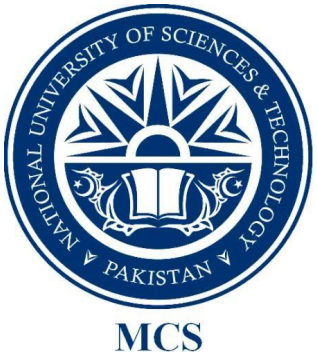
METAMORPHIC TECHNIQUES AND THEIR APPLICATION ON HAVEX MALWARE



By

Zainub Mumtaz

A thesis submitted to the Faculty of Information Security Department, Military College of Signals, National University of Science and Technology, Pakistan in partial fulfillment of the requirements for the degree of MS in Information Security

August 2017

SUPERVISOR CERTIFICATE

This is to certify that **NS Zainub Mumtaz** student of **MSIS-12** course has completed her MS thesis titled **'Metamorphic Techniques and their application on Havex Malware'** under my supervision. I have reviewed her final thesis copy and I am satisfied with her work.

_____

Dr. Mehreen Afzal

ABSTRACT

Most of the commercial antiviruses are signature based, that is, they use existing database signature to detect the malware. Malware authors use code obfuscation techniques to evade detection by antiviruses. Metamorphic malware change their internal structure hence evading signature based detection. Different detection techniques can be found in literature to detect the obfuscated malware also. Havex is an exclusive malware used in cyberespionage campaign launched by a group of attackers, Dragonfly or Energetic Bear, which infects OS Windows. It has targeted multiple sectors so far such as industrial/machinery, manufacturing, pharmaceutical, construction, education and Information Technology. Its main target is ICS and SCADA systems. Havex uses multiple attack vectors to compromise the targets with the coordination of Command and Control infrastructure to download the set of payloads.

For effective defense against the malware, their construction needs to be explored. This includes the study of different obfuscation techniques and possibilities of their extension. This thesis focuses on obfuscation technqiues of malware including dead code insertion, instruction substitution and function permutation. The objective is to make detection difficult by implementing subject techniques which bypass detection. Havex malware is used as a proof of concept for our antivirus evasion strategy. We have used Hidden Markov Models (HMM), which is a statistical based machine learning detection method, to test the effectiveness of our code morphing. This has shown the strength of our implemented obfuscation techniques.

DECLARATION

I hereby declare that no portion of work presented in this thesis has been submitted in support of another award or qualification either at this institution or elsewhere.

_____

Zainub Mumtaz

DEDICATION

"In the name of Allah, the most Beneficent, the most Merciful"

This thesis is dedicated to my parents, husband and siblings who have graciously endured my busy schedule during the degree and been the source of constant support and encouragement for me.

ACKNOWLEDGMENT

I would like to thank my supervisor Dr. Mehreen Afzal, Professor, MCS, NUST, for her continuous support, perpetual motivation, enthusiasm and immense knowledge sharing. I am indebted to her for the valuable time and energy that she gave to this work. This thesis would never have happened without her help and patience.

I owe my sincere thanks to all my Guidance and Examination Committee members (GEC), Dr Monis Akhlaq, Dr Baber Aslam and Lecturer Waleed Bin Shahid for their encouragement, insightful comments and hard questions.

I am extremely grateful to Dr Imran and Dr Abdul Ghafoor for their unprecedented support and information sharing.

I am thankful also to all faculty members of institute for their continuous encouragement and guidance throughout the course of this research work. Last, but by no means least, I am grateful to my junior Mr. Zohaib Shahid and my friend Tehreem Nisa for their support, guidance and encouragement.

TABLE OF CONTENTS

LIST OF FIGURES

LSIT OF TABLES

Tables **Caption** **Page No**

GLOSSARY

AST Abstract Syntax Tree

G2 Second Generation virus generator

HMM Hidden Markov Model

IR Intermediate Representation

LLVM Low Level Virtual Machine

MPCGEN Mass Code Generator

NGVCK Next Generation Virus Creation Kit

VCL32 Virus Creation Lab for Win32

Chapter 1

INTRODUCTION

1.1 Introduction

This chapter gives the introduction of the thesis by discussing the metamorphism performed on different commercial and research based metamorphic engines. Section 2 discusses the overview of the metamorphism and a few engines that are available readily online to perform the obfuscation on malware. It also discusses about the research based metamorphic engines that are developed previously to test the strength against the Hidden Markov Model detection system and the difference of our engine from the former ones. Section 3 states our problem statement. Section 4 enumerates the objectives of our research. Section 5 discusses the sequence of the thesis.

1.2 Overview

As the technology is becoming advanced, the security embedded in it is also reaching the new highs. Malware authors are also using complex techniques to evade the detection by anti-viruses. Metamorphism is considered one of the strongest techniques to bypass detection by changing the structure of the code in each copy of the malware whereas the functionality remains the same.

Metamorphism can be an obstacle for pattern recognition of the code therefore it is used as an additional security measure in many software to avoid different kinds of attacks. Metamorphic code can also be used to bypass signature based detection and other forms of detection techniques **???**. According to the previous researches **??** done in the domain, Hidden Markov Model (HMM)

based detectors have been considered successful in detection of metamorphic malware which have otherwise been able to bypass commercial anti-viruses.

Some metamorphic engines are already available that can be used for generating structurally different copy of the same code every time. A few examples of such engines include G2 (Second Generation virus generator), MPCGEN (Mass Code Generator), NGVCK (Next Generation Virus Creation Kit), VCL32 (Virus Creation Lab for Win32) and MetaPhor **?**.

Research based metamorphic engines have been developed in **?** and **?** that work on the assembly code level. Morphing at the high-level source code is easier but not as effective as the control over the final executable file is much lesser than the low level source code. In **?** and **?**, metamorphic engine has been developed based on Low Level Virtual Machine Compiler Framework. In this, code morphing is done on IR bytecode level which is beneficial in numerous ways. LLVM provides support for multiple languages and compile-time, link-time, run-time and idle-time optimizations for codes written in these languages. In this project, we are extending the work done in **?** and **?** by designing metamorphic engine which works on LLVM bytecode level.

LLVM has a three-phase compiler design with a front end, an optimizer and a back end that supports multiple languages and target platforms. Front-end takes source code as input and performs parsing of the code, checks it for errors and builds the Abstract Syntax Tree (AST) to represent the input code.

The optimizer performs different optimizations on the code to reduce the time for execution of the code by discarding off the redundant computations and unused instructions. The optimizer is language independent. The back end translates the code correctly to the target framework utilizing the random features of the target architecture. Code morphing on LLVM IR bytecode level is efficient as it provides fine grained control over the resulting executable file.

Our developed metamorphic engine is tested against the Hidden Markov Model (HMM) analysis **?**. We are using three metamorphic techniques in our engine include dead code insertion, instruction substitution and subroutine permutation.

1.3 Problem Statement

Although metamorphic engines developed recently have shown the ability to bypass HMM based detection, there are some limitations in their implementations. There are more code obfuscation techniques as well which can be explored along with the limitations of the previous implementations to bypass HMM based detection. It is important to understand and extend already existing implementations of metamorphic engines to understand the weaknesses of our detection strategies like Hidden Markov Model.

1.4 Objectives

The main objectives of thesis are discussed below:

1. To understand, implement and analyze the limitations of already existing implementations of metamorphic techniques.

2. To extend already existing implementations of metamorphic engines.

3. To provide proof of concept by using Havex malware.

4. To analyze the effectiveness using HMM detection strategy.

5. To take account of factors which help in improving the results.

1.5 Research Methodology

First of all, a thorough literature review was carried out on the metamorphic techniques and the previous work done to increase the strength and robustness of the metamorphic techniques. Limitations of the existing implementations were studied in detail. After this, modifications in the previously implementations were proposed and implemented. Once the implementation phase was complete, Havex malware was used for the proof of concept. Test files were generated by applying obfuscations on the Havex malware using Linux coreutil-8.23 files and tested against the HMM malware. The results were generated and recommendations were provided to improve the results.

1.6 Contributions

The research contributions of the research are as follows:

1. A metamorphic engine is developed incorporating three different metamorphic techniques in this research. Dead code insertion is performed by integrating dead code from Linux core utility files into the malware file. After this instruction substitution is performed in which one instruction can be replaced by multiple other instructions. These instructions are chosen based on the random number generator. Last phase is the application of function permutation. In this phase, the sequence of functions is randomized and an altogether new code is generated.

2. We have implemented and HMM based detection system and Havex malware is used as the proof of concept. Havex malware has been passed through all the phases of the metamorphic engine and the resultant files are tested against the HMM based detector. We used different percentages of the dead code such as 10%, 20%, 30% and 50% to show how the results differ based on the increasing percentage of the dead code. The results in the form of graphs show the

comparison of HMM scores which determine whether the malware file has been able to evade detection or not.

3. An extensive literature review has been done on the metamorphic techniques and a research paper was accepted in 5th International Cryptology and Information Security Conference held in Malaysia on June'2016.

1.7 Sequence of Thesis

Next chapters of the thesis are categorized in the following ways. Chapter 2 gives the brief description of different types of malware and different malware obfuscation techniques adopted by the malware. It also discusses the metamorphic techniques that are used by the metamorphic generators. Chapter 3 discusses the environment of metamorphic engine generation and that of metamorphic malware detection. Its overall design of LLVM compiler infrastructure, the detailed description of the IR byte code, different types of malware detection techniques and the planning and execution of the Hidden Markov Model used to evaluate the test results is covered. The architecture and implementation of the code generator is covered in Chapter 4. Chapter 5 showcases the experimental results. Lastly, Chapter 6 gives the conclusion along with the future work that can be carried out.

Chapter 2

MALWARE AND MALWARE OBFUSCATION TECHNIQUES

2.1 Introduction

This chapter briefs about different types of malware that exist and various obfuscation methods that are used by the malware to evade the detection. Section 2 discusses the varieties of malware that exist in the wild. Section 3 gives the brief details of evolution of malware concealment strategies of the years. Section 4 discusses the main obfuscation techniques used by the metamorphic malware, which is the most complex type of malware, to bypass malware detectors.

2.2 Malware

Malware is any malicious program that is used to harm a system by disrupting its operation, stealing information, illegally using its resources or performing any other malicious activity. Any software with malicious intent comes under the umbrella of malware including computer viruses, worms, trojan horses, bots, adware, spyware, rootkits and other hostile programs. The first malware was in the form of virus with the intention of only highlighting the security loopholes or to boost off technical ability **?**. After that, different classes of malware started infecting the systems and the motivations behind using the malware changed also. The three most prominent categories of malware are viruses, worms and Trojans which are discussed in detail in the subsections below.

2.2.1 Virus

According to the keywords, the following search strings were employed to First PC-based computer virus was developed in 1980s. A virus replicates itself upon execution and propagates by copying itself to other programs. It depends on a host program for execution. It does not have the ability to reproduce itself and needs human interaction for infecting other computers. For instance, it can either be downloaded from the internet or gets transported using floppy disk, CD-ROM, DVD-ROM or USB stick etc. Some of the important features of virus are given in **?**. According to which the viruses are self-replicating, their population growth is positive and they are parasitic in nature.

2.2.2 Worm

Worm is also self-replicating but it is different from the virus in two ways. First, it does not depend on a host program for execution. Second, it does not require human interaction for spreading. In other words, it propagates via computer networks, unlike virus which can only infect while attaching itself to the host program **??**. Some of the important features of virus are given in **?**. According to which the worms are self-replicating, their population growth is positive and they are not parasitic in nature.

2.2.3 Trojan

Unlike virus and worm, trojan disguises itself as a legitimate program but performs malicious tasks in the background. It comes with a browser plugin, a screensaver, an unsuspicious email with attachment or a game etc **?**. It does not try to infect other programs or propagate. It can act as a backdoor to connect to the controller and download additional files and programs, change system settings or infect other files on the system. Some of the important features of virus are

given in **?**. According to which the trojans are not self-replicating, their population growth is zero and they are parasitic in nature.

2.3 Malware Obfuscation Techniques

Malware writers keep on evolving the malware by employing different obfuscation techniques to bypass detection by anti-viruses. These techniques make detection very challenging for antivirus vendors **??**. These obfuscation strategies include encryption, oligomorphism, polymorphism and metamorphism. The history of malware is shown in Figure 2.1.

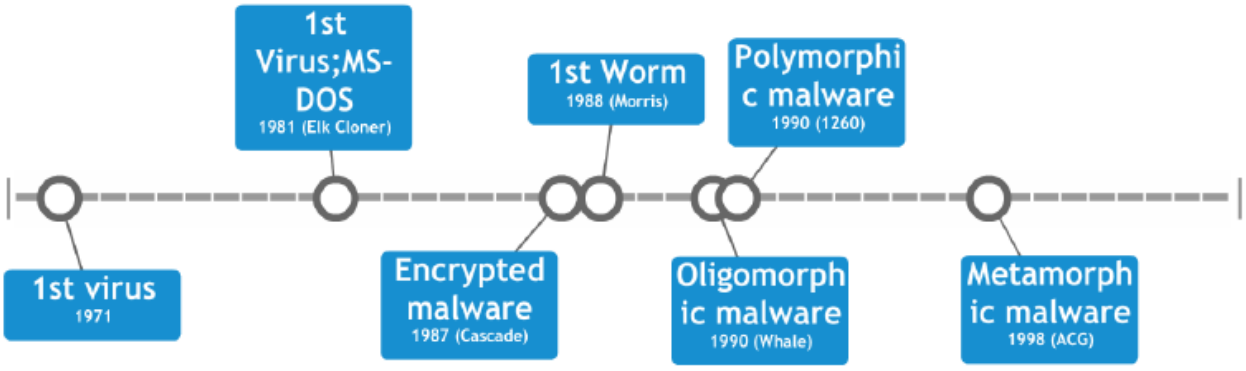


Figure 2.1: History of malware

2.3.1 Encrypted Malware

Encryption of malware first started in 1988 with the virus, Cascade **?**. This is the earliest and simplest form of malware concealment. The purpose is to hide the virus body to make it difficult to analyze by applying encryption algorithms. The resultant virus consists of two parts; one is encrypted virus body and the other one is the decryptor. The decryptor contains the code for encryption and decryption of the virus body. On execution, the virus body is decrypted using a decryptor and a new encrypted copy of virus is created using a different key. Different encryption

keys are used every time for encrypting the virus body which makes it look different in every generation. This is shown in the Figure 2.2. Encryption of malware can be of different types such as simple encryption which involves simple operations without any key, static encryption key, variable encryption key, substitution cipher and strong encryption. Anti-viruses do not detect encrypted malware at once. However, it still gets detected by anti-viruses after they obtain the decryptor using its code pattern.

