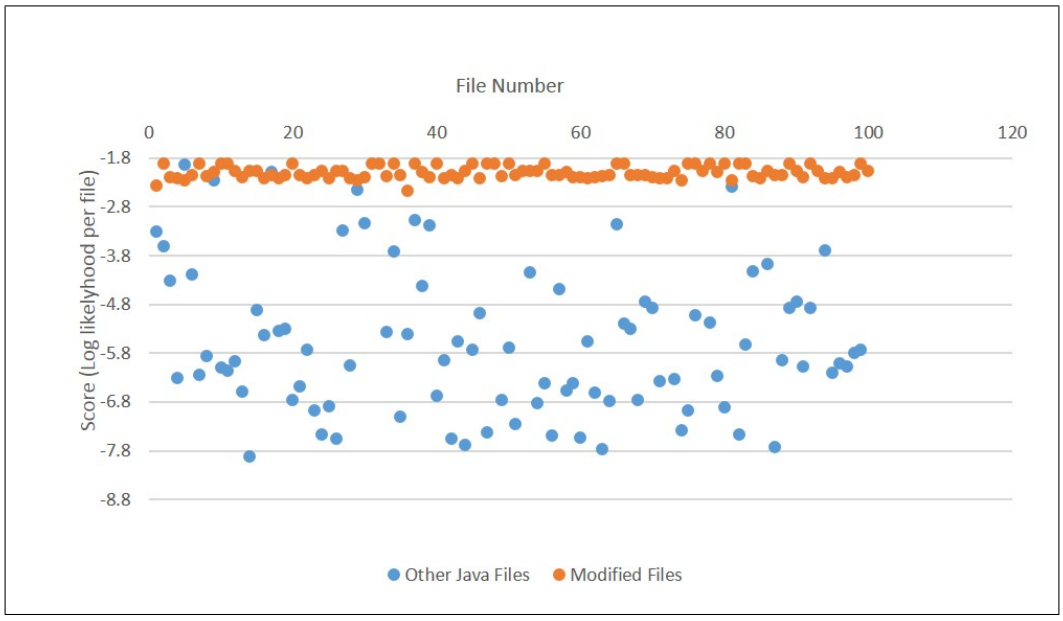

Figure 21: HMM result with subroutine permutation

7.3 Instruction Permutation

Instruction permutation is used to modify the class files. The HMM-based approach is unable to detect this permutation. It cannot easily distinguish between the modified and the normal files for all the increments of percentage. In the graph below, we can see that there is less though enough overlap between the normal and the modified files, evading HMM-based detection:

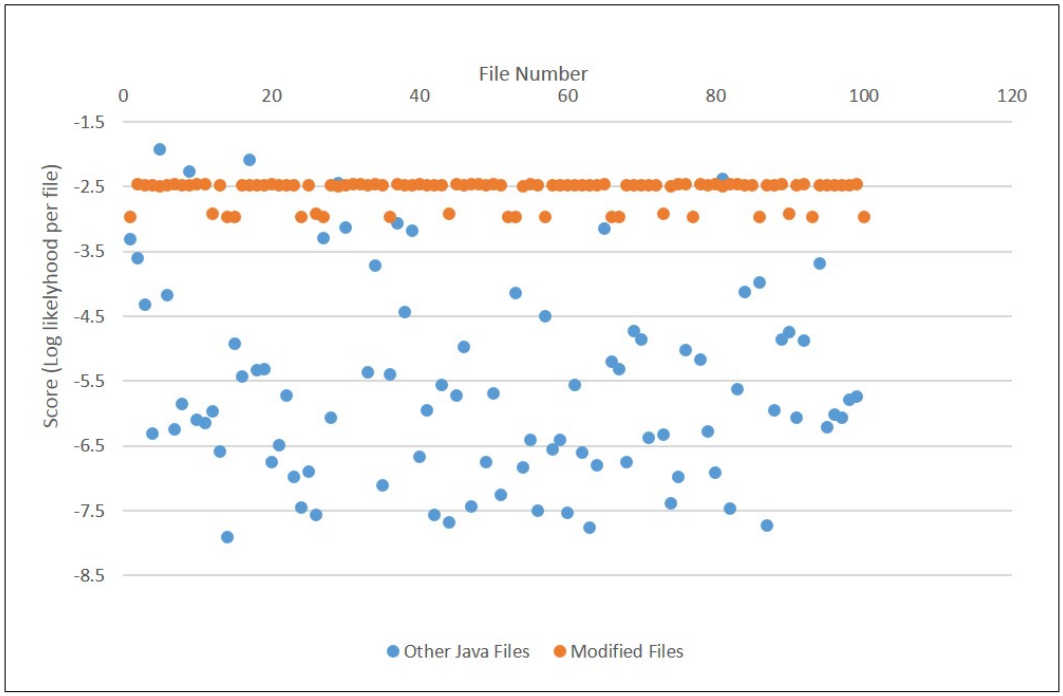

Figure 22: HMM result with instruction permutation

7.4 Dead-code Insertion and Subroutine Permutation combination

With obfuscating the files with 40% of dead-code insertion and subroutine permutation, we can evade HMM-based detection of original files. In the graph presented below, we can see that it is difficult to distinguish between the normal and the modified files:

