San Jose State University
SJSU ScholarWorks

Master's Projects | Master's Theses and Graduate Research

Fall 2015

Metamorphic Code Generator based on bytecode of LLVM IR

Arjun Shah
San Jose State University

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

 Part of the Computer Sciences Commons

Recommended Citation

Shah, Arjun, "Metamorphic Code Generator based on bytecode of LLVM IR" (2015). *Master's Projects*. 433.
DOI: https://doi.org/10.31979/etd.zs6a-xpqu
https://scholarworks.sjsu.edu/etd_projects/433

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Metamorphic Code Generator based on bytecode of LLVM IR

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

Arjun Shah

December 2015

© 2015

Arjun Shah

ALL RIGHTS RESERVED

The Designated Project Committee Approves the Project Titled

Metamorphic Code Generator based on bytecode of LLVM IR

by

Arjun Shah

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

December 2015

Dr. Thomas Austin	Department of Computer Science
Dr. Mark Stamp	Department of Computer Science
Dr. Kong Li	Department of Computer Engineering

ABSTRACT

Metamorphic Code Generator based on bytecode of LLVM IR

by Arjun Shah

Metamorphic software is famous for changing the internal structure of the code while keeping the functionality same. In order to escape the signature detection along with some advanced detection techniques, many malware writers have used metamorphism as the means. On the other hand, code morphing technique increases the diversity of the software which is considered to be a potential security advantage.

In our paper, we have developed a metamorphic code generator based on the LLVM framework. The architecture of LLVM has a three-phase compiler design which includes the front end, the optimizer and the back end. It also gives assistance to various source languages and designs which can be considered as a target. LLVM Intermediate Representation(IR) is the most important aspect of LLVM that uses a common IR bytecode within its optimizer. As a result of this, the compilation process of LLVM can transform any high-level language to its IR bytecode. The metamorphic code generator that we have developed works at this IR bytecode level. Leveraging on the dead code obfuscation technique from the previous research, we have implemented a much more difficult technique of instruction substitution at the IR bytecode level. Hence this paper discusses the implementation of obfuscation techniques like dead code insertion, subroutine reordering, and instruction substitution. The effectiveness of these techniques have been tested by using the Hidden Markov Model.

ACKNOWLEDGMENTS

The journey towards the conclusion of this thesis has involved a lot of important people who have supported my work and helped me throughout the taxing but enjoyable process. I want to begin by thanking the one person who continually encouraged me and took the hours for active involvement in my project, Dr. Thomas Austin, my project advisor. He has been a solid contributor to my ideas and stimulated me to discover novel techniques throughout the duration of the project, and I am extremely grateful to him for his provision.

Additionally, I want to recognize my committee members, Dr. Mark Stamp and Dr. Kong Li who have been giving valuable suggestions and insights. I really appreciate their support in aiding this project to successful completion.

TABLE OF CONTENTS

CHAPTER

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

With the growth in the information technology sector, the threats are also increasing at the same pace. ‘‘Its adversaries evolved from the 15 year old script kiddy, to the professional hacker employed by organized crime’’ [20].

Metamorphic viruses are the viruses which change their appearance while maintaining their functionality. This is a powerful technique in order to evade the signature detection. The pattern recognition of anti-virus software is avoided by making use of the metamorphic code. The metamorphic viruses lead to changing their own binary code into a temporary code obtained by editing the code of their own and then translating back to the machine code.

In Stamp and Wong (2006) and Attaluri et al. (2008), Hidden Markov Model also known as HMMs are used to detect metamorphic viruses - including metamorphic viruses that evaded detection caught by commercial signature scanners [2].

Some metamorphic malware generators available are [2, 4, 32]

- NGVCK (Next Generation Virus Creation Kit) [4]
- MPCGEN (Mass Code Generator) [4]
- G2 (Second Generation Virus Generator) [2]
- VCL32 (Virus Creation Lab for Win32) [32]
- NEG (NoMercy Excel Generator) [2]

- MetaPhor [4]

In this paper, we have designed a metamorphic generator using the framework of the Low Level Virtual Machine (LLVM) compiler [1]. LLVM is a project focusing on the working of compilation with emphasis on Just In Time(JIT) compilation and cross-file optimization (links code together from different languages and optimizes across various file boundaries). For creating components that have few dependencies on each other and at the same time integrating well with other existing compiler tools, LLVM focuses on a modular compiler architecture.