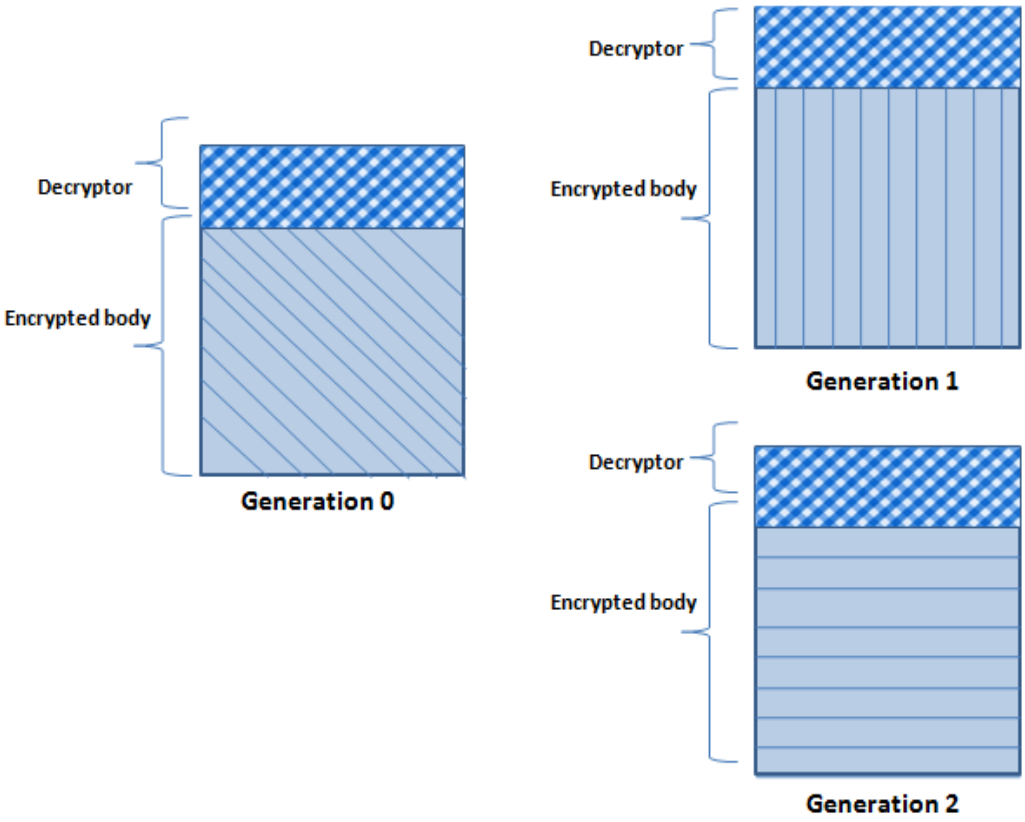


Figure 2.2: Different encryption keys produces different encrypted malware bodies but decryptor remains constant

2.3.2 Oligomorphic Malware

Encrypted malware get detected by anti-viruses because the decryptor remains constant in each generation. The solution to this problem is to create viruses in which body of decryptor is mutated.

Easiest way to achieve this is to use a set of finite decryptor loops and let the virus choose one in each generation. This idea led to the creation of first Oligomorphic virus in 1990 known as Whale **?**. It carried 30 different decryptor loops, from which the virus would choose one. Signature based detection depending on byte pattern of decryptor, though it is a achievable solution, but it is not a practical way since the anti-virus will have to look for multiple decryptor patterns instead of one **?**. Therefore, oligomorphic viruses make the detection process more difficult for signature based scanning engines **?**. This concept is explained in the Figure 2.3.

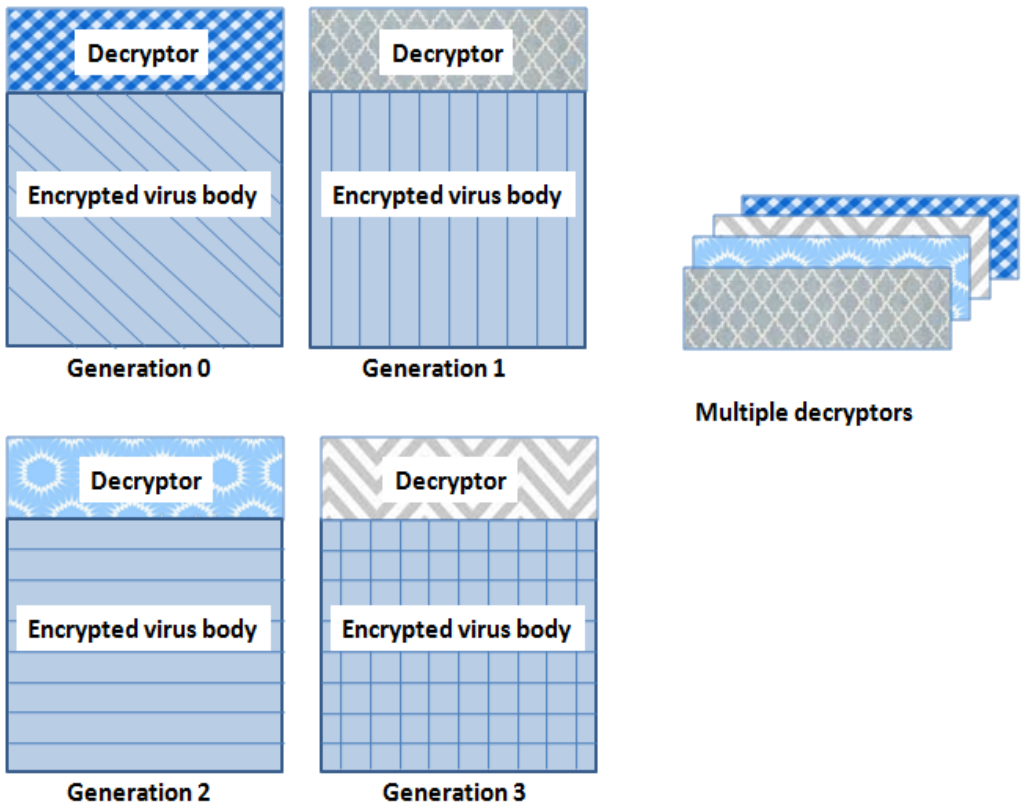


Figure 2.3: A different decryptor is used in each generation of Oligomorphic malware

2.3.3 Polymorphic Malware

Polymorphism is more progressive and sophisticated form of oligomorphism. First polymorphic malware was the 1260 virus **?**. It overcame the limitations of both encrypted and oligomorphic

viruses. It is better than encryption and oligomorphism in the sense that it has the capability of generating unlimited new decryptors **?**. Decryptor code is mutated from generation to generation in such malware. This is done using a mutation engine which uses various code obfuscation techniques such as instruction substitution, garbage code insertion, register swap etc **??**. Detailed description of these techniques will be covered in Section 3. Furthermore, each new decryptor may use several encryption techniques to encrypt the constant virus body, as well **??**. This concept is explained in the Figure 2.4

Although polymorphic malware is not subjected to signature based detection as it lacks unique decryptor patterns, it can still be detected using code emulation technique. Anti-viruses detect polymorphic malware using this technique by executing the viruses in the virtual environment and detecting virus body dynamically.

2.4 Metamorphic Malware and their techniques

Metamorphic engines employ different code obfuscation techniques to generate highly morphed copies of metamorphic malware. The metamorphic techniques change the structure of the malware but the behavior remains the same. Therefore, different copies of the same malware will look structurally different but will perform the same functionality that is why such malware are also known as "bodypolymorphic" **??**.

These code obfuscation techniques are of different types. Some operate on the control flow while others operate on the data section of the program. Control flow obfuscation involves reordering of instructions through insertion of jumps. Data flow obfuscation can be done in many ways such as equivalent code substitution, subroutine permutation, dead code insertion, register renaming, and transposition **???**.

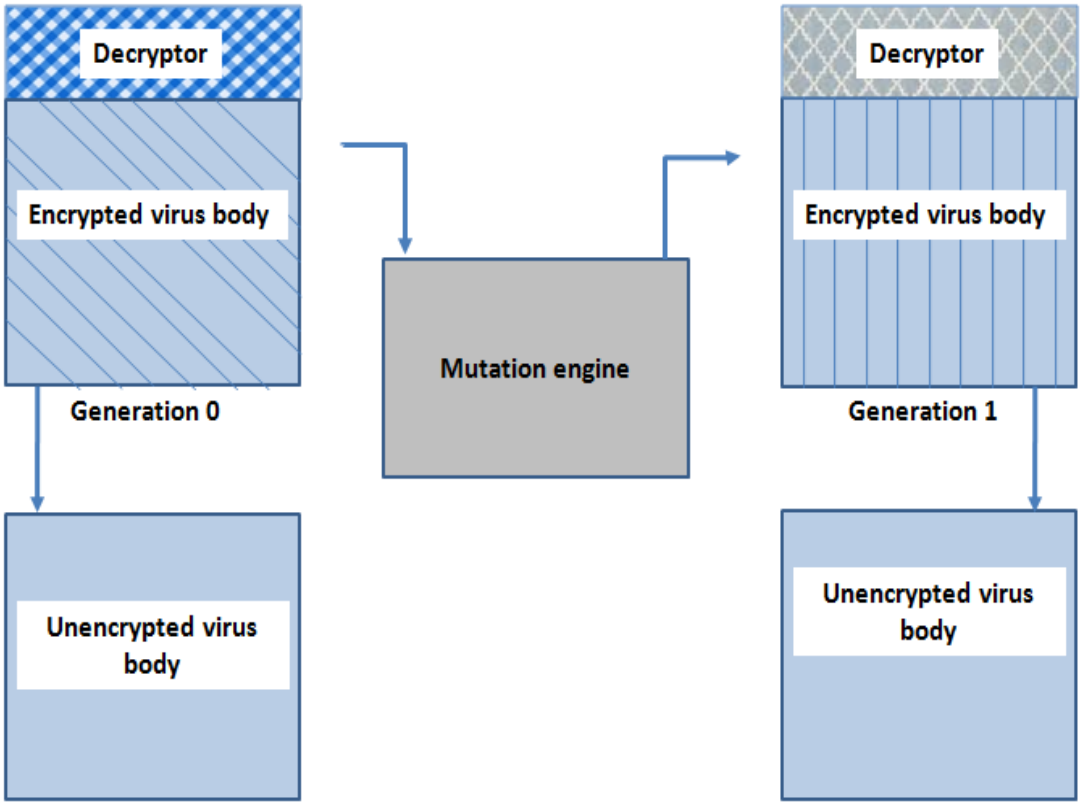


Figure 2.4: Infinite decryptors are produced by mutation engine

2.4.1 Register Swap

This is the simplest technique for metamorphism of malware.The first metamorphic malware, W95/Regswap virus,implemented register swap in December 1998 as a metamorphic technique **?**. Operand registers are swapped with different registers. For example, replacing POP ECX by POP EAX. Now two generations will have the same sequence of instructions but different operands. It can also be named as register renaming and register usage exchange. It can thwart the traditional pattern matching technique for malware detection but it can still be detected using half-byte detection technique and wildcards. An instance of register swap is given in Figure 2.5.

2.4.2 Subroutine Permutation

In this technique, the order of subroutines is changed in each generation to generate malware with completely different layout. This does not affect the behavior of the malware because the execution order is not affected. A virus with n subroutines can have n! different subroutine permutations. This means that a program with 10 subroutines will have 10! subroutine combinations. For example, BadBoy with 8 subroutines and W32/Ghost with 10 subroutines can generate 8! = 40, 320 and 10! = 3, 628, 800 different generations respectively **?**. However, both of them can be detected with search strings as the content of each subroutine remains the same **??**. Both register swapping and subroutine permutations are simple techniques with respect to statistical based detection. This concept is explained in the Figure 2.6.

2.4.3 Dead code insertion

This is another technique to change the appearance of the malware which helps in bypassing statistical based detection. This dead code has no affect on the behavior of the program in which it is added. Win95/Zperm virus appeared in June and September of 2000 incorporated this technique **??**. This malware is shown in the Figure 2.7.

Dead code can either be executed in the program or it can just stay within the code without being executed. However, it will not affect the program functioning in either case. Dead code, when executed, makes it trickier for the optimizer to detect. Dead code can be added from other files in the form of functions or blocks. This concept is explained in the Figure 2.8.

It can be in the form of simple do-nothing instructions such as 'NOP', 'add ecx,0','mov ebx, ebx', 'sub eax,0' or instructions from normal (benign) files can be copied and inserted into malware file as mentioned in **????**. Infinite generations of malware can be produced using these infinite sets

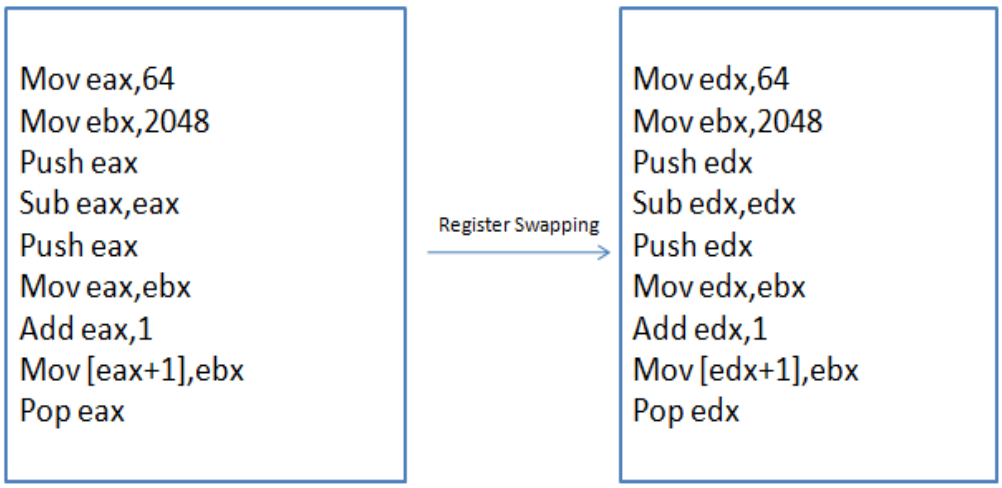


Figure 2.5: Register Swap

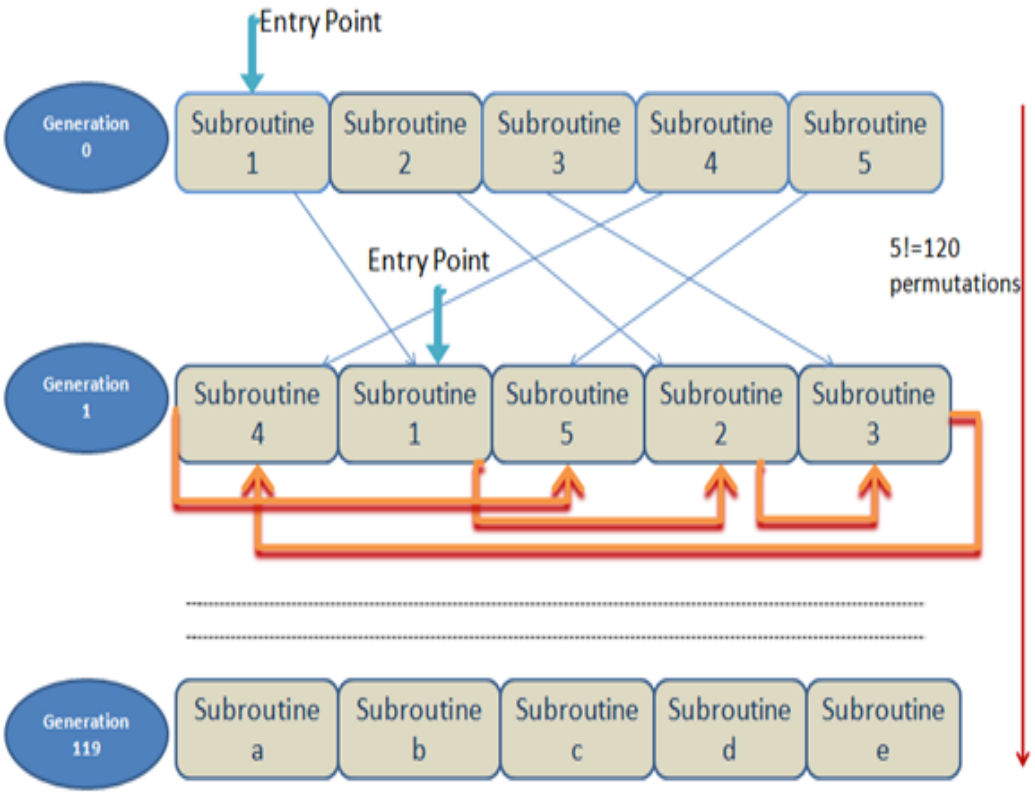


Figure 2.6: Subroutine Permutation for 5 different subroutines

of dead instructions.

2.4.4 Instruction Substitution

Instruction substitution or equivalent code substitution involves replacement of an instruction by another instruction or a group of instructions that are equivalent to the original instruction and have the same functionality.

Some opcodes are characteristic to the malware files so it is reasonable to replace these opcodes to make these files look more like normal files using instruction substitution. For example, add eax,1 can be replaced by not eax, neg eax and inc edx can be replaced by add edx,1, not edx, neg edx. Instruction substitution is a powerful technique for evading signature detection and altering code statistics.However, instruction substitution is relatively difficult to implement at the assembly code level. The W32/MetaPhor virus is one of the metamorphic virus generators that incorporate the instruction substitution technique **??**. One such example is given in the Figure 2.9.