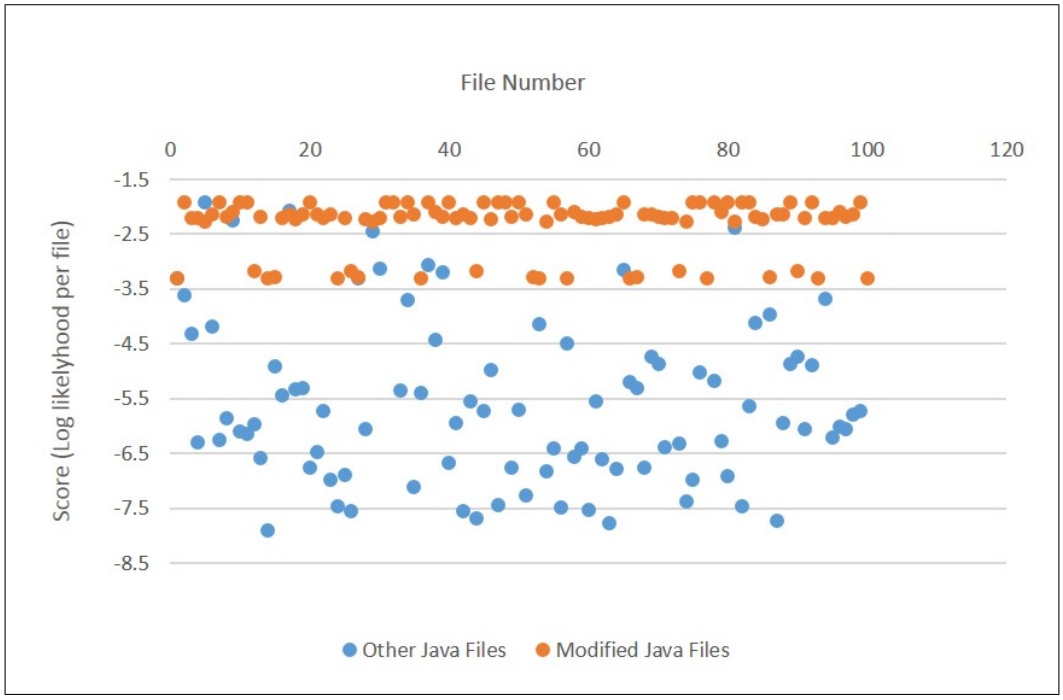

Figure 23: HMM result with dead-code and subroutine permutation

7.5 Dead-code Insertion and Instruction Permutation combination

When using dead-code insertion and instruction permutation in combination, it is observed that, a high percentage of obfuscation needs to be performed to evade HMM-based detection. This combination of techniques is able to escape HMM-based over 60% of obfuscation. The figure below shows the results when using 60% of dead-code insertion and instruction permutation:

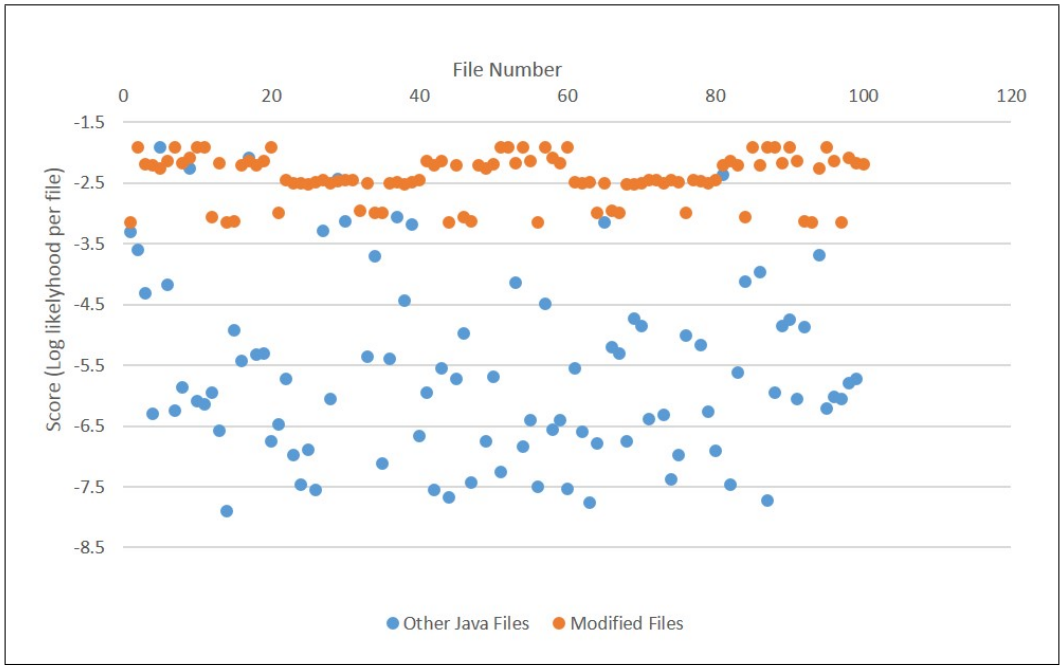

Figure 24: HMM result with dead-code and instruction permutation

7.6 Subroutine Permutation and Instruction Permutation combination

The combination of subroutine permutation and instruction permutation is able to escape HMM-based detection. Although there is very little overlap between the normal files and modified files, it is not possible to distinguish between them. The figure below shows the result of combination for these techniques:

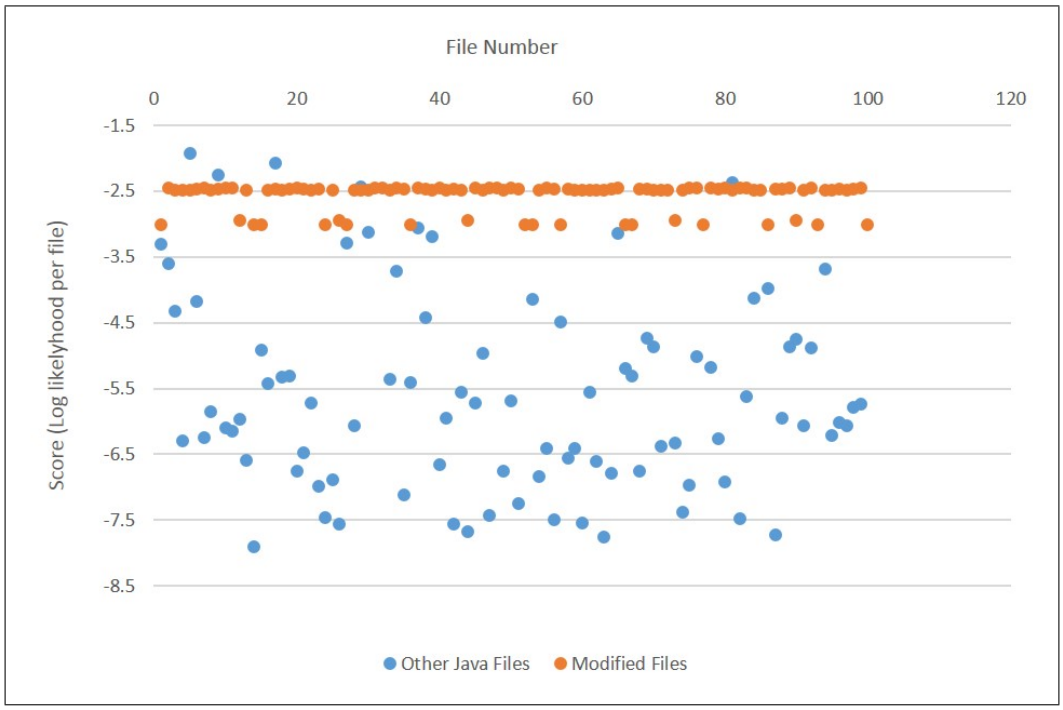

Figure 25: HMM result subroutine permutation and instruction permutation

7.7 Combination of all three techniques

It is is observed that when using all the three techniques together, it is possible to evade HMM-based detection. Staring at 20%, we increment the obfuscation. It is observed that, for 40% the normal files and modified files overlap. Further increase in percent of obfuscation results into more overlapping of files. The figure below shows the results for using this combination:

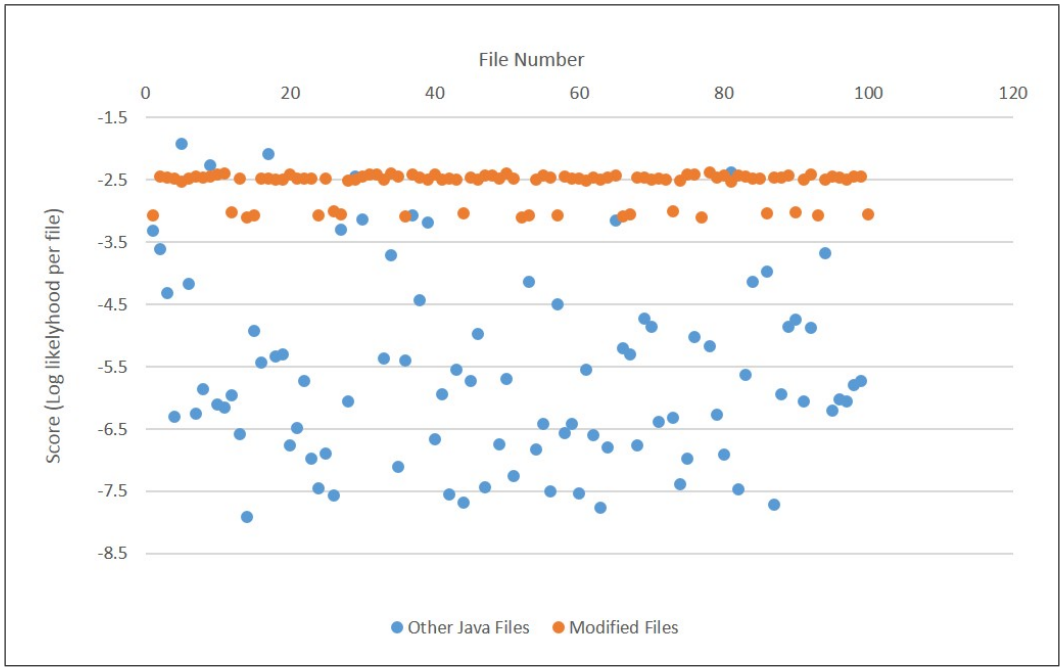

Figure 26: HMM result with all three techniques

From the above observations and charts presented in Appendix A, it can be observed that the most effective modifications to the files are observed when using the dead-code insertion individually, and combination of dead-code insertion with instruction permutation. Along with that, the combination of all three techniques have an effective result. The combination of dead-code insertion with subroutine permutation also shows a considerable overlap between the normal files and modified files. The tale below summarizes the techniques and their effectiveness:

Technique	Detected with HMM?	% of obfuscation required to evade HMM
Dead-code Insertion	No	40%
Subroutine Permutation	Can be detected partially	-
Instruction Permutation	No	-
Dead-code Insertion + Subroutine Permutation	No	40%
Dead-code Insertion + Instruction Permutation	No	60%
Subroutine Permutation + Instruction Permutation	Can be detected partially	-
Combination of all three techniques	No	40%

Table 3: Obfuscation Results for different techniques

Additional detailed charts for all the techniques and their combinations are presented in Appendix A.

CHAPTER 8

Conclusion and Future Work

This project proposed the idea of using bytecode manipulation libraries in order to manipulate Java class files, so as to escape detection by HMM. There are various bytecode manipulation libraries like Javassist, BCEL, ASM, Soot, out of which, ASM was chosen for performing various code obfuscation techniques. The code obfuscation techniques, which included dead code insertion, subroutine permutation, and instruction permutation were performed abiding by the rules that are necessary to achieve obfuscation without changing the meaning of the original code.

In order to better understand the efficiency of each technique, they were implemented individually and in combinations to achieve the best possible obfuscation. It was observed that the combination of all three techniques, and using instruction permutation, dead-code insertion individually provides the best possible code obfuscation. Along with that, and using combination of dead-code insertion with other techniques individually also provide a good obfuscation.

It was observed that the HMMs could partially some obfuscation techniques correctly like subroutine permutation, as there was very little overlap between normal and modified files. But, the model failed to detect modifications in techniques like instruction permutation, dead-code insertion and various other combinations. The tool was successfully able to evade detection in most of the cases.

A good future project would be to perform obfuscation using libraries like Soot, BCEL, and compare their efficiency with the obfuscation performed by ASM. These libraries provide a similar framework and hence, their advantages and drawbacks with

respect to ASM need to be studied.

LIST OF REFERENCES

[1] Threat Assessment of Malicious Code and Human Threats, NIST, FIPS 197, March 10, 1994, http://csrc.nist.gov/publications/nistir/threats/

[2] What Is the Difference: Viruses, Worms, Trojans and Bots?, http://www.cisco.com/web/about/security/intelligence/virus-worm-diffs.html, accessed September 2014.

[3] E. Konstantinou and S. Wolthusen, ''Metamorphic Virus: Analysis and Detection'', https://www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-02.pdf

[4] M. Stamp, *Information Security: Principles and Practice*, second edition, Wiley, 2011

[5] X. Li, P. Loh and F. Tan, ''Mechanisms of Polymorphic and Metamorphic Viruses'', http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6061171&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D6061171

[6] P. Szor, *The Art of Computer Virus Research and Defense*, Pearson Education, 2005

[7] A. Walenstein, R. Mathur, M. R. Chouchane, and A. Lakhotia, The design space of metamorphic malware. , March 2007.

[8] P. Ferrie and P. Szor, Zmist oportunities, Virus Bulletin, pages 6âĂŞ7, March 2001

[9] P. Haggar, Java Bytecode, July 01, 2001, http://www.ibm.com/developerworks/library/it-haggar_bytecode/

[10] A. Puls, Diving into Bytecode Manipulation: Creating an Audit Log with ASM and Javassist, https://blog.newrelic.com/2014/09/29/diving-bytecode-manipulation-creating-audit-log-asm-javassist/

[11] E. Kuleshov, ASM. http://asm.ow2.org/, accessed November 2014.

[12] Apache Commons BCEL. http://commons.apache.org/proper/commons-bcel/, accessed November 2014.

[13] Code Generation Library. https://github.com/cglib/cglib, accessed November 2014.

[14] S. Chiba, Javassist. http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/, accessed November 2014.

[15] A. White, Serp. http://serp.sourceforge.net/, accessed November 2014.

[16] B. O'Neill, Cojen. https://github.com/cojen/Cojen/wiki, accessed November 2014.

[17] Soot. http://sable.github.io/soot/, accessed November 2014.

[18] S. Chiba and M. Nishizawa, An Easy-to-Use Toolkit for Efficient Java Bytecode Translators, http://www.csg.ci.i.u-tokyo.ac.jp/paper/chiba-gpce03.pdf

[19] A. Einarsson and J. D. Nielsen, A Survivor's Guide to Java Program Analysis with Soot, http://www.brics.dk/SootGuide/sootsurvivorsguide.pdf

[20] E. Bruneton, R. Lenglet, T. Coupaye, ASM: a code manipulation tool to implement adaptable systems.