LLVM is a three phase design with the front-end, the optimizer , and the back-end being the major components. The front end parses the source code, checks it for the errors, and builds an Abstract Syntax Tree(AST) which is language specific so that it can represent the input code.

The optimizer along with the back-end runs on the code. The optimizer performs the task of improving the time it takes to run the code, eliminating repetitive computations, and is not dependent on the language. The back end(code generator) is responsible for mapping the code onto the target instruction set. It not only makes the code correct but it also generates code that makes use of random features of the supported architecture. Register allocation, Instruction selection, and instruction scheduling are the common parts of a compiler.

The morphing tool functions at this IR level and also provides benefits which are parallel to the morphing at the source code stage.

The assessment of the morphing technique designed by us is done by hidden Markov model (HMM) analysis [18]. The obfuscation techniques used in this research are the instruction substitution, dead code insertion, and subroutine permutation

method.

Different chapters in our paper are categorized in the following ways. Chapter 2 gives the information of viruses and various techniques to detect it. Chapter 3 discusses the metamorphic techniques that are used by the metamorphic generators. The overall design of LLVM compiler infrastructure and the detailed description of the IR byte code is explained in Chapter 4. Chapter 5 covers the planning and execution of the Hidden Markov Model used to evaluate the test results. The architecture and implementation of the code generator is covered in Chapter 6. Chapter 7 showcases the experimental results. Lastly, Chapter 8 gives the conclusion along with the future work that can be carried out.

CHAPTER 2

Malware

Malware (also known as ‘‘malicious software’’) is a file or code, when delivered over a network is capable of infecting, stealing, exploring or conducting virtually any behavior that an attacker wants [5].

Despite of its various types and different capabilities, malware aims commonly to achieve some of the following goals:

- Provide remote control for an attacker to access a corrupted machine
- Gather and steal data which is sensitive
- Spamming unsuspicious targets from the infected machine

Malware is a term that includes almost all types of softwares having malicious content. Some of them are: Trojans, Viruses, Rootkits, Worms, Spyware [5]. Hence before focusing on metamorphic case generation we will present some basic information on malware and its types.

2.1 Different types of Malware

There are many kinds of malware like Adware, Spyware, Viruses, Worms, Trojans, Rootkits, Backdoors, Keyloggers, Botnets, etc. but we will discuss mainly the prominent ones : Viruses and Worms.

2.1.1 Worms

A worm is a self- replicating computer program which is responsible for spreading malicious code by penetrating into an operating system [8]. Worms can be harmful in many ways. For example, it can arrive in an email, examine your own contacts, and then send a copy of themselves to the others making others think as though it's from you. Worms are well-known for its own extension over different computers, but unlike a virus, it can travel without any human action. Hence the worms are considered to be different than the virus. [12, 13].

2.1.2 Virus

A Virus is a program or a code which is contagious. It reproduces itself when any software is run by attaching itself on another piece of software [7]. Viruses can be responsible for a variety of malicious activities. For example, viruses can be held accountable for corrupting the data, deleting it, using the user's email-id to spread it, or empty everything on a hard disk.

Virus signatures are a way to catch hold of virus patterns residing in the ordinary programs. It does so by scanning through the patterns and most modern antivirus programs try to find it using this approach. Some viruses employ techniques in order to avoid the above signature detection. Such viruses alter their code on each infection. Some techniques are discussed to evade the signature detection in the following subsections.

2.1.2.1 Encrypted Viruses

One of the basic methods of escaping signature detection is to encrypt the virus body and this is known as simple encryption. The virus is comprised of a decryption

component and virus code's encrypted copy. Considering the above logic, a virus scanner is able to indirectly detect the presence of the virus.