2.4.5 Transposition

Reordering the instructions to change the sequence without affecting the behavior of the code is called transposition or instruction permutation. This will make the binary sequence of the code different than the original code bypassing signature based detection. However, this can be done only if the two instructions are independent of each other. An example of instructions is given in the Figure 2.10.

Figure 2.11 gives the 3 conditions which are pre-requisite to the reordering of instructions.

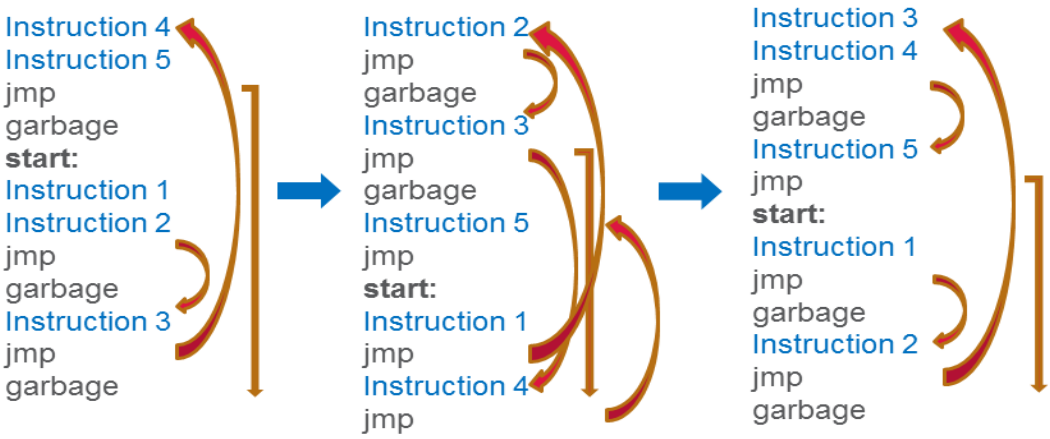


Figure 2.7: Win95/Zperm virus involving Subroutine Permutation

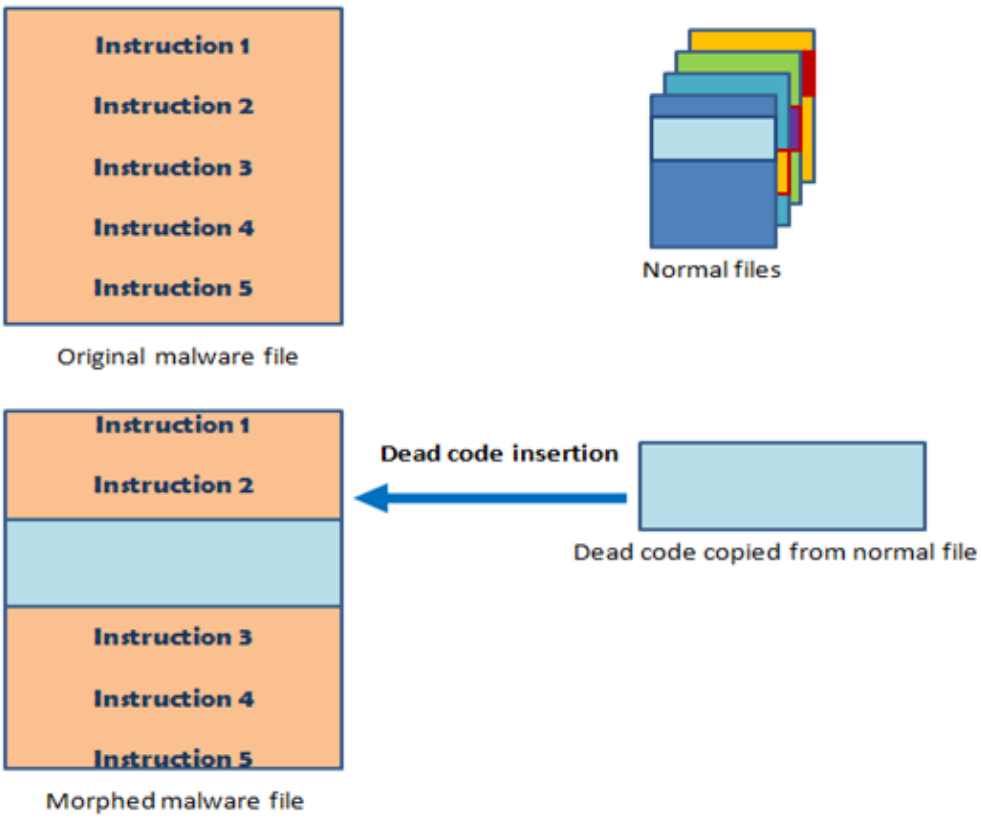


Figure 2.8: Dead Code Insertion

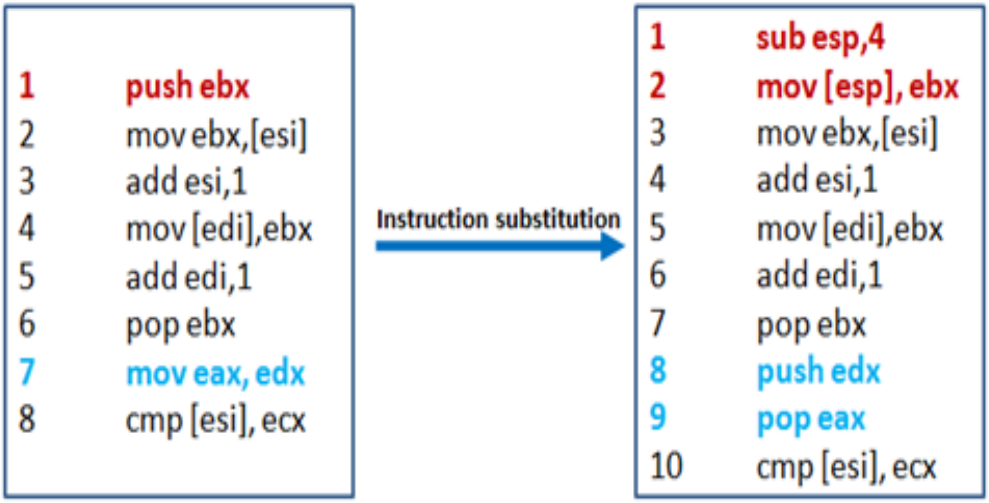


Figure 2.9: Instruction Substitution

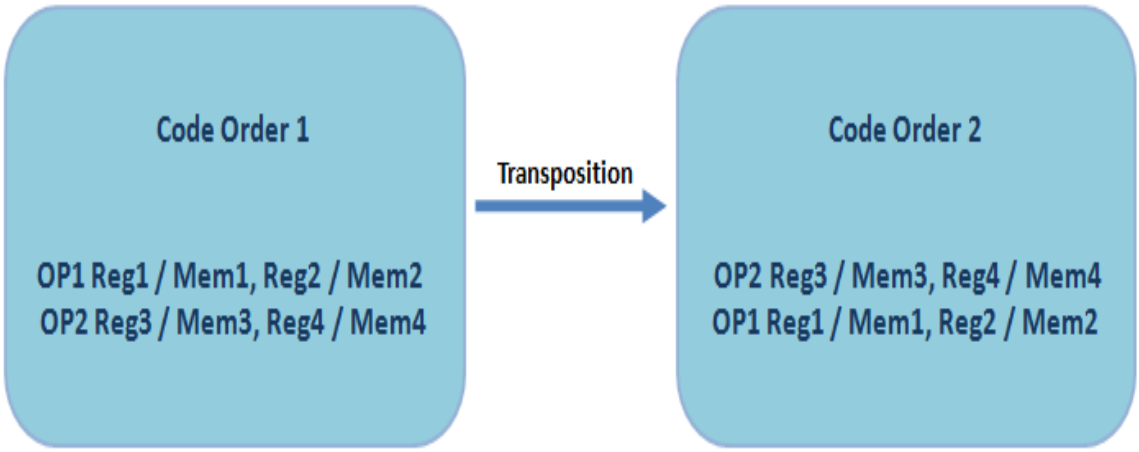


Figure 2.10: Transposition

2.4.6 Formal Grammar Mutation

Formal grammar mutation is a formalization of many existing morphing techniques. Formal Grammar Mutation was used for the first time for the formalization of code morphing techniques in **??**. In general, traditional morphing engines are non-deterministic automata (NDA), since transitions are possible from every symbol to every other symbol. The symbol set is the set of all possible instructions. It means, any instruction can be followed by any other instruction. By formalizing mutation techniques, one can create formal grammar rules and can apply these rules to create viral copies with great variation **?**.

Two different forms of a simple polymorphic decryptor code are shown in the Figure 2.12. It is possible to make 960 different decryptors using this sample and Formal Grammar Mutation shown in the Figure 2.13 **?**.

2.5 Conclusion

In this chapter, different malware types were discussed including viruses, worms and trojans along with their important features. The malware has evolved over the years to change its appearance and make detection more complex. The different evolutionary forms of these malware give the anti-malware vendors all the more reasons to make the detection strategies more effective. Metamorphic malware is the most complex type of malware obfuscation. It implements various metamorphic techniques to change the structure of the malware. These techniques are also discussed in this chapter.

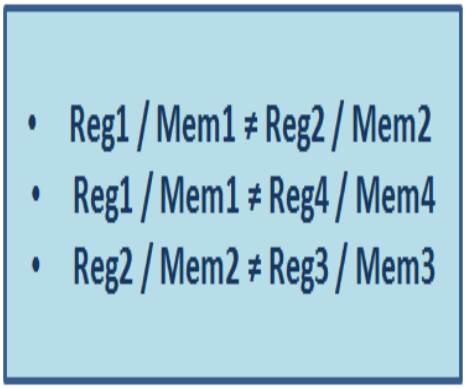


Figure 2.11: Conditions to meet for Transposition

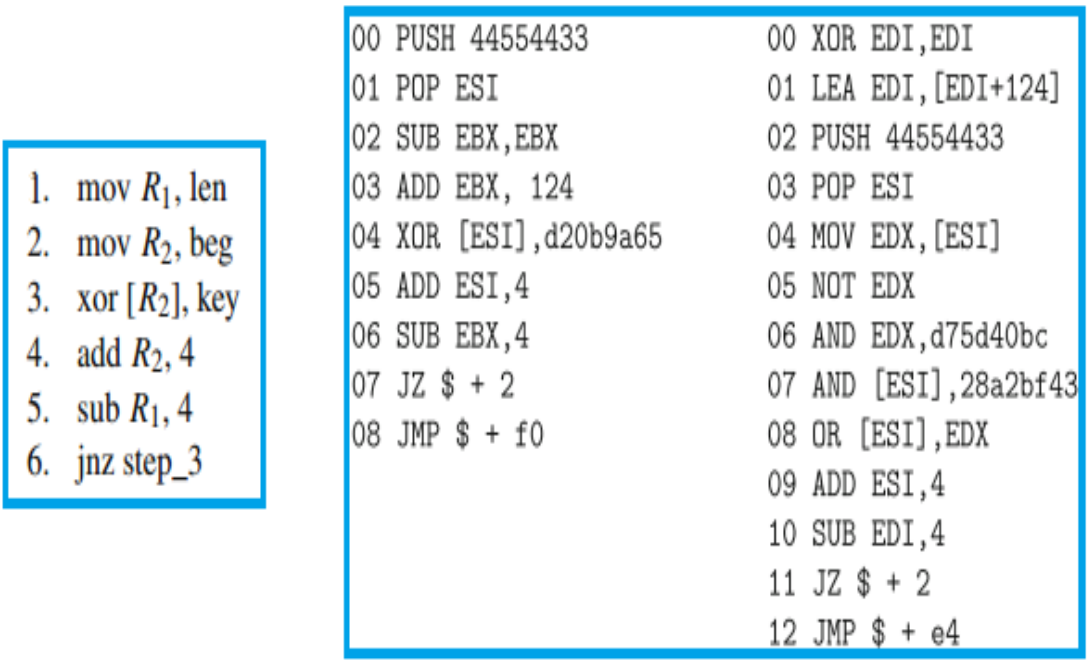


Figure 2.12: A simple polymorphic decryptor template and its mutated code

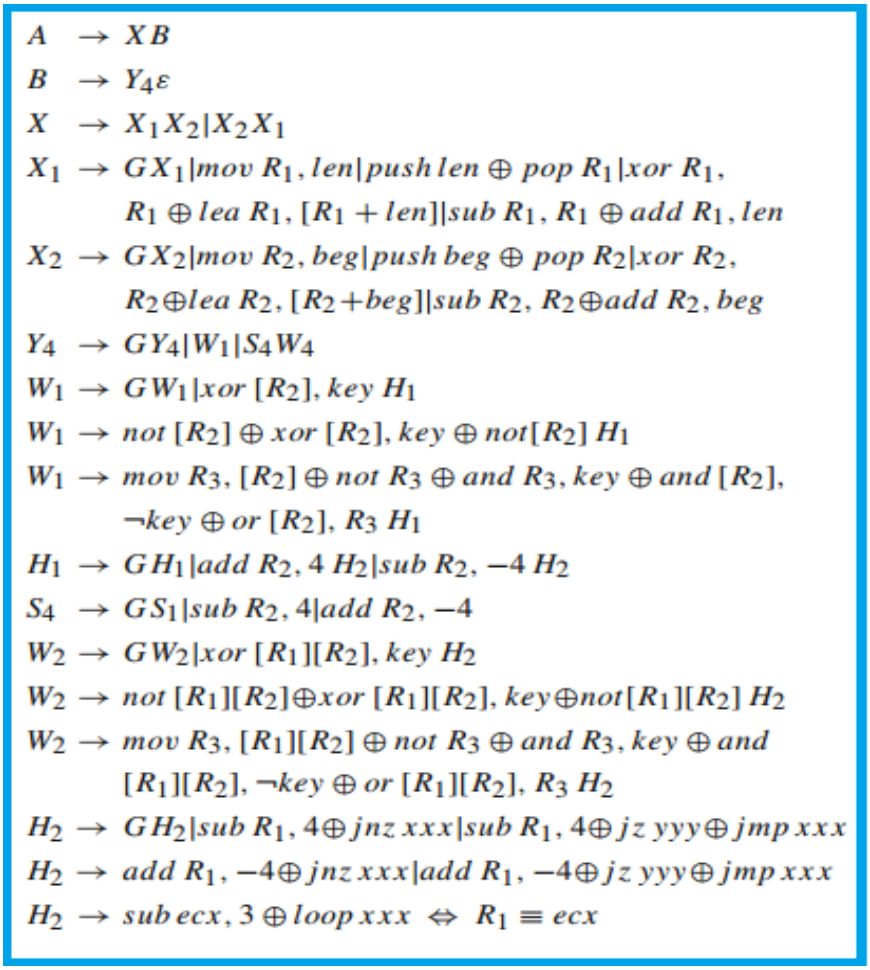


Figure 2.13: Formal Grammar for creating decryptor variants

Chapter 3

ENVIRONMENT OF MALWARE OBFUSCATION AND DETECTION

3.1 Introduction

In this chapter, the discussion is focused on the environment used for implementing the metamorphic engine. In section 1, it tells that LLVM is used as a compiler for writing in metamorphic code. It also discusses the benefits, the tools used in this thesis and salient features of using this framework. Section 2 briefs about the malware detection techniques used by the anti-malware vendors to detect the malware. Section 3 discusses the environment used for metamorphic malware detection.

3.2 Environment for Metamorphic Engine

For the creation of our metamorphic code generation, we have used the LLVM compiler framework. LLVM is an umbrella project that is responsible for the hosting and developing of many low level tool components that are in compatibility with the existing tools such as Unix tools. Metamorphic code generation is done on IR bytecode level using LLVM in this project instead of assembly or source code level. Intermediate Representation is also not dependent on the platform. Virtual addresses are assigned at the bitcode level not at the assembly level. Therefore, developing the metamorphic copies on IR level rather than assembly level provides benefits over the assembly level.