[21] E. Bruneton, ASM 4.0 A Java bytecode engineering library, http://download.forge.objectweb.org/asm/asm4-guide.pdf

[22] E. Kuleshov, Using the ASM framework to implement common Java bytecode transformation patterns, http://asm.ow2.org/current/asm-transformations.pdf

[23] A. Kalbhor, ''A Tiered Approach to Detect Metamorphic Malware With Hidden Markov Models'', Master's report, Department of Computer Science, San Jose State University, 2014, http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1356&context=etd_projects

[24] A. Venkatesan, ''Code Obfuscation and Virus Detection'', Master's report, Department of Computer Science, San Jose State University, 2008, http://scholarworks.sjsu.edu/etd_projects/116/

[25] M. Stamp, ''A Revealing Introduction to Hidden Markov Models'', Department of Computer Science, San Jose State University, 2012 https://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf

[26] W. Wong and M. Stamp, ''Hunting for Metamorphic Engines'', Department of Computer Science, San Jose State University, http://vxheaven.org/lib/aww00.html

[27] ClassVisitor, ASM. http://asm.ow2.org/asm40/javadoc/user/org/objectweb/asm/ClassVisitor.html, accessed September 2014.

[28] J. Borello and L. Me, ‘‘Code Obfuscation Techniques for Metamorphic Viruses’’, Feb 2008, http://www.springerlink.com/content/233883w3r2652537/

[29] S. Gupta, N. Gupta and R. Gupta, ‘‘Objects and Method Calling in Java Virtual Machine’’, Jan 2014, http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.429.682&rep=rep1&type=pdf

[30] Java. http://www.java2s.com/, accessed November 2015.

[31] J. Memon, and F. Memon, ‘‘Preventing Reverse Engineering Threat in Java Using Byte Code Obfuscation Techniques,’’, Nov 2006, http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=4136912&url=http%3A%2F%2Fieeexplore.ieee.or%2Fxpls%2Fabs_all.jsp%3Farnumber%3D4136912