XORing every byte with some constant value in a virus to have an exclusive-or equation is an old technique of encryption [6].

2.1.2.2 Polymorphic Viruses

Polymorphic infections defeat the weakness of the decryptor segment of encoded viruses which is not encrypted. It works the same way as encrypted viruses but in this virus, the decryption module is additionally adjusted on every iteration. Subsequently this sort of infections leave no parts that stay same between infections making it extremely hard to identify it utilizing signatures. [6, 10].

But still an antivirus software can detect these viruses by using a code emulator. The emulator is responsible for emulating the decryption cycle and decrypting the encrypted virus body dynamically.

2.1.2.3 Metamorphic Viruses

Just as polymorphic viruses beat the weakness of the encrypted viruses, in the same way metamorphic viruses overcome the shortcoming of polymorphic viruses. Many advanced metamorphic techniques have been created by the virus writers in order to make viruses more impervious to copying. A metamorphic virus is known for changing its decryptor on every infection along with its virus body.

According to Muttik [14], ''Metamorphics are bodypolymorphic''. Newly generated viruses appear completely different from each other, not decrypting to the same virus content and hence making it difficult for the detection. A metamorphic virus

changes its ‘‘shape’’ but not its conduct. The diagrammatic representation of this is given by Szor and is shown in Figure 1.

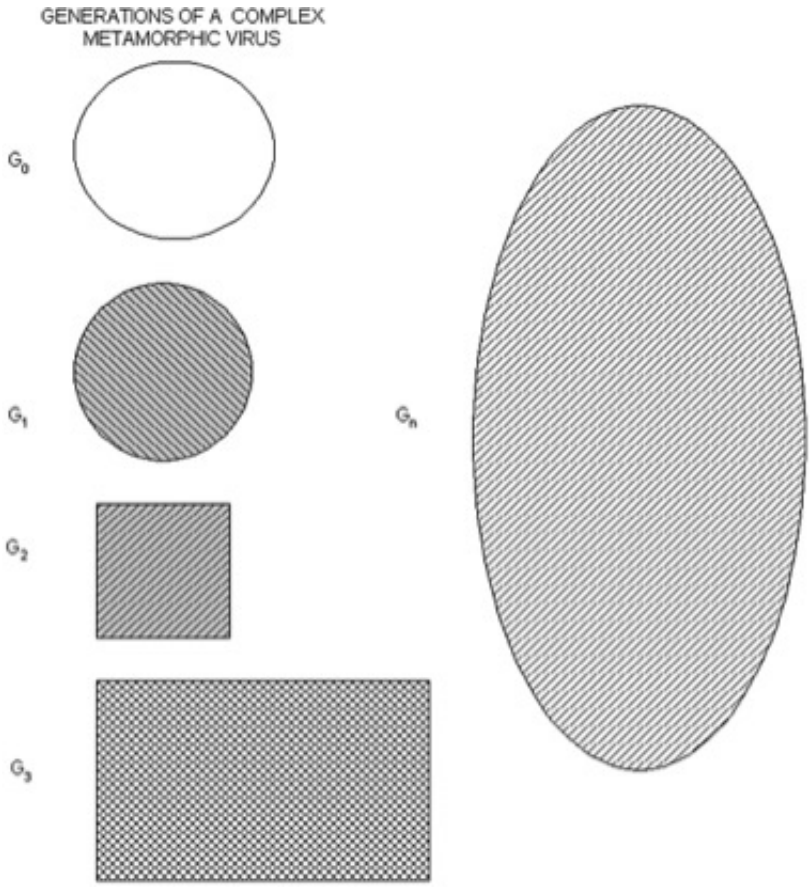

Figure 1: Various forms of a metamorphic virus [9]

2.2 Malware Detection Techniques

In order to counter the malware evolving there are many malware detection mechanisms that have been incorporated. We will discuss some of the detection techniques in the following sub sections.