3.3 LLVM

LLVM has many distinct features such as tools which make it better than the GCC compiler. For example, the Clang Compiler which is a C/C++/Objective-C compiler. However, the major feature which makes LLVM unique than other compilers is its design. LLVM is a common infrastructure and it implements a vast range of statically and dynamically combined languages such as Java, .Net, Python, Ruby, languages supported by the GCC and many other lesser known languages **??**.

3.3.1 LLVM Compiler Design

Three phase design of the tradition compiler is the most famous design of the static compiler which has three main components; front end, back end and the optimizer. The front end takes the input source code, parses it and checks it for the errors and then creates an Abstract Syntax Tree (AST) based upon the source code. This AST can also be converted to a new representation for optimization. The purpose of the optimizer is to perform numerous transformations of the code as required in order to optimize the code and reduce its running time. These transformations can be of various types such as elimination redundant instructions, removing dead code, and it is mostly not dependent on language and target. Now, the back end which is also known as code generator maps the code on the target architecture.

3.3.2 Benefits of using LLVM

An important advantage of using LLVM is that compiler supports multiple languages or target architectures. A common optimizer can be used for a front end written in any language or a back end for any target architecture can compile from it. This is shown in figure. This means that for supporting a new language, only the front end for that language has to be written and the optimizer

and back end can be reused. Similarly, to support new target architecture only the new back end has to be written while front end and optimizer can be reused. If the compiler was not based on the three phase design and a new language had to implemented, a new compiler would be needed for it. N * M compilers would be needed to support N languages and M target architectures.

Another important advantage of this compiler is that different skills are required for front end, back end and optimizer. This makes it easier for a front end individual to boost and maintain their part of compiler and can focus on their part only **??**.

3.3.3 LLVM IR

An important aspect of this compiler design is LLVM Intermediate Representation (LLVM IR). It supports the mid-level analysis and different forms of transformations of the compiler. Different actions can be performed on it such as simple optimizations, inter-procedural optimizations, whole program analysis etc. LLVM IR is a first class language with well structured semantics.

LLVM is a low level RISC-like virtual machine which performs linear operations such as addition, subtraction, comparison and branching. It also supports labels and is an unorthodox form of assembly language. It is different from RISC-like instruction set in a way that it is strongly typed. For example, i32 means a 32-bit integer, i32** means a pointer to a pointer of 32-bit integer and some abstraction of machine details is there. Instead of using permanent registers it uses infinite number of temporary registers with a % character.

It can take three forms such a textual form, on-disk bitcode format which is dense and data structure supported in-memory form. Intermediate Representation of LLVM consists of three components:

1. Module: it consists of functions and global variables

2. Functions: they represent the instruction set

3. Global variables: they consist of the values that are used by the functions

3.3.4 LLVM Tools

LLVM has different kinds of tools that can be used to perform different tasks such as program compilation, generation of IR bytecode and performing different kind of optimizations on the code. A number of tools are used in this thesis like llvm-gcc, llvm-as, lli, llvm-dis, llc, llvm-link and opt. The details and syntax for using these tools is discussed in the Appendix.

3.3.5 Opt (Command for running optimizations)

This is LLVM optimizer and analyzer. IT bytecode files are taken as an input and different optimizations and/or analyses can be performed and analysis results or optimized output files are generated.

One can also write their own optimizers built on the already existing ones according to the requirements. One example of such optimizer passes is also given in this project in which dead code is called and instruction substitution is performed on it. LLVM already has many existing "pass classes" as well and new passes can also be developed. Opt command has the following format.

opt < optimizer name >< input bytecode or bitcode file >< output bytecode or bitcode file >

3.4 Malware Detection Techniques

Malware is becoming increasingly powerful every day. The techniques used to detect malware are also evolving to prevent the malware from infecting the computer systems. There are three

main categories for malware detection techniques.

1. Signature based detection

2. Anomaly based detection

3. Statistical Analysis based detection

3.4.1 Signature based detection

This is the simplest and the most frequently technique and for the detection of malware used by anti-viruses **??**. The repository or database of signatures of malware is maintained and is used to pattern match with any malware that may try to infect the system. This database of signatures is obtained by examining the disassembled code of the malware. The database is regularly updated by the anti-virus to include the new signatures. The advantages of this technique are; it is simple, has great accuracy and fewer resources are required to detect the malware **?**. However, it can only detect known malware since it will not have signature for any new or unknown malware. It cannot detect complex malware such as polymorphic (without using code emulation) and metamorphic malware. Another disadvantage is that, the repository has to be updated regularly to include new signatures **?**.

3.4.2 Anomaly based detection

This is also known as behavior based or heuristic based detection. This overcomes the problems of signature based detection by detecting unknown malware or zero-day attacks. It has two phases; one is, training phase and the other is, detection phase **?**. Detection systems learns the behavior of the known malware in the training phase and detects unknown or new malware based on that training. One method of analysis is to execute the malware in the virtual machine and analyze its

behavior. If malware shows suspicious activities such as creating files, overwriting files, hiding files and replication etc, then it is flagged as malware **?**.

Another method of detection is to analyze the decompiled code of the malware. Some instructions are more common in malicious files than normal files. If the percentage of instructions that are more common in malware files crosses the threshold, the files is flagged and alert is generated. The disadvantage of this technique is that it needs to update the data describing the system behavior and the statistics in normal profile but it tends to be large. It needs more resources like CPU time,memory and disk space **?**. Also, it can only detect malware based on previous experience. This means if a new malware is different than the previous ones, it may go undetected. Due to this, the rate of false alarms generated is very high **??**.

3.4.3 Statistical Analysis based detection

Statistical analysis and machine learning techniques are also being used for detection of malware. One such technique is the application of Hidden Markov Model (HMM). This is a comparatively newer technique which has two phases **????**. First phase is representing the statistics of the malware sample set on the HMM model and training the model. Second phase is for detection of malware belonging to the same family as the sample malware set. It has been found that Profile Hidden Markov Model is very effective for malware detection. A drawback of this method is; in order to detect the malware, the detection system has to be trained first for which data set is needed. Secondly, this whole process of filtering the data, disassembling them, training and scoring the whole dataset can be time consuming **?**.

3.5 Environment for Metamorphic Malware Detection

Metamorphism is considered as one of the strongest techniques to evade detection by anti-viruses. However, Hidden Markov Models (HMMs) have been used previously and are considered effective in detecting the metamorphic malware **???????**. It is one of the machine learning techniques used for detecting the malware. Metamorphic engines change the appearance of the malware but there is still some similarity between malware belonging to the same family. For this, a method has been devised in the past to train the HMM based on the sequence of opcodes belong to the same malware family **??**. HMM builds a training model based on the input opcode sequences belonging to the same family. This model can be used to score the binaries and to classify the test file as benign or a member of a malware family that the model is representing. A threshold is defined for virus and benign files which is used to classify the executables as malware or benign. A detailed working of HMM along with its usage for malware detection is explained in this section.

Hidden Markov Models (HMMs) are used for statistical pattern analysis. The Hidden Markov Model describe observation series generated by Markov or Stochastic process. A Markov process is related to the sequence of observed symbols. The Markov model in HMM is "hidden" which means that it is not directly observable but it can be seen as the sequence of observation associated with the states **???**. Table 3.1 shows the notations used in Hidden Markov Model **?**.

Previous researches carried out on HMM proved that HMM is effective for different types of modelings such as protein modeling and speech recognition. It has also been used for the software piracy detection **??**. An HMM can be trained using an input data. Based on the observation sequences, the training models can be generated. Features of the input data are represented by the states in the training models, whereas the statistical properties of these features are represented using transition and observation probabilities. Each individual elements in the training data is

Table 3.1: Hidden Markov Model Notations

Notation	Represents
T	Length of the observed sequence
N	Number of states in the model
M	Number of distinct observation symbols
O	Observation sequence $(O_0, O_1, ..., O_{T-1})$
A	State transition probability matrix
B	Observation probability distribution matrix
π	Initial state distribution matrix

mapped to an observation symbol. New observation sequences are matched against the trained HMM to see if they have any similarity to the one in the training model. One training model is made for malware from a single family. Now whenever a new file comes, it is tested against these models. If the probability to see such a sequence is high, it means that the sequence has similarity with the malware family **???**.

The three algorithms used to solve the three problems using HMMs are Forward Algorithm, Backward Algorithm and Baum-Welch Algorithm. Following are the three problems which need to be addressed using HMMs **??**:

Problem 1. Given that the model $\lambda = (A, B, \pi)$ and an observation sequence O is known, find $P(O|\lambda)$ or likelihood of the observation sequence. Forward Algorithm is used to solve this problem.

Problem 2. The model $\lambda = (A, B, \pi)$ is known, find an optimal state sequence for the Markov process. This optimal state sequence is found out using the Backward Algorithm. It exposes the hidden part of the model.

Problem 3. Given the observation sequence *O*, the number of unique symbols *M* and the number of states *N* are known, find the model $\lambda = (A, B, \pi)$ that maximizes the probability of *O*. This can be done efficiently by adjusting the model parameters using the Baum-Welch Algorithm.

3.5.1 HMM and Virus Detection

Research has already proved that HMM is successfully used for the detection of metamorphic malware **??**. HMM has to be trained using the input data to generate the training models. This trained model, now represents the statistical properties of the malware family. A new binary is scored based on this trained HMM. This score shows the similarity of the new binary file with the malware family represented by the model. File is classified based on the threshold value.

To build a training model, a number of files belonging to the same malware family are disassembled using IDA Pro or some other disassembler. Unique assembly codes are obtained from these disassembled file which form the HMM symbols. These opcode sequences obtained from different malware files belonging to the same family are concatenated to generate a long opcode sequence. This opcode sequence is then used to train the HMM. The set of unique opcodes from this sequence represent the observation symbols in the HMM model **??**. An example of such model is given in the figure.

After the HMM training model is produced, it can be tested using any file to determine whether the file belongs to the same malware family or not. If the score of the binary file is greater than the threshold, than the file belongs to the malware family otherwise it is considered as benign.

3.5.2 Log Likelihood Per Opcode

In order to score the files, product of probabilities needs to be computed. As T increases, the product of probabilities tends to O exponentially. This problem is solved by normalizing the results using forward and backward algorithms **?**.

This is called log likelihood. It is the sum of log state transition probabilities which means its value depends on the length of the sequence. Longer sequence will have higher log probability than the shorter one. However, the sequence in the test set can be of different length than that of the training set. This problem is avoided by dividing the log likelihood by the total number of opcodes in the sequence **??**.

3.5.3 Effectiveness of HMM Detection

HMM is considered very effective for the detection of metamorphic malware based on the research carried out previously **?**. NGVCK is an outstanding metamorphic code generator which produces files with a lot of variation. However, it also failed to bypass HMM in **?**. Its detection rate is almost 90% [12].

3.5.4 Evading HMM Detection

Previous researches have tried to bypass HMM based detection by writing the metamorphic engines **?**. Metamorphic engine is written in such a way that it inserts dead code in the malware file based on the dynamic scoring algorithm. This means the dead code is inserted only if it makes the log likelihood score of malware file closer to the score of benign files.

The research carried out in **?** shows that instead of inserting random blocks of dead code, long dead code sequences or subroutines are better at evading the HMM based detection. HMM based

detector fails when 35% dead code is inserted. However, in case of inserting subroutines it fails only at 30%.

In **?**, metamorphism was applied on the LLVM IR bytecode. In this dead code was inserted in the form of functions in the malware files and these functions were also called within the program. HMM failed to detect the morphed code at relatively low rates. HMM started miss-classifying the files after more than 20% of dead code was inserted. These results form the basis of our research.

3.6 Conclusion

In this chapter, we have discussed the LLVM which is the framework used by us for development of the metamorphic engine. Along with this, we have also discussed the Hidden Markov Model. This is a statistical detection technique which is used to test the effectiveness of our metamorphic techniques. We generate the training Hidden Markov Model using the training files and then test the morphed copies of the Havex malware against this model.

Chapter 4

PROPOSED OBFUSCATION TECHNIQUES AND THEIR IMPLEMENTATION

4.1 Introduction

This chapter discusses in detail the metamorphic techniques which are implemented to strengthen the malware. This chapter describes the techniques used by us in detail to generate each copy by metamorphic engine. Each generated copy is different from the base file but the functionality does not change. Also the percentage of inserted code in the base file tells how much the morphed copy will differ from that of the base file. If the percentage of morphing code in the malware base file is more, the file will look more like a benign file. The opcode counts and sequences of the morphed file determine whether it is a normal file or a malware file. Hidden Markov Model based analysis has proved itself capable of distinguishing the malware files from the benign files. In this project, we have developed a metamorphic engine which is tested on the HMM based detector. Section 2 discusses these metamorphic techniques in detail. It also gives the detailed methodology that we have proposed and developed to create diverse copies of malware.

4.2 Metamorphic Techniques Used

We have implemented metamorphic techniques at IR bytecode level. Previous work on metamorphic techniques has been done on Assembly code level.It is difficult to implement all of the metamorphic techniques mentioned in Chapter 3 on IR level. In this project we have implemented

three of these techniques. This is done by inserting dead code subroutines from the benign files into the malware files. The order of these subroutines is also randomized to add to the variation between the generated copies. We also substitute the instructions with equivalent instructions. The metamorphic techniques implemented in this research are discussed below.

4.2.1 Instruction Substitution

As discussed in Chapter 3, instructions can be written in multiple ways. This means to add diversity to the code, an instruction can be replaced by other instructions without affecting the functionality of the code. In LLVM, a pass can be implemented which replaces an instruction with some other instruction.

4.2.2 Dead Code Insertion

Insertion of dead code into a file can be done by taking instructions from the benign files which when added to the the base file do not affect the functionality of the base file while adding to the diversity of the opcodes.

IR bytecode has design strategy of RISC type instructions. Due to this, adding dead instructions is a hideous task. So instead of adding instructions, we insert complete subroutines of dead code using llvm-link tool. IR bytecode of the dead subroutine is obtained and inserted in the base file IR bytecode.

In this project, core-utils Linux command files **?** have been used as benign files from which dead code subroutines are taken.

4.2.3 Subroutine Permutation

IR bytecode is in the form of text so this technique is implemented by simple reordering of the sequence of the functions. This helps in bypassing any detection based on pattern matching. C++ code has been implemented which takes the subroutines, reorders them and save them in a text file.

4.3 Stages of our metamorphic engine

Our metamorphic engine has four passes through which the base malware file passes to generate its structurally diverse copies. Applying only a single pass also has the ability to generate the metamorphic malware but here we have applied three different metamorphic techniques in these passes. These multiple passes make the malware a lot more complex and different from its base malware. Different stages of our metamorphic engine are discussed and explained in the subsections below. Figure 4.1 shows the high level architecture of our metamorphic engine.

4.3.1 Dead code insertion

This is the first pass in which we specify the base file, the morphing file which is the benign file from which dead code is taken, and the percentage of dead code which has to be inserted. The percentage of dead code to be inserted tells the number of lines which will be added to the malware base file. This percentage is calculated by determining the number of lines of the malware file to find out the percentage of the benign code lines in the morphed file. On the basis of this number, functions are inserted, which have number of lines approximately equal to the line numbers, to the base file. This integration is done using llvm-link tool. This pass is discussed in detail below.