APPENDIX

Threshold Approach Results

A.1 4-State HMM for combination of all three techniques

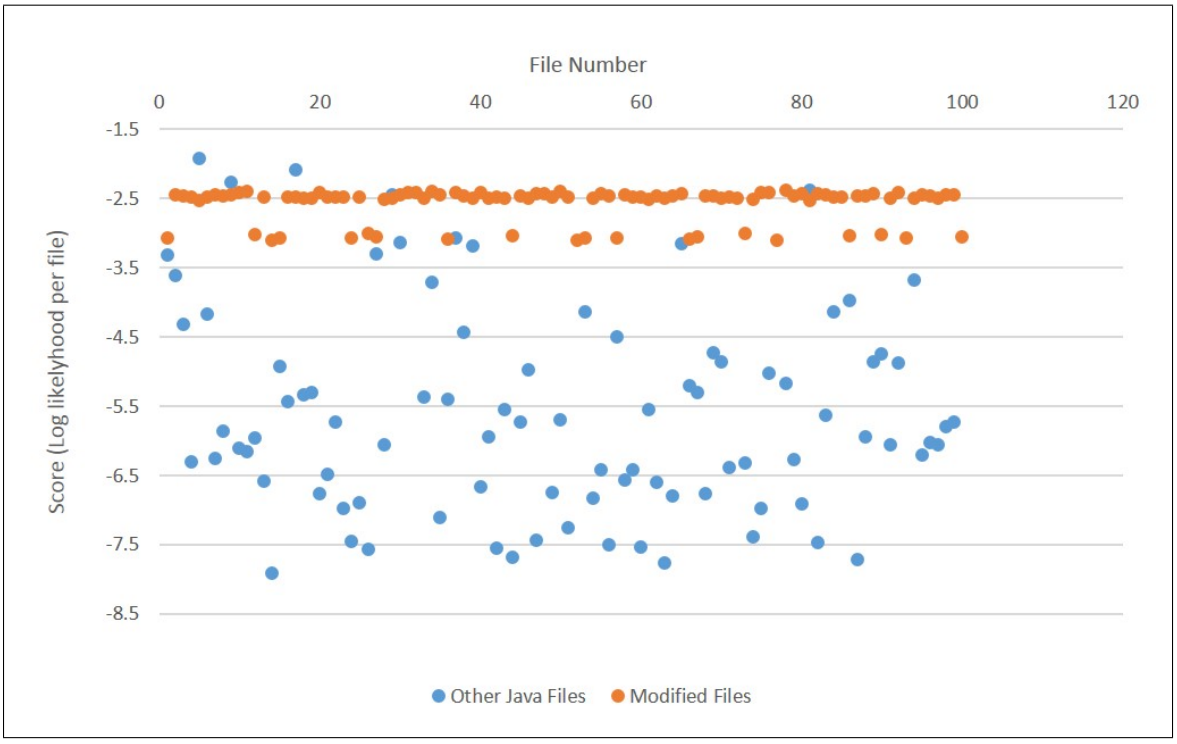

Figure A.27: Combination of all three techniques

A.2 4-State HMM for Subroutine Permutation

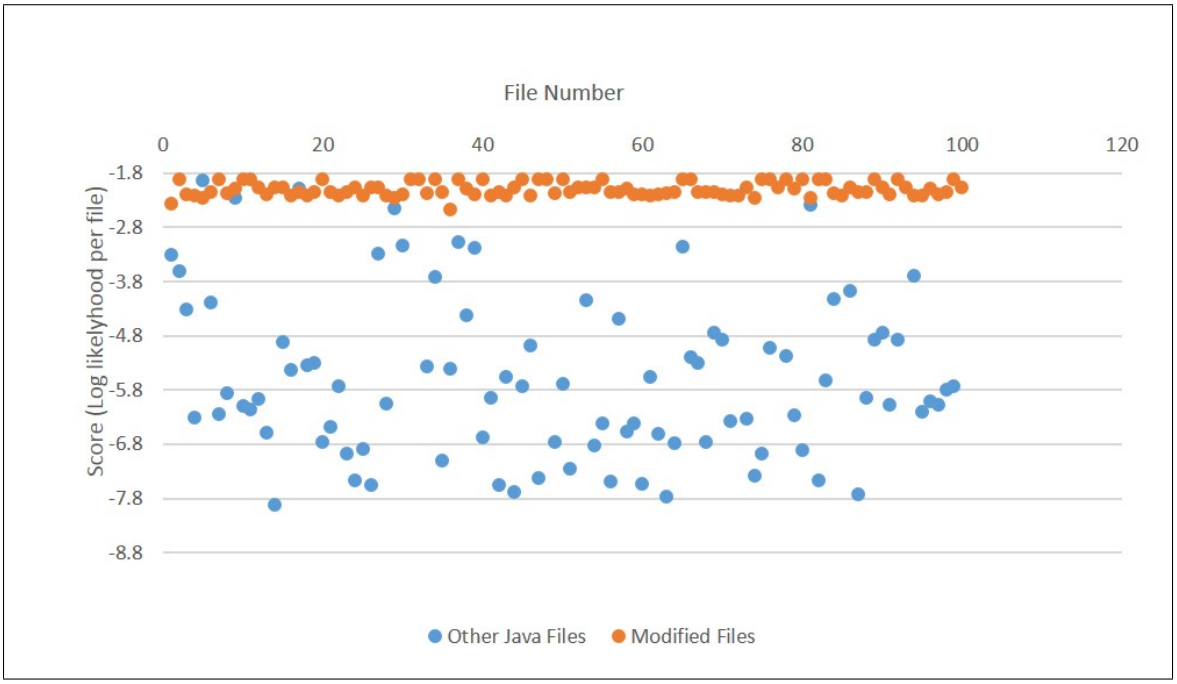

Figure A.28: Subroutine permutation HMM