2.2.1 Signature Based Detection

This technique searches for a known identity or a signature that it has encountered before for each intrusion that takes place. It creates a copy of the malware that is already known. The signature can represent various forms like a cryptographic hash of a file or a byte sequence in the file. This method of detecting malware is one of the most common method that antivirus software uses. There is one major limitation of this detection method. It cannot detect the malicious files whose signature does not exist in the database. Knowing this fact, many attackers alter their codes by writing ''polymorphic'' and ''metamorphic'' viruses to escape the virus signature match and retain the malicious functionality [17].

2.2.2 Heuristics Based Detection

Heuristics based detection approach aims to overcome the loophole left behind by the signature based detection. It is normally used to find new infection in the files that have not been seen or encountered before. For example, an antivirus tool might search for the presence of uncommon instructions or junk code in the processed file. ''Some common heuristic techniques are File Emulation, File Analysis and Generic Signature detection'' [22]. File Emulation technique runs the file in virtual mode to see what it would do if executed. Whereas a File Analysis method scrutinizes the file in order to make sure that the intent and the purpose of the file is not to harm anyone. The biggest drawback of heuristics is in wrongly classifying the valid files as malicious files [17].

2.2.3 Behavioral Detection

Rather than emulating the execution, this technique sharply observes how the program executes. Behavioral detection tries to find a malware by observing behaviors that are suspicious, such as unpacking of malicious code, observing keystrokes or changing the contents of the hosts file. An antivirus tool can take advantage to detect the presence of malware which is not seen on the protected system by noticing such actions. The actions mentioned above might not be considered strong enough individually to be identified as a malware in case of heuristics. But, when considered as a group of actions, they can suggest that there is a malicious program. Antivirus tools making use of such behavioral techniques are classified in the category of host intrusion prevention systems (HIPS) [17].

2.2.4 Cloud-Based Detection

Cloud based detection collects the data from protected computers and instead of analyzing it locally it analyses it on the provider's system. It captures the relevant information about the file and provides it to the cloud system to process it.

Hence the processing that needs to be done by local antivirus software becomes minimum. Moreover, patterns related to malware properties can be derived by the cloud engine vendor by collecting data from various machines.

Some antivirus components conclude by observing the attributes locally which is in contrast to this detection method [17]. Being an individual user, one can take advantage from the experience of the other users by making use of the cloud-based engine.

2.2.5 Hidden Markov Model Based Detection

Hidden Markov models (HMMs) are well known for detecting mathematical and analytical patterns. ‘‘They can be viewed as a machine learning technique and as a discrete hill climb’’ [30]. An HMM can be categorized as a machine learning tool where a user only needs to mention some essential parameters. Any dataset can be trained using the HMM. If the training is successful, then it can be used to test and find data that is similar. The testing data can be given scores based on the training model. Higher scores suggests that the testing data matches the training model.

CHAPTER 3

Metamorphic Techniques

Metamorphic viruses are known for obfuscating the content by making use of multiple techniques which include instruction substitution (replacing existing instructions with other equivalent ones), register exchange (making use of unique registers in each generation), permutation (subroutine reordering), dead code insertion (adding NOP and other ‘‘do nothing’’ lines) and transposition (reordering instructions which are not order dependant). The alternate way to obfuscate is to make use of multiple jump instructions. We have used instruction substitution, dead code insertion and subroutine reordering in our metamorphic code generator. The following sub sections discuss some of the metamorphic techniques.

3.1 Register exchange/swap

Register swap is a straightforward method switching the register from batch to batch. It accomplishes this without changing the system code and its conduct. Figure 3 shows the application of this technique. The original code shown in Figure 2 is changed by applying this technique (used by Win95/Regswap virus). An example to illustrate this is: ‘‘PUSH EBX’’ can be substituted with ‘‘PUSH EDX’’ [15]. Random searching can make this technique unusable.

3.2 Subroutine Reordering

Subroutine reordering is a technique in which the original code is obfuscated by changing the function order. This changing of subroutines is done in a random manner. For a code having n number of subroutines, this technique can generate n!