1. Compile the selected benign file using llvm-gcc command or clang command and generate its

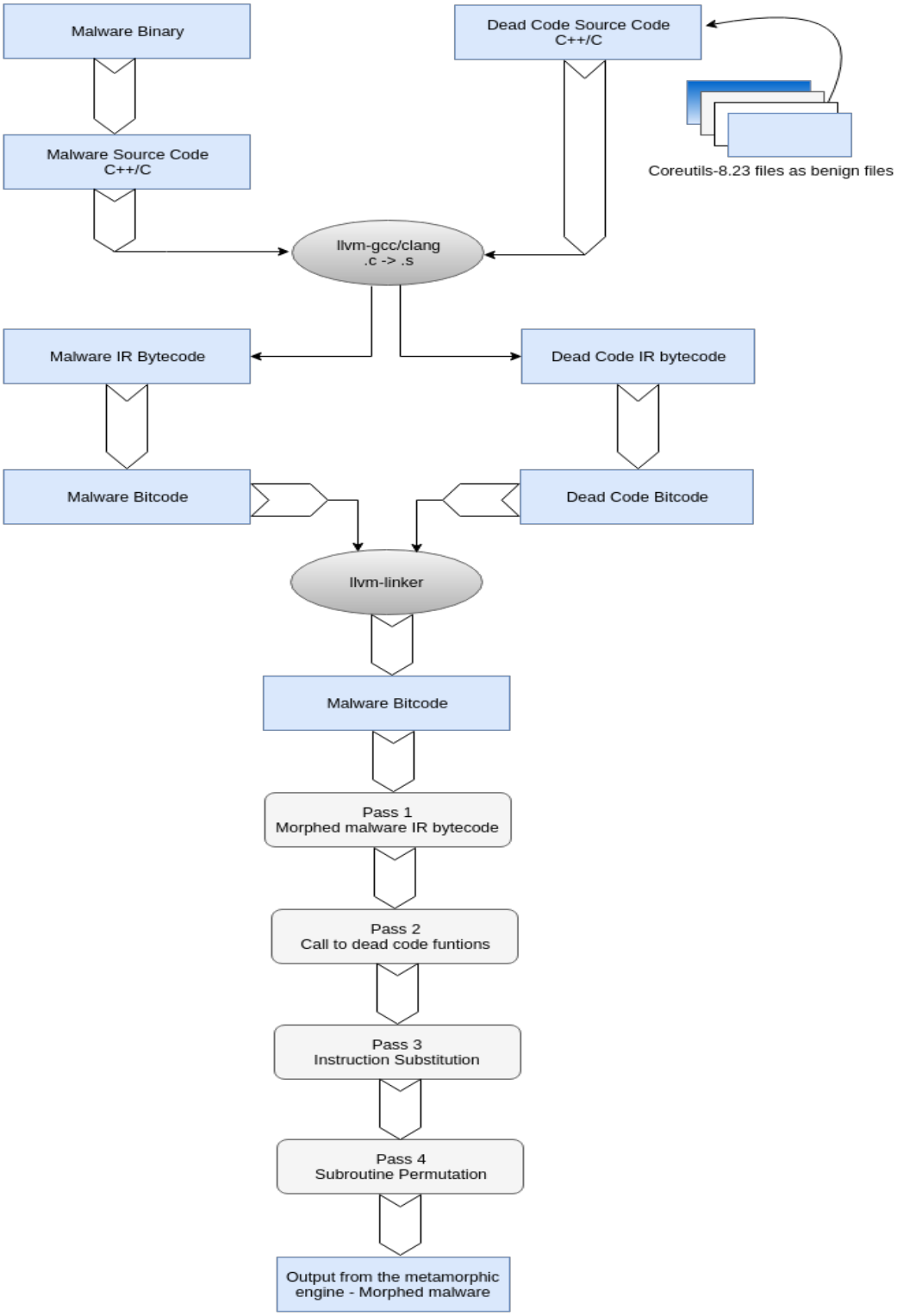


Figure 4.1: Architecture of our metamorphic engine

LLVM IR bytecode in the same directory.

2. Function dependencies are determined based on this bytecode.

3. Number of lines is calculated for every subroutine.

4. The percentage of dead code to be inserted tells the number of lines to be inserted. Greedy strategy is used to find out the best approximate of functions based on their number of lines.

5. If the number of lines of two functions are same, then functions are selected on the basis of random number.

5. The chosen subroutines are copied to a temporary IR bytecode file.

6. Bitcode files are generated for the base malware bytecode and the temporary bytecode using llvm-as tool.

7. These two bitcode files are linked using llvm-link tool.

8. If two names in the base and temporary bitcode file are in conflict with each other, replace the one in the temporary string with a random string. Goto 7.

9. Delete the temporary IR bytecode file.

This process is explained in the form of flow chart in the 4.2.

For every function, it is checked that which sub-functions are called by that function. These sub-functions are also copied as a dead code. Therefore, it is important to find out the lines of the sub-functions too. Figure 4.3 shows the screenshot of the output of showing the number of lines for the individual functions.

The purpose of adding dead code into the malware files is to increase the diversity of the morphed

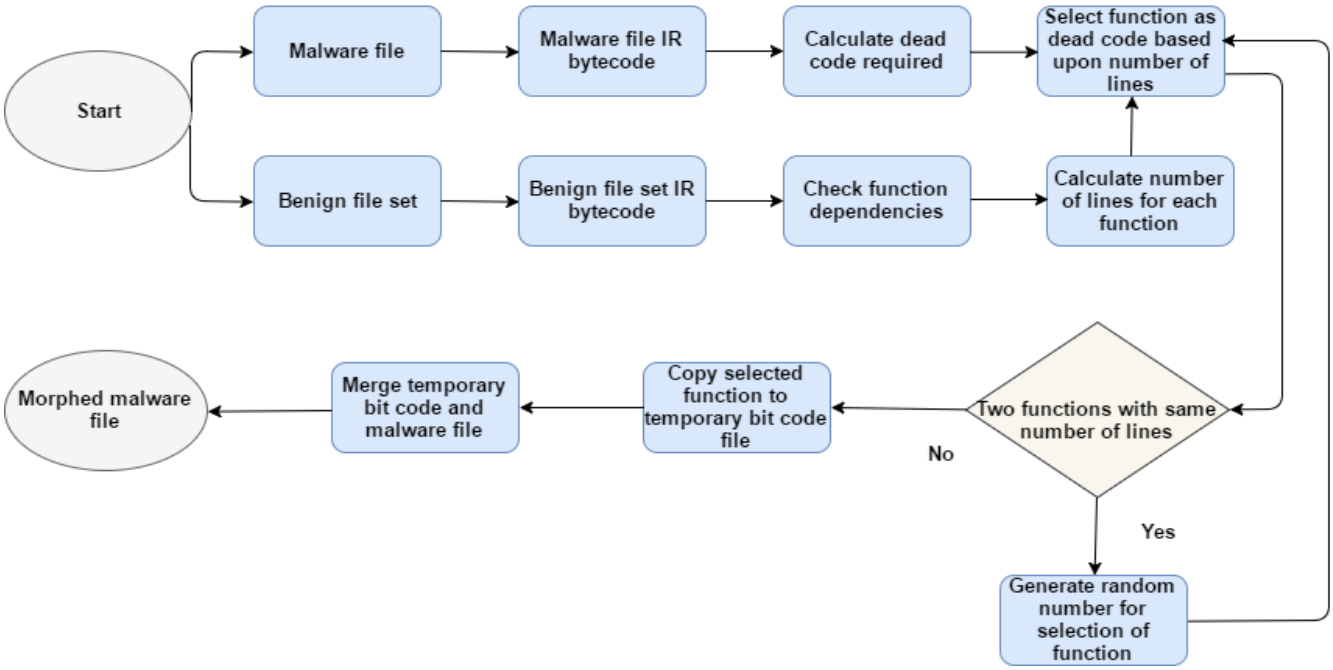


Figure 4.2: A flow chart demonstrating the insertion of dead code from benign files

malware files. As the number of benign files are infinite, the number of malware files that can be generated using dead code from the benign files is also infinite.

4.3.2 Subroutine Permutation

In this pass, functions are rearranged randomly in the IR bytecode file. This does not alter the overall affect of the functions. The algorithm for this pass is discussed in the following steps.

1. Read IR bytecode file and make a temporary file containing all the global variables from the IR bytecode file.

2. Generate random number from 1 to total number of functions using a random number generator.

3. Write the function obtained as a result of generating random number and write the function to the temporary IR bytecode file. If the function already exists in the temporary file, do not copy it.

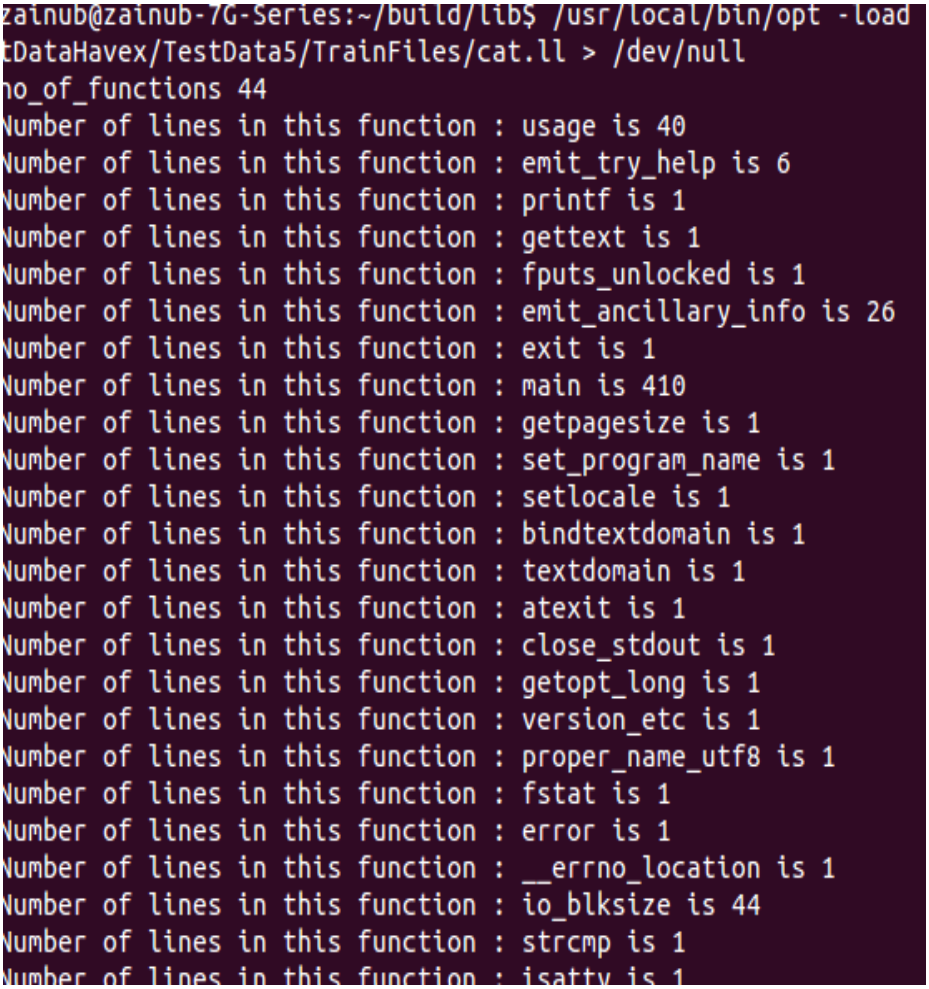


Figure 4.3: Number of lines in individual functions

4. Repeat the above two steps until all the functions are written to the temporary bytecode file.

A flow chart given in the Figure 4.4 explains this pass in a better way.

This pass is implemented on LLVM runOnModule(Module&) function because it has to iterate over all the functions in the module. The number of functions found in the module are calculated. If number of functions in the module is n, random number is generated n times. This random number is associated to each function in the module. If a random number is 23, the function on 23th number has to be written to the temporary file. The new bytecode file will have all the functions as in the input module but with a different order.

4.3.3 Instruction Substitution

This pass is designed in such a way that it reads all the instructions in the file and checks whether the instruction can be replaced by other equivalent instructions mentioned in Section . If any such instruction is found, it is replaced by an equivalent instruction randomly. This pass operates on Function class. This pass is implemented based on the algorithm given in the following steps. Figure 4.5 explains this process in the form of a flow chart.

1. The pass iterates over all the functions.

2. Read every instruction in the function.

3. If the instruction is in the list of instructions (AND, OR, XOR, Addition, Subtraction) that can be substituted, replace it with its equivalent instruction.

4. For an instruction with multiple possible substitutes, a random number is generated. Instruction is replaced by the other instruction based on that random number.

5. The amount of instruction replacement is done based on the random number.

This pass is designed to run on the LLVM module so that it goes through all the functions that are there in the module. It checks every instruction if it has binary operator in it or not. If so, the operator is matched to be in the list of operators that are supported by our pass. For the research purpose, we have implemented this pass to work on five different types of operators including, AND, OR, XOR, Addition and Subtraction.

The instructions are replaced by other equivalent instructions which perform the same functionality. The list of possible substitutes for each type of instruction is given in the Figure 4.6. Usually instruction substitution does not add much to the security but seeding the instruction with

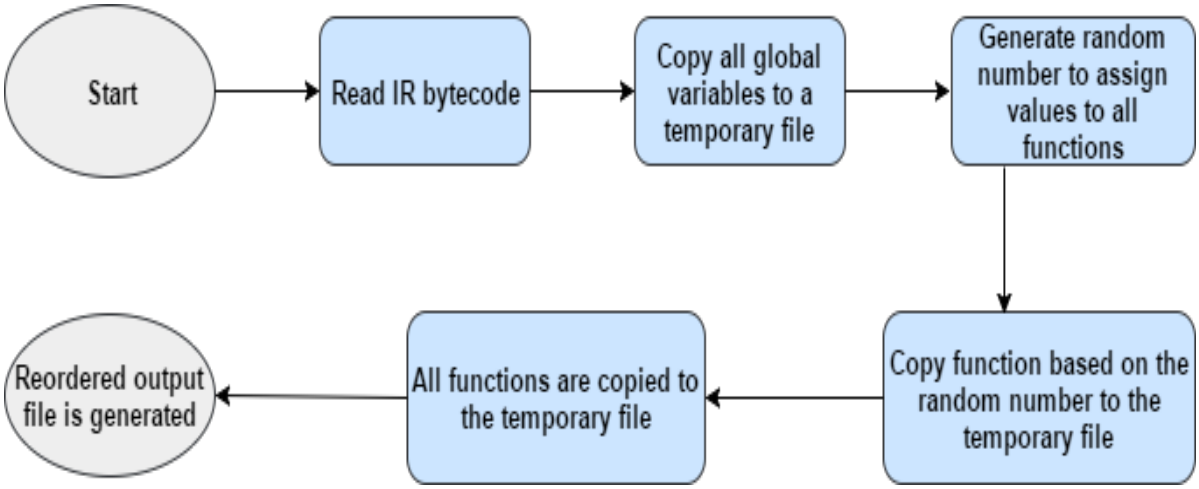


Figure 4.4: A flow chart demonstrating the function permutation

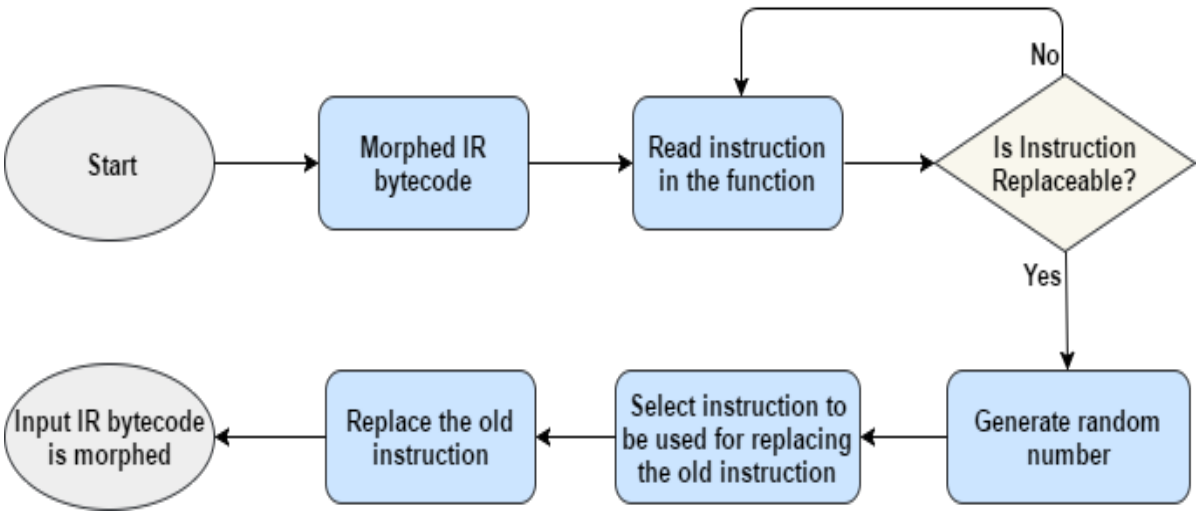


Figure 4.5: A flow chart explaining the replacement of instructions

pseudo-random number generating every time can be very effective.