A.3 4-State HMM for Instruction Permutation

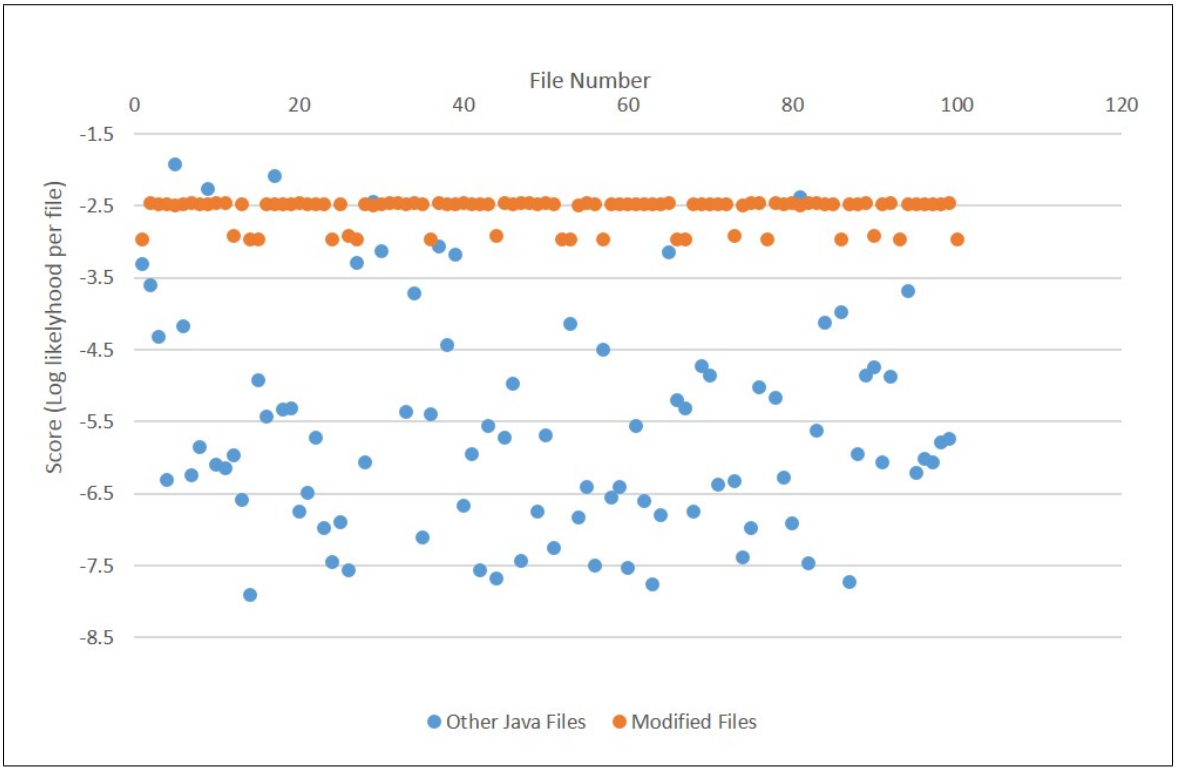

Figure A.29: 4-State HMM for instruction permutation

A.4 4-State HMM for Dead-code Insertion and Subroutine Permutation combination

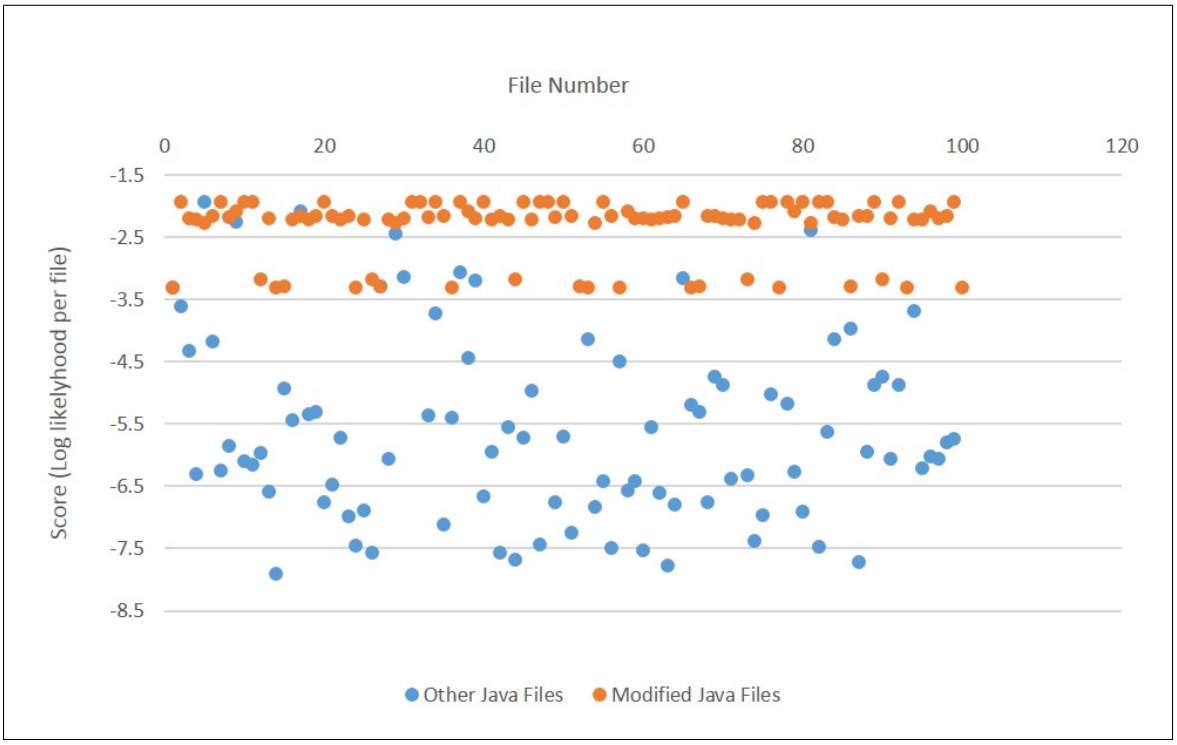

Figure A.30: 4-State HMM for Dead-code Insertion and Subroutine Permutation combination