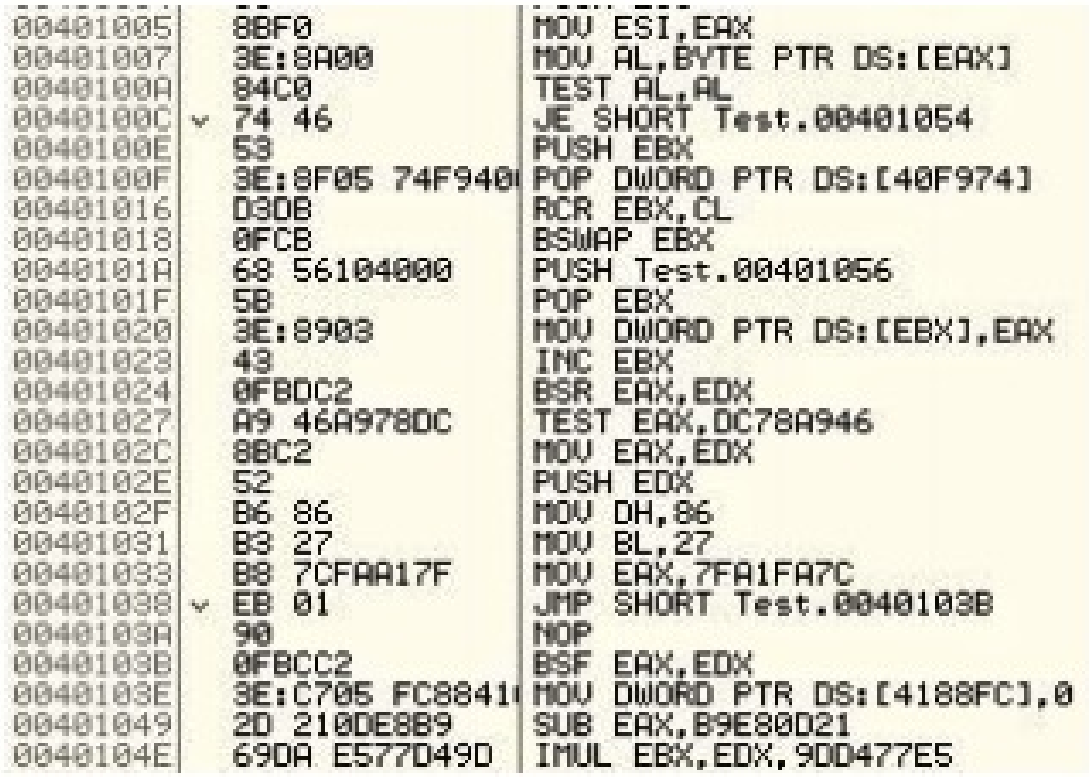

Figure 2: A Sample Code [15]

different combinations. An example to illustrate this is, Win32/Ghost having ten subroutines leads to 10! = 3628800 different generations [15]. Search strings can detect this, whereas some scanners might need to face it mathematically. Figure 4 explains this technique.

3.3 Instruction Substitution

Instruction substitution is used for updating/changing the existing code by replacing some lines of code with other lines that are equivalent. This obfuscation technique makes use of the fact that there can be more than one way of representing an instruction/operation. One of the example of instruction substitution is to replace XOR with a SUB and MOV with PUSH/POP. Figure 5 shows another example of