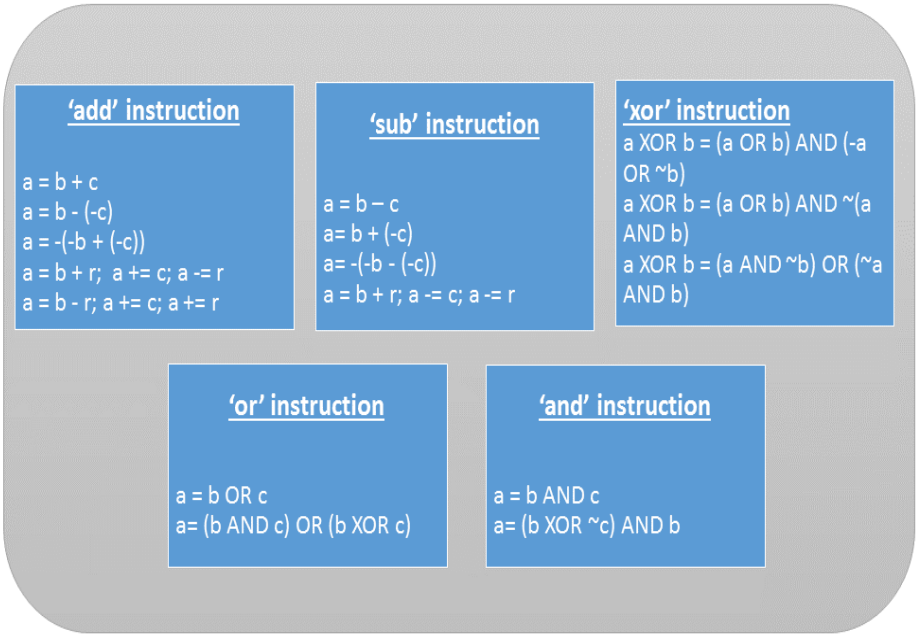


Figure 4.6: List of equivalent instructions

In LLVM, there is an instruction to replace one instruction with another. The instruction is ReplaceInstWithInst(oldInst,newInst) in which old instruction is the instruction to be replaced and newInst is the instruction which replaces the old instruction. The input module is then updated to make the modifications made to the file.

4.4 Conclusion

In this chapter, we have discussed the metamorphic techniques implemented by us to make the malware more robust against the defense mechanisms. Three techniques have been implemented by us including dead code insertion, instruction substitution and function permutation. These are explained with the detailed description of our algorithms along with the figures and flow charts.

Chapter 5

EXPERIMENTS AND RESULTS

5.1 Introduction

This chapter discusses in detail the experiments which are performed by us on the metamorphic engine implemented by us. We discuss our base malware Havex and why we have chosen it for research purpose in our thesis. This chapter discusses the details of applying the metamorphic techniques on the Havex malware. Furthermore, HMM technique is used for the evaluating the performance of the engine. The usage of this technique with reference to the Havex malware is described in this chapter. Section 1 discusses the Havex malware which is our base malware and the reason we have chosen this malware. Section 2 discusses the benign files that we have used for dead code insertion in our metamorphic engine. Section 3 explains the training of HMM and classification of files. It also gives the details of our results along with the graphs.

5.2 Havex: Base malware file

Energetic Bear is involved in making several Advanced Persistent Threats (APTs) on the energy sector malware. Havex is one of the tools used by this group to launch attack operations. Its main target is to attack the ICS and SCADA systems. Havex is exclusively used for cyber espionage. It has also attacked multiple other sectors as industrial/machinery, manufacturing, pharmaceutical, construction, education and Information Technology. Havex infiltrates the most critical environments through spear phishing, watering hole and legitimate ICS downloads injected with

malware[1]. It is a trojan which communicates with the Command and Control server and sends required data from the target to the server. Havex is implemented in C language.

For research purpose, we downloaded a sample of Havex binary from ?. The next step is to disassemble the malware binary to get the source code of the malware. We used Retargetable Decompiler ? to recover the C source code of the Havex as shown in Figure 5.1.

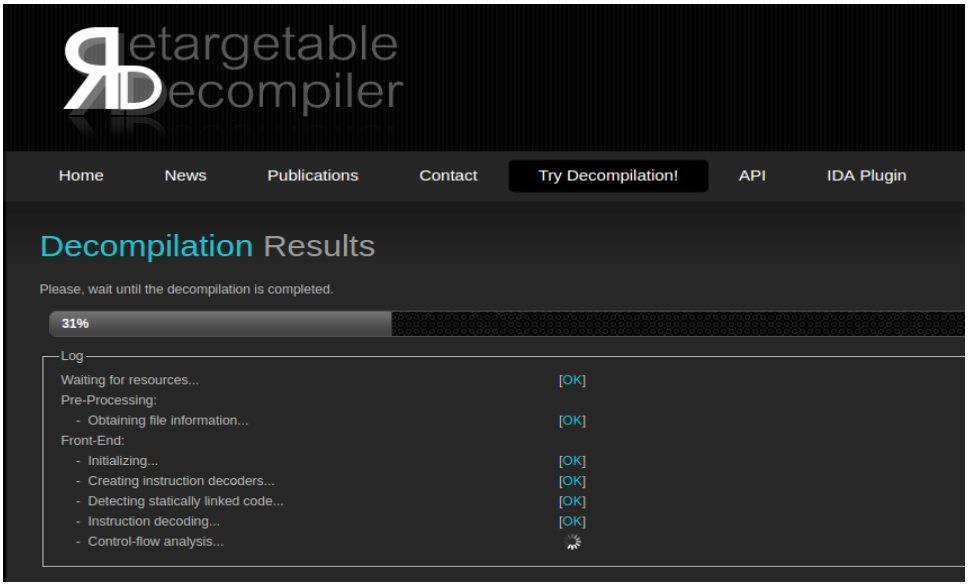


Figure 5.1: Decompiling Havex using Retargetable Decompiler

5.3 Benign files

Benign files from chosen from Linux coreutil-8.23 ? command files as the morphing files. The reason for choosing these files is, these files also perform the same low level operations as our base file ?. These files are also written in C language. We chose 50 coreutil command files to use them to morph our base malware. For producing training data set and testing data set, a percentage of dead code from these benign files were added to the base malware file. This generated 50 unique samples for each type of data set. A list of files used for the creating training and testing data set is given in the Figure 5.2.

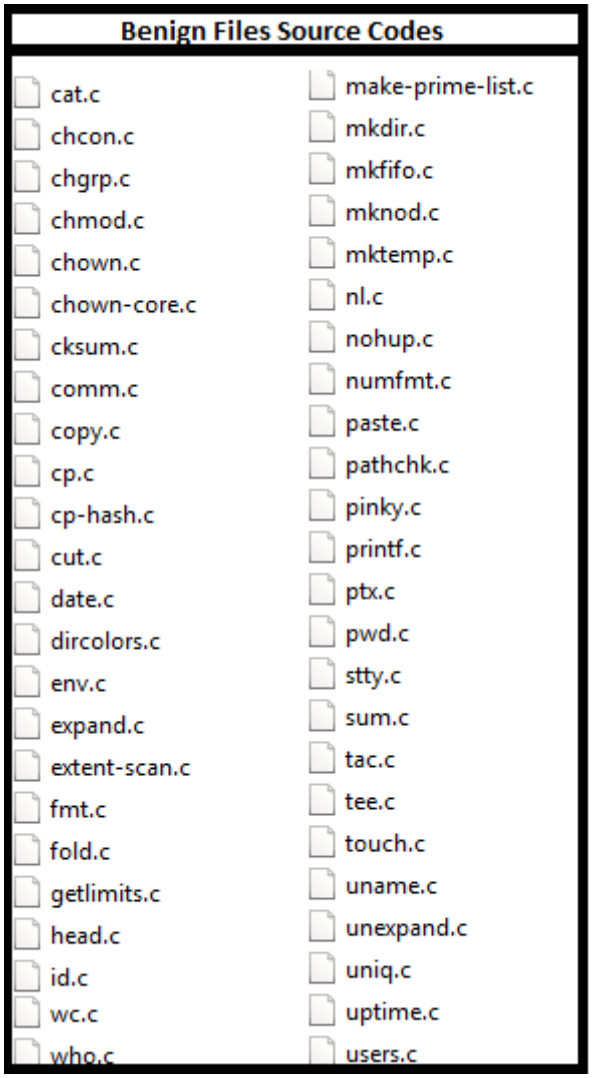


Figure 5.2: A list of morphing files used

5.4 Training and Classification Process

As it has been mentioned in the above section that code from the benign files is added to the normal files. This is a dead code insertion phase. After this, calls are made to these dead functions so that the optimizers perceive that the dead code is being used. A percentage is defined considering the lines of the malware file. For example, if 5% dead code insertion is required in the training files, the number of lines in the Havex file will be calculated such that the percentage of dead code in the morphed Havex file is 5% and that of Havex is 95%. This is shown in Figure 5.3.

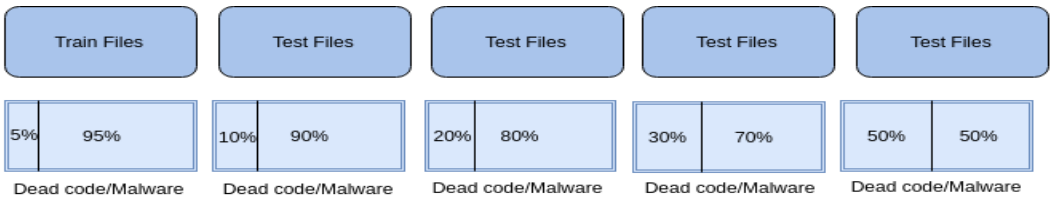


Figure 5.3: Percentage of dead code and malware in the training and testing files

As it can be seen in the Figure , there are five types of data sets. One data set is for training and four data sets are for testing of the metamorphic engine. There are 50 samples in each type of data set. We train our Hidden Markov Model (HMM) based on the 50 samples in the training set. For training of the model, we add 5% dead code in the malware file from 50 benign files. 5% dead code is added in the base file to avoid over-fitting of the data. After this, the morphed malware file is passed through the instruction substitution and function permutation passes respectively. Figure 5.4 shows the generation of training and testing samples for the Hidden Markov Model.

At the end, we obtain 50 different morphed samples of the base malware. These 50 files are used for the training of the model.

Testing is carried out on 4 different types of data sets with 50 samples each. We have added 10%, 20%, 30% and 50% dead code in the malware file copies to generate the morphed copies of the malware. With the increasing percentage of dead code from the benign files, the morphed malware starts resembling the benign files. The effects of the percentages of dead code will be discussed in the section.

5.4.1 Translating opcodes to numbers

As discussed in Chapter 6, when sample files are generated, they are in LLVM IR bytecode format. For the training and analysis of Hidden Markov Model, opcodes of the samples are needed. Therefore, LLVM IR bytecode files are converted to assembly format and opcodes are extracted

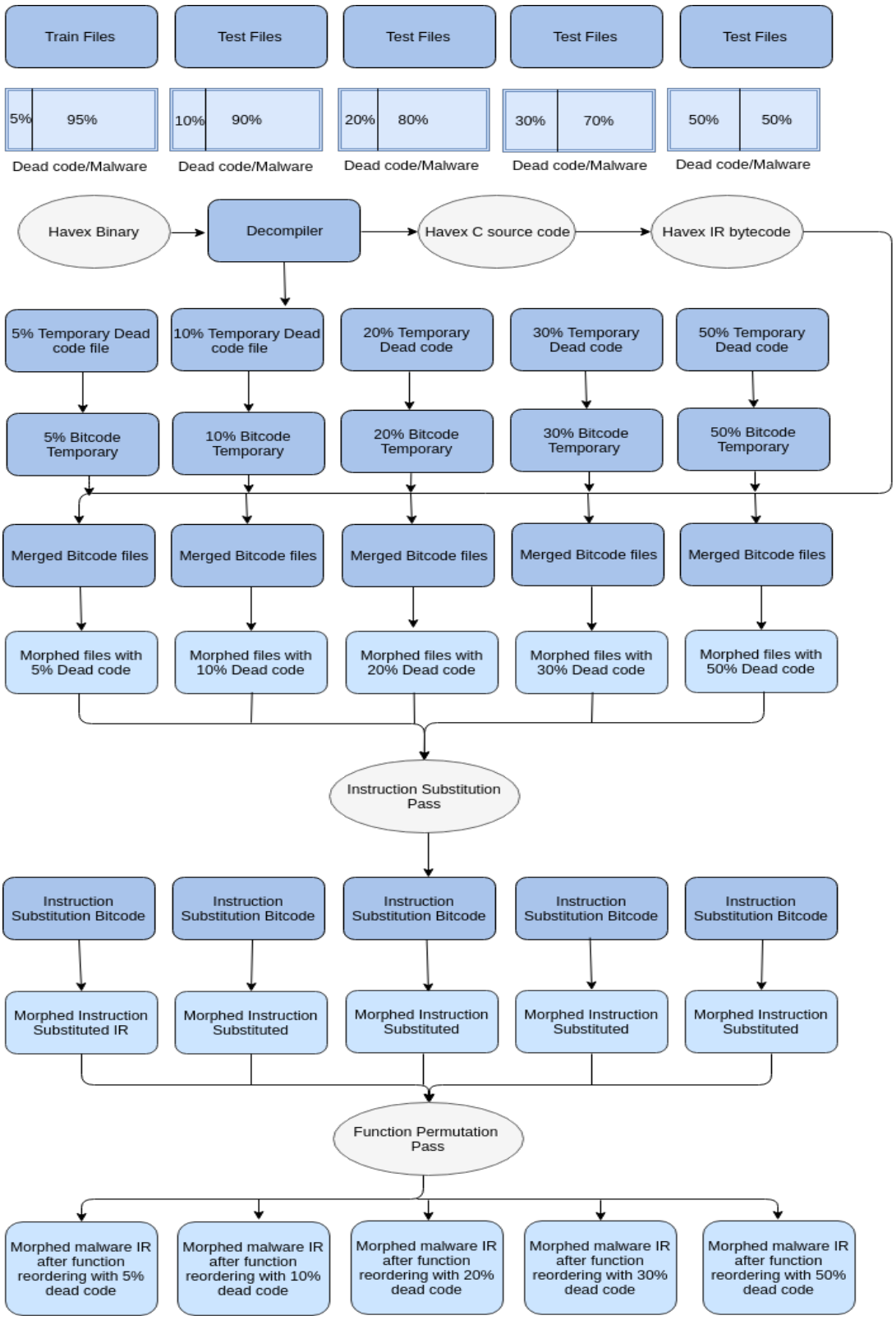


Figure 5.4: Process of generation training and testing samples

from each assembly file. These opcode sequences for the individual files are concatenated together and unique opcodes are saved in a file. Each unique opcode is assigned an index number. Figure 5.5 shows unique opcodes corresponding to their index numbers.