A.5 4-State HMM for Dead-code Insertion and Instruction Permutation combination

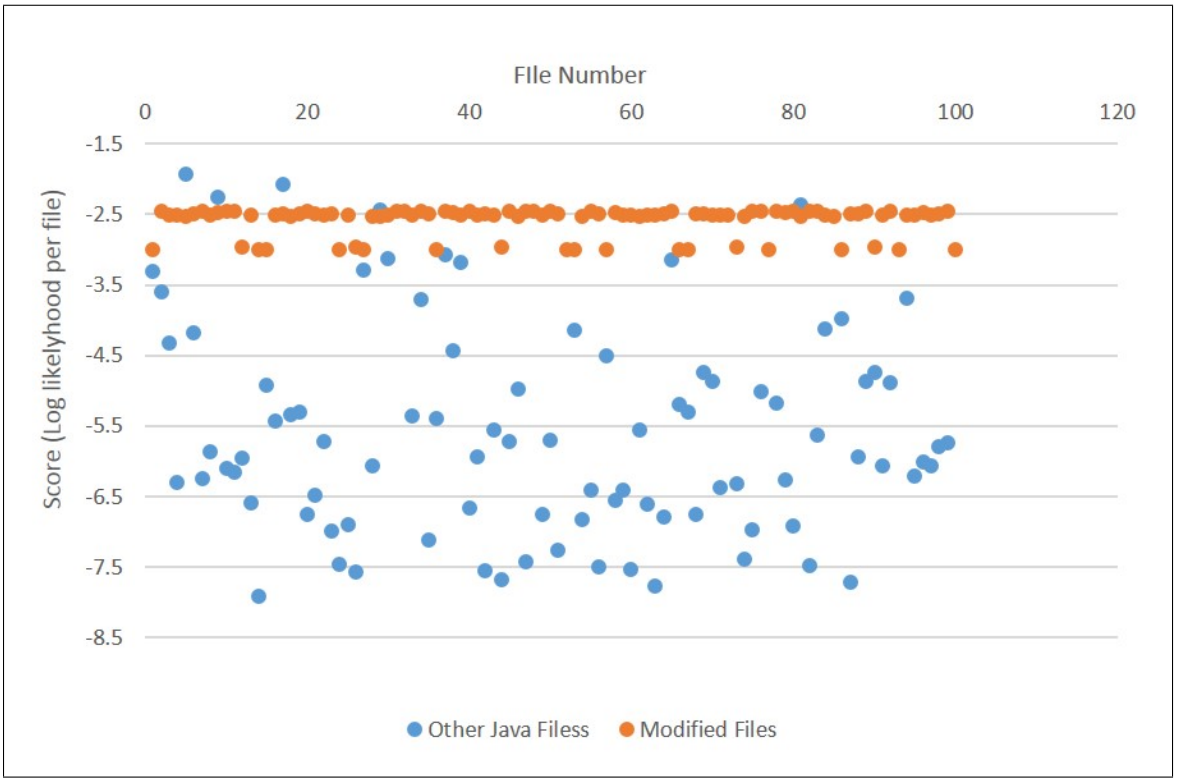

Figure A.31: 4-State HMM for Dead-code Insertion and Instruction Permutation combination

A.6 4-State HMM for Subroutine Permutation and Instruction Permutation combination

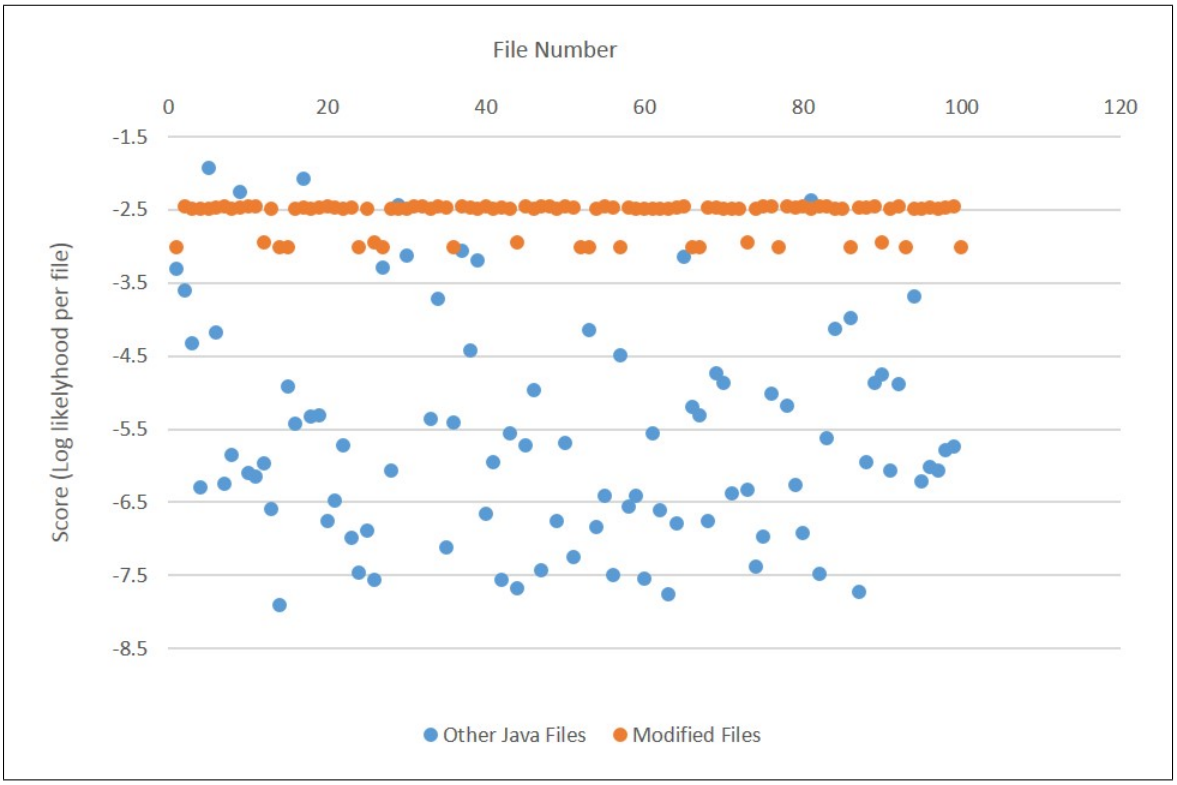

Figure A.32: 4-State HMM for Subroutine Permutation and Instruction Permutation combination