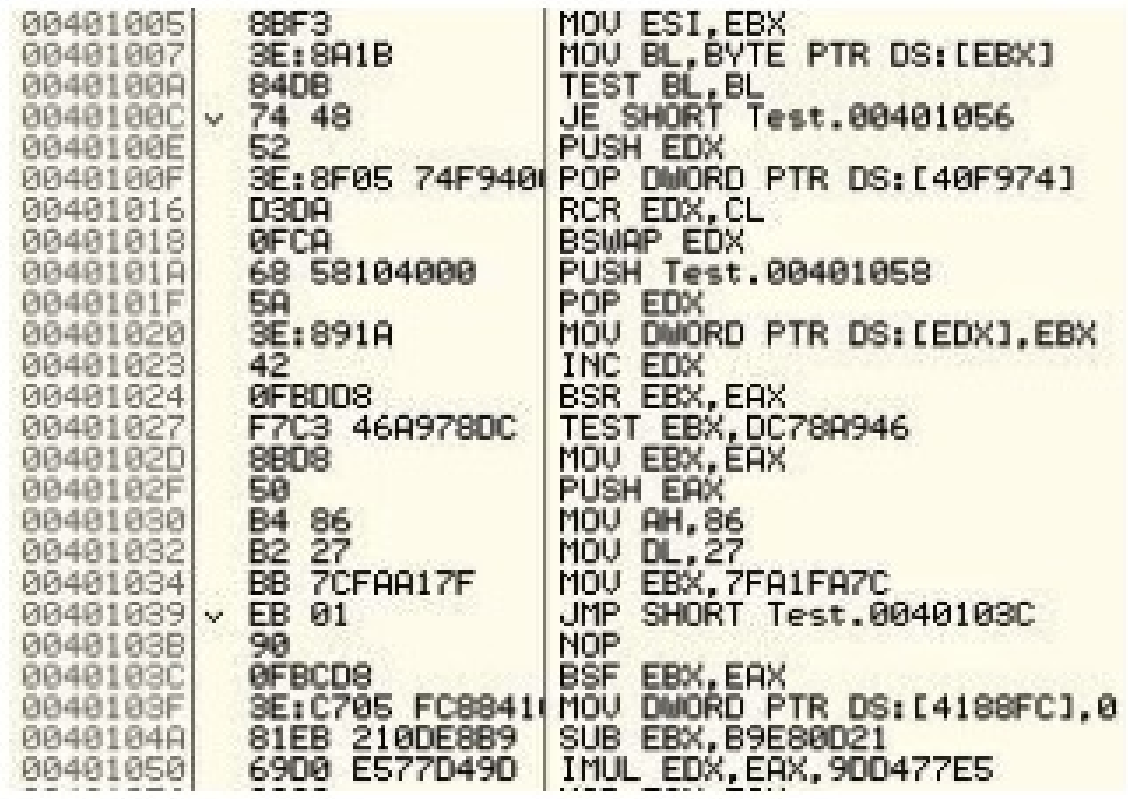

Figure 3: Register Swap [15]

instruction substitution.

Instruction substitution evades signature detection and hence is known to be a powerful technique. However, implementing instruction substitution at assembly code level is not so easy. The ‘‘W32/MetaPhor’’ [11] virus makes use of the substitution technique.

3.4 Code Transposition

Code transposition is a way to permute the code. The grouping of the code guidelines of an original code are reordered such that the behavior of the code is not changed. There are two methods to achieve this as shown in Figures 6 and 7 respectively.

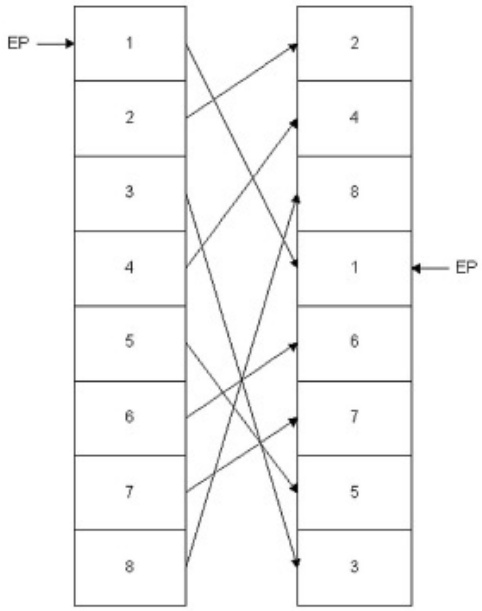

Figure 4: Subroutine Permutation [11]

The first method shown in Figure 6 makes use of random shuffling to change the order of instructions randomly, and then it makes use of the unconditional branches and jumps to recover the original execution order. Since the original code can be recovered without any difficulty by getting rid off the unconditional loops and jumps