Opcode	Index	Opcode	Index	Opcode	Index	Opcode	Index	Opcode	Index	Opcode	Index
pushq	1	testb	26	fstpt	51	movsd	76	mulq	101	xorq	126
movq	2	movabsc	27	fldt	52	cvtsi2sdl	77	orl	102	testw	127
subq	3	incq	28	decl	53	fldl	78	andb	103	xorb	128
movl	4	cmpb	29	jbe	54	fdivl	79	notb	104	sarq	129
testl	5	movzbl	30	subl	55	fdivs	80	cmovneq	105	incb	130
je	6	movsbl	31	setns	56	jp	81	andq	106	notq	131
callq	7	movzwl	32	fildll	57	fnstcw	82	movups	107	subb	132
xorl	8	addl	33	fadds	58	movw	83	cmovel	108	decb	133
addq	9	ja	34	fmulp	59	fldcw	84	cmovnel	109	shrb	134
popq	10	movsbq	35	fdivrp	60	fistpll	85	shll	110	imull	135
retq	11	jb	36	fstpt	61	imulq	86	rep;movsq	111	btl	136
testq	12	js	37	fildl	62	shrq	87	adcl	112	shlb	137
movb	13	setne	38	fld	63	negl	88	seta	113	addb	138
leaq	14	divq	39	flds	64	cld	89	negb	114	idivq	139
jmp	15	adcq	40	fdivp	65	rep	90	orb	115	movswq	140
decq	16	jns	41	fsubp	66	sarl	91	fstpl	116	setbe	141
cmpl	17	andl	42	faddp	67	shrl	92	fstps	117	sbbl	142
jg	18	jle	43	fld1	68	orq	93	movss	118	cvtsi2sd	143
leal	19	jmpq	44	fmuls	69	btq	94	movswl	119	movapd	144
jae	20	negq	45	fiaddl	70	cmpw	95	setb	120	divsd	145
cmpq	21	sete	46	fchs	71	divl	96	setl	121	xorpd	146
jne	22	incl	47	fxch	72	movaps	97	cwtl	122	movupd	147
jge	23	jl	48	fucompi	73	cltd	98	xorps	123	mulsd	148
movslq	24	setg	49	cltq	74	idivl	99	ucomisd	124	jnp	149
shlq	25	fldz	50	cvtsi2sdl	75	notl	100	setp	125	cmoveq	150

Figure 5.5: Unique opcode mapping to index

An example of opcode to index number translation is given in the Figure 5.6.

5.4.2 Modeling the 'base' HMM

Once the opcodes are converted to numbers, HMM can be trained using a scoring technique similar to that mentioned in **?**. According to the previous researches **???**, the number of hidden states does not make any difference into the HMM malware classification. In this research, the number of hidden states used are N=10.

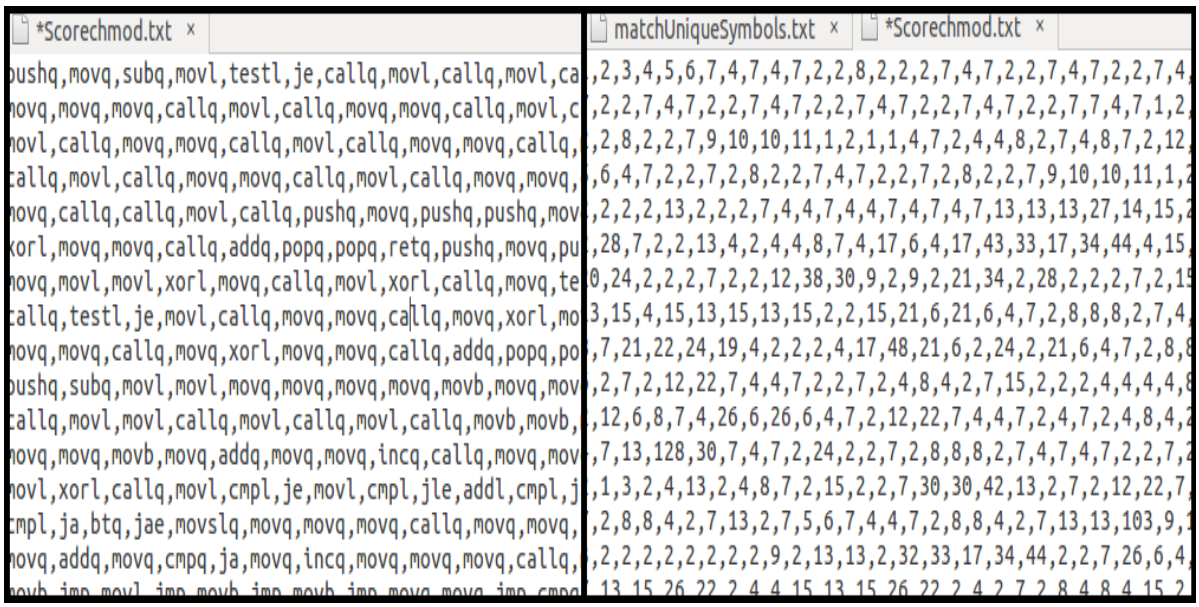


Figure 5.6: Opcodes converted to numbers for benign file 'chmod.c'

As mentioned in the section, we use 5% rate of morphing for the training of the HMM. 50 samples of such files are used and the numbers translated from the opcodes are used for the training. A model generated from training on the 50 random morphed files referred to as 'base HMM' by us. This model is used to generate scores for all the other test files.

5.5 HMM based scoring and detection

After training the HMM, next task is to score the files. First of all, the morphing files will be scored against the model. These morphing files are the set of 50 coreutil-8.23 Linux command files. Scores for all the test file samples will be discussed in reference to the score of the morphing files. The graph given in the Figure 5.7 shows the scores of the morphing files.

These scores are compared to the scores of the base files with 5% rate of morphing. If the score of the morphing file is greater than the threshold score, the morphing file is considered as benign. Otherwise, the morphing file is considered as part of the base Havex family. If scores of the morphing files are in the score zone of the base malware files, the threshold of the HMM is not

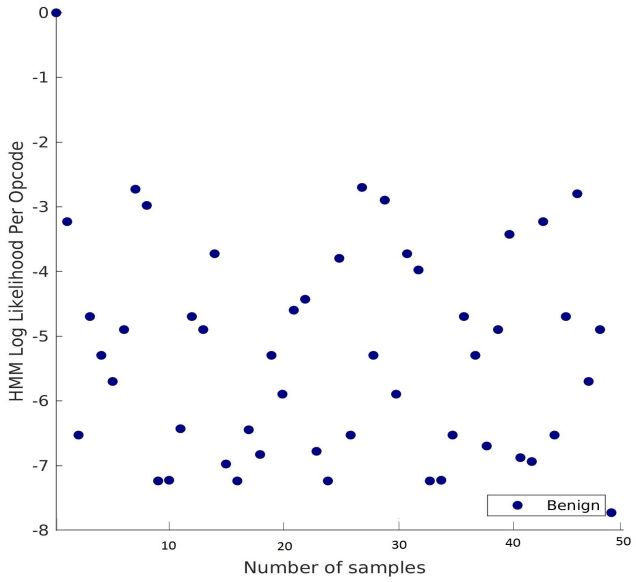


Figure 5.7: HMM LLPO Scores of the benign coreutil Linux command files

considered good enough as it is not classifying the files correctly. The scores of 50 base files against the 50 morphing files are shown in the graph given in the Figure 5.8. The scores of all the morphing files are lesser than the scores of the morphed files. Hence, the detection percentage at 5% rate of morphing is 100%.

5.5.1 10% and 20% rate of morphing

After 5% rate of morphing we increase the percentages to 10% and 20% to see the behavior of HMM in detecting the morphed Havex malware. For both the percentages, 50 sample test files are generated. These scores are then plotted against the base files and the scores are compared. When the rate of morphing in the files is increased, the size of the files is also increased because the number of lines added in the file increases. To prevent the sizes of the files from affecting the HMM scores, number of opcodes per file are used to normalize the log likelihood score. This score is log likelihood per opcode. An instance of this is given in the Figure 5.9.

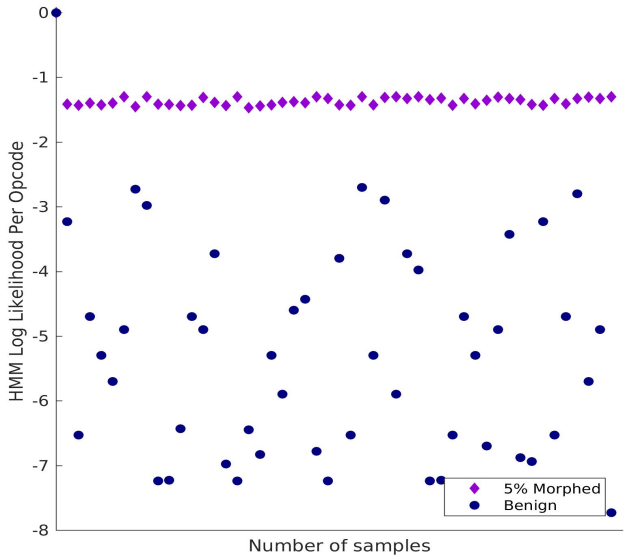


Figure 5.8: HMM LLPO Scores with 5% dead code

```
-   testSeq=importdata("/home/zainub/hmm/TrainTest/TesttHavex/asm50_copy.txt");
-   [a,b]=size(testSeq);
-   prior_hat;
-   A_hat;
-   B_hat;

-   LogLikelihood = dhmm_logprob(testSeq, prior_hat, A_hat, B_hat)
-   fprintf('Opcode = %d\n',b);
-   disp('LLPO= LogLikelihood/Opcode')
-   disp(LogLikelihood/b)

-   save ('/home/zainub/hmm/TrainTest/FinalTest/asm50_copyTestScore', 'LogLikelihood');
```

```
ommand Window

Trial>> test
Running test for asm50_copy.txt

LogLikelihood =

  -1.1539e+05

Opcode = 73374
LLPO= LogLikelihood/Opcode
   -1.5727
```

Figure 5.9: Calculating log likelihood per opcode for a test file

Log likelihood Per Opcodes gives us the idea whether the file is a Havex file or a benign file. LLPO of 10% and 20% morphed files are plotted against the benign files respectively in the Figure 5.10 and Figure 5.11.

The scores of the malware files are greater than the scores of the benign files. This means that although the Havex LLPO score with 10% rate of morphing has decreased to some extent but the morphed files are still detectable by the HMM. Our next move is to increase the percentage to 20% to see the effect. As shown in Figure , the scores have gotten better for 20% rate of dead code insertion and some scores of Havex morphed copies have started merging with those of the benign files. It shows that at 20%, HMM has started giving the false results. Our idea of making HMM fail is showing positive results at this percentage.

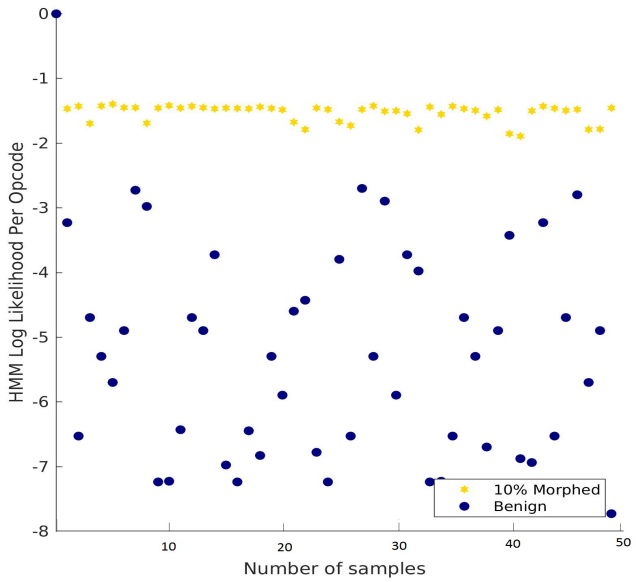


Figure 5.10: HMM LLPO Scores with 10% dead code

5.5.2 30% and 50% rate of morphing

To further understand the strength of our metamorphic engine against the HMM based detector, we increase the percentage of morphing to 30% and 50% with 50 samples each. LLPO of these

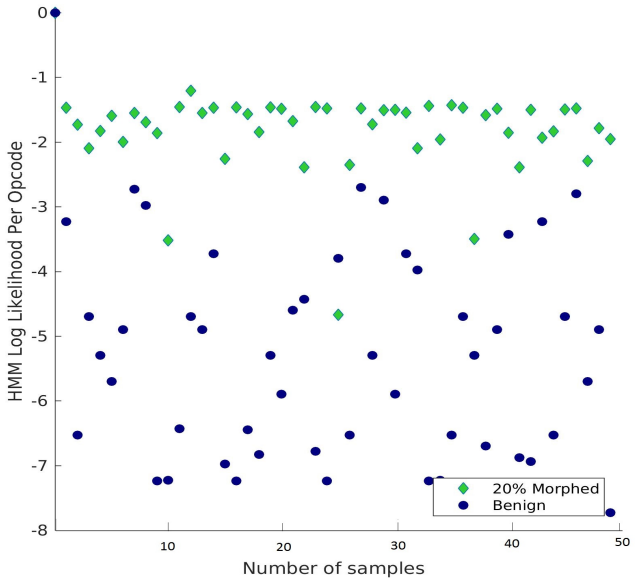


Figure 5.11: HMM LLPO Scores with 20% dead code

files are obtained and plotted against those of morphing files as shown in Figure. With 30% morphing, the LLPO scores for the morphed files has gotten too low, many morphed Havex files have started resembling the benign files. This is exactly what we wanted to achieve with the obfuscation of Havex. With 50% morphing, the scores of the morphed Havex files have gotten low to the extent that the HMM detector is unable to distinguish between the malware and benign files. The result of our experiments on 30% and 50% morphed files is shown in the graphs given in the Figure 5.12 and Figure 5.13 respectively.

These HMM Results are given in the form of tables in Figure 5.14 and Figure 5.15.

To understand the effect of percentage of morphing on the Havex malware on the HMM LLPO scores, the results of LLPO for each percentage is shown in the Figure 5.16.

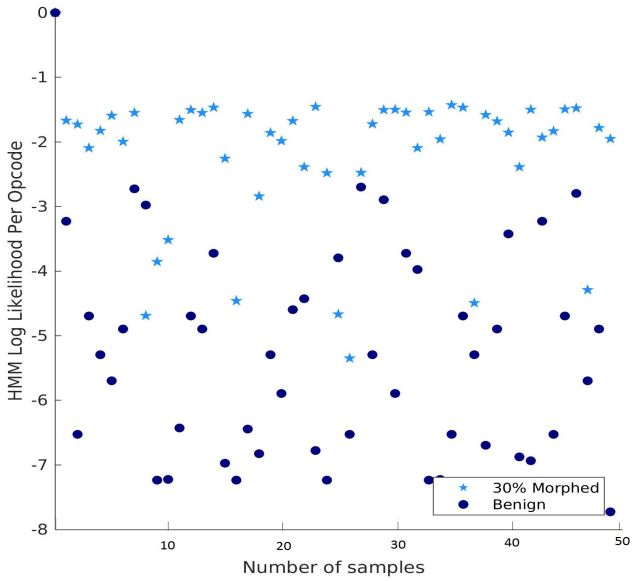


Figure 5.12: HMM LLPO Scores with 30% dead code

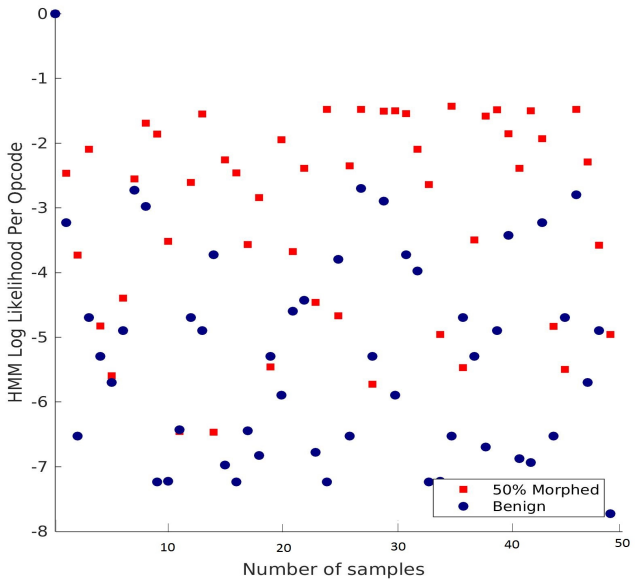


Figure 5.13: HMM LLPO Scores with 50% dead code

LLPO scores for 5% dead code morphed files	-1.417 -1.417 -1.4301 -1.399 -1.4286 -1.3977 -1.3003 -1.4541 -1.3003 -1.416 -1.4201 -1.43887 -1.43 -1.31 -1.39 -1.43606 -1.3003 -1.47 -1.4401 -1.42632 -1.386 -1.377 -1.393 -1.301 -1.33 -1.427 -1.4301 -1.299 -1.4286 -1.30977 -1.3003 -1.32977 -1.3003 -1.343 -1.3203 -1.434 -1.3272 -1.4097 -1.35282 -1.304 -1.3287 -1.343 -1.4203 -1.434 -1.3272 -1.4097 -1.3282 -1.304 -1.3287 -1.303	Min Score= -1.4541 0 morphed files with score < -2.7
LLPO scores for 10% dead code morphed files	-1.47 -1.47 -1.4301 -1.699 -1.4286 -1.3977 -1.453 -1.4541 -1.693 -1.46 -1.4201 -1.45887 -1.43 -1.451 -1.469 -1.4606 -1.463 -1.47 -1.4401 -1.4632 -1.486 -1.677 -1.793 -1.461 -1.483 -1.67 -1.7301 -1.479 -1.4286 -1.5077 -1.503 -1.5477 -1.798 -1.443 -1.559 -1.434 -1.472 -1.497 -1.582 -1.484 -1.857 -1.893 -1.503 -1.434 -1.4632 -1.497 -1.482 -1.794 -1.787 -1.4573	Min Score= -1.893 0 morphed files with score < -2.7
LLPO scores for 20% dead code morphed files	-1.5477 -1.47 -1.7301 -2.099 -1.8286 -1.5977 -1.999 -1.5541 -1.693 -1.86 -3.5201 -1.45887 -1.209 -1.551 -1.469 -2.2606 -1.463 -1.57 -1.84401 -1.4632 -1.486 -1.677 -2.393 -1.461 -1.483 -4.67 -2.35301 -1.479 -1.7286 -1.5077 -1.503 -1.5477 -2.098 -1.443 -1.959 -1.434 -1.472 -3.497 -1.582 -1.484 -1.857 -2.393 -1.503 -1.9334 -1.83632 -1.497 -1.482 -2.294 -1.787 -1.9573	Min Score= -4.67 3 morphed files with score < -2.7
LLPO scores for benign files	-3.23 -3.23 -6.53 -4.7 -5.3 -5.7 -4.9 -2.73 -2.98 -7.24 -7.23 -6.43 -4.7 -4.9 -3.73 -6.98 -7.24 -6.45 -6.83 -5.3 -5.9 -4.6 -4.43 -6.78 -7.24 -3.8 -6.53 -2.7 -5.3 -2.9 -5.9 -3.73 -3.98 -7.24 -7.23 -6.53 -4.7 -5.3 -6.7 -4.9 -3.43 -6.88 -6.94 -3.23 -6.53 -4.7 -2.8 -5.7 -4.9 -7.73	Max Score= -2.7 0 benign files with score > -1.4541 0 benign files with score > -1.893 15 benign files with score > -4.67

Figure 5.14: HMM Results for morphed files with 5%, 10% and 20% dead code against the HMM Threshold

LLPO scores for 30% dead code morphed files	-1.68 -1.67 -1.7301 -2.099 -1.8286 -1.5977 -1.999 -1.5541 -4.693 -3.86 -3.5201 -1.65887 -1.509 -1.551 -1.469 -2.2606 -4.463 -1.57 -2.84401 -1.8632 -1.986 -1.677 -2.393 -1.461 -2.483 -4.67 -5.35301 -2.479 -1.7286 -1.5077 -1.503 -1.5477 -2.098 -1.543 -1.959 -1.434 -1.472 -4.497 -1.582 -1.684 -1.857 -2.393 -1.503 -1.9334 -1.83632 -1.497 -1.482 -4.294 -1.787 -1.9573	Min Score= -4.693 8 morphed files with score < -2.7
LLPO scores for 50% dead code morphed files	-2.55 -2.47 -3.7301 -2.099 -4.8286 -5.5977 -4.399 -2.5541 -1.693 -1.86 -3.5201 -6.45887 -2.609 -1.551 -6.469 -2.2606 -2.463 -3.57 -2.84401 -5.4632 -1.9486 -3.677 -2.393 -4.461 -1.483 -4.67 -2.35301 -1.479 -5.7286 -1.5077 -1.503 -1.5477 -2.098 -2.6443 -4.959 -1.434 -5.472 -3.497 -1.582 -1.484 -1.857 -2.393 -1.503 -1.9334 -4.83632 -5.497 -1.482 -2.294 -3.5787 -4.9573	Min Score= -1.893 0 morphed files with score < -2.7
LLPO scores for benign files	-3.23 -3.23 -6.53 -4.7 -5.3 -5.7 -4.9 -2.73 -2.98 -7.24 -7.23 -6.43 -4.7 -4.9 -3.73 -6.98 -7.24 -6.45 -6.83 -5.3 -5.9 -4.6 -4.43 -6.78 -7.24 -3.8 -6.53 -2.7 -5.3 -2.9 -5.9 -3.73 -3.98 -7.24 -7.23 -6.53 -4.7 -5.3 -6.7 -4.9 -3.43 -6.88 -6.94 -3.23 -6.53 -4.7 -2.8 -5.7 -4.9 -7.73	Max Score= -2.7 0 benign files with score > -1.4541 0 benign files with score > -1.893 15 benign files with score > -4.67

Figure 5.15: HMM Results for morphed files with 30% and 50% dead code against the HMM Threshold

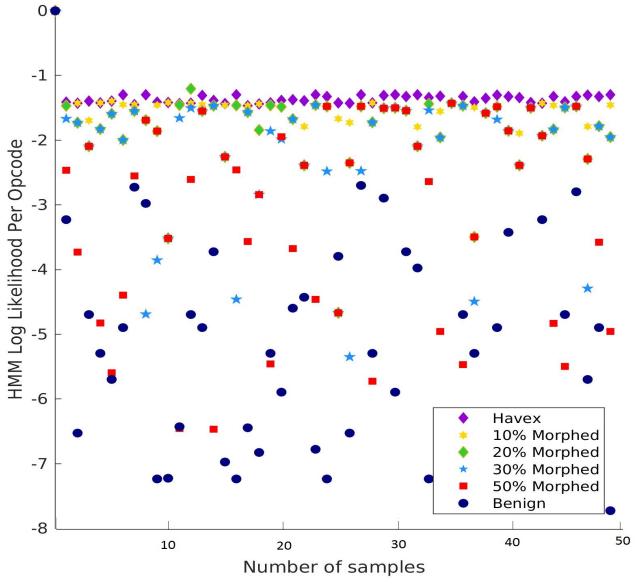


Figure 5.16: HMM LLPO scores for various percentages of dead code against the benign files

5.6 Conclusion

In this chapter, we have discussed the experiments performed by us and results obtained from these experiments are shown in the graphs. The results show that HMM based detector is effective till 20% rate of morphing. Above this, the Log Likelihood score of the morphed files gets closer to those of the benign files and HMM starts miss-classifying the files. The reason of adding dead code from normal files is that frequency of certain opcodes is more in a normal file. Such code snippets when added to the Havex file, increase the percentage of 'benign specific' opcodes in the malware. For instance, 'callq' appears more frequently in the benign files than the Havex file. Also, when the calls are made to the dead code functions, that raises the frequency of 'callq' in the Havex malware. Similarly, 'je' and 'jmp' do not appear as frequently in the Havex malware as in the benign files. Thus, the addition of such opcodes in the malware file makes it appear more like a benign file.

Similarly, 'Havex' files have more percentage of 'xor' opcodes, it can be replaced by 'add' and

'or' instructions using instruction substitution. Function permutation also affects the score of of Havex file as it prevents from pattern matching of the malware.

Chapter 6

CONCLUSION

In this document, we discussed the metamorphic techniques used by the malware and how we have implemented these techniques in our metamorphic engine. We have improvised the techniques used by previous metamorphic engines and used Havex malware as a proof of concept.

In our metamorphic engine, three different metamorphic techniques are used including dead code insertion, instruction substitution and function permutation. In dead code insertion, dead code subroutines are copied from the benign files to the malware files so that the malware files start looking like normal files. To make detection of this code difficult, calls are made to these subroutines. We applied instruction substitution for different operations and function permutation at the end to generate more diverse results. For the proof of concept, we used the Havex malware and tested the morphed Havex files against the HMM based detection. The results showed us that our metamorphic engine is highly effective and makes HMM fail even with a little morphing.

Experiments performed by us on the Havex malware show that as the dead code percentage increases in the Havex malware files after the insertion of dead code, morphed files start resembling the benign files. The frequency of certain opcodes is more in the benign files than in the malware file. Therefore, the Log Likelihood Per Opcode for the morphed Havex gets closer to that of the benign files. The results produced by performing the experiments with a combination of different percentages of dead code, instruction substitution and function permutation technique show that with the rate of morphing above 20%, the HMM based detector is unable to detect the files effec-

tively. Around 30% rate of morphing, a fairly large amount of files go undetected by the HMM based detector. In this project, we are dealing with five different sets of arithmetic or logical instructions for substitution. Further work can be carried out be on more complex instructions. This can make it even stronger. Currently, this technique is only applied on integers. It can be studied how to extend it to the floating data type also.

Since Havex has almost a million lines of code, it was not time efficient to perform obfuscation on it. Further research can be carried out on how to make the process faster for huge files.

More strong metamorphic techniques like register swap and code transposition can also be included along with the instruction substitution to further strengthen the malware.

APPENDIX

.1 LLVM tools with examples

1. llvm-gcc (Command for converting .c file to .s file) is a tool used to convert source code of a file to the IR bytecode after checking for the errors of syntax in the source code file. IR bytecode files are saved with the .s or .ll extension. An example of usage of this command is given below:

llvm-gcc -emit-llvm -S hellofile.c

This generates hellofile.s in the same file containing hellofile.c.

Clang also works like llvm-gcc. It is used specifically for C/C++ languages. Clang command for llvm-gcc is given below.

clang -S -emit-llvm hellofile.c

2. llvm-as (Command for converting .s file to .bc file) takes assembly file as an input, reads it and converts it into bitcode and then writes the bitcode into standard output or file which is a target specific executable.

llvm-as -f hellofile.s

3. lli tool is used for executing the bitcode file using the interpreter.

lli hellofile.s.bc

This executes the hellofile program.

4. llvm-dis (Command for converting .bc file to .s file) is used for the conversion of LLVM bitcode file to assembly or LLVM IR bytecode file which is human readable.

llvm-dis -f hellotest.bc -o opttest.s

This command can also be used for checking how the code optimization was done.

5. llc is used to generate native assembly code based on the LLVM input source file.

llc -f opttest.bc

Native code assembly file opttest.s is generated as an output of this command.

6. llvm-link takes multiple bitcode files as input and links them together to generate a single output bitcode file. For example,

llvm-link -o finalout.bc hellotest.s.bc hellofile.s.bc

BIBLIOGRAPHY

S. M. Sridhara and M. Stamp, "Metamorphic worm that carries its own morphing engine," *Journal of Computer Virology and Hacking Techniques*, vol. 9, no. 2, pp. 49–58, 2013.

W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, 2006.

D. Lin and M. Stamp, "Hunting for undetectable metamorphic viruses," *Journal in computer virology*, vol. 7, no. 3, pp. 201–214, 2011.

T. Waits, "Virus construction kits," in *Virus Bulletin Conference*, 1993, pp. 111–118.

A. Shah, "Metamorphic code generator based on bytecode of llvm ir," 2015.

T. Tamboli, T. H. Austin, and M. Stamp, "Metamorphic code generation from llvm bytecode," *Journal of Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 177–187, 2014.

M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, p. 6, 2012.

J. Aycock, *Computer viruses and malware*. Springer Science & Business Media, 2006, vol. 22.

I. You and K. Yim, "Malware obfuscation techniques: A brief survey," in *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. IEEE, 2010, pp. 297–300.

A. Balakrishnan and C. Schulze, "Code obfuscation literature survey," *CS701 Construction of compilers*, vol. 19, 2005.

P. Beaucamps, "Advanced polymorphic techniques," *International Journal of Computer Science*, vol. 2, no. 3, pp. 194–205, 2007.

P. Szor, *The art of computer virus research and defense*. Pearson Education, 2005.

P. Desai, "Towards an undetectable computer virus," Ph.D. dissertation, San Jose State University, 2008.

B. B. Rad, M. Masrom, and S. Ibrahim, "Evolution of computer virus concealment and anti-virus techniques: a short survey," *arXiv preprint arXiv:1104.1070*, 2011.

X. Li, P. K. Loh, and F. Tan, "Mechanisms of polymorphic and metamorphic viruses," in *Intelligence and Security Informatics Conference (EISIC), 2011 European*. IEEE, 2011, pp. 149–154.

A. Govindaraju, "Exhaustive statistical analysis for detection of metamorphic malware," 2010.

W. Wong, "Analysis and detection of metamorphic computer viruses," Ph.D. dissertation, San Jose State University, 2006.

N. Idika and A. P. Mathur, "A survey of malware detection techniques," *Purdue University*, vol. 48, 2007.

P. Szor and P. Ferrie, "Hunting for metamorphic, symantec security response."

E. Filiol, "Metamorphism, formal grammars and undecidable code mutation," *Self*, vol. 50, p. 2078, 2007.

P. V. Zbitskiy, "Code mutation techniques by means of formal grammars and automatons," *Journal in Computer Virology*, vol. 5, no. 3, pp. 199–207, 2009.

C. Lattner and V. Adve, "Architecture for a next-generation gcc," in *GCC DevelopersâĂŹ Summit*, 2003.

"Retargetable decompiler," May 2017. [Online]. Available: https://retdec.com/

M. Stamp, *Information security: principles and practice*. John Wiley & Sons, 2011.

J. Landage and M. Wankhade, "Malware and malware detection techniques: A survey," in *International Journal of Engineering Research and Technology*, vol. 2, no. 12 (December-2013). IJERT, 2013.

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.

S. Venkatachalam, "Detecting undetectable computer viruses," 2010.

A. Venkatesan, "Code obfuscation and virus detection," 2008.

M. Stamp, "A revealing introduction to hidden markov models," *Department of Computer Science San Jose State University*, 2004.

S. Kazi and M. Stamp, "Hidden markov models for software piracy detection," *Information Security Journal: A Global Perspective*, vol. 22, no. 3, pp. 140–149, 2013.

"Coreutils - gnu core utilities," May 2017. [Online]. Available: https://www.gnu.org/software/coreutils/coreutils.html

"Malwr - malware analysis by cuckoo sandbox," May 2017. [Online]. Available: https://malwr.com/

D. Baysa, R. M. Low, and M. Stamp, "Structural entropy and metamorphic malware," *Journal of computer virology and hacking techniques*, vol. 9, no. 4, pp. 179–192, 2013.

"List of llvm optimizer passes," May 2017. [Online]. Available: http://llvm.org/docs/Passes.html

"Llvm analysis and transform passes," May 2017. [Online]. Available: http://llvm.org/docs/Passes.html#id63

"Llvm command guide tool," May 2017. [Online]. Available: http://llvm.org/docs/CommandGuide/

"Llvm helloworld in c, overview on llvm tools and explains how to compile code using llvm," May 2017. [Online]. Available: http://projects.prabir.me/compiler/wiki/LLVMHelloworldInC.ashx

"Llvm ir bytecode format," May 2017. [Online]. Available: http://llvm.org/releases/1.3/docs/BytecodeFormat.html

"Llvm programming manual," May 2017. [Online]. Available: http://llvm.org/docs/ProgrammersManual.html

"llvm-link tool manual page," May 2017. [Online]. Available: http://llvm.org/docs/CommandGuide/llvm-link.html

"Llvm source code," May 2017. [Online]. Available: http://llvm.org/releases/download.html

S. Priyadarshi, "Metamorphic detection via emulation," 2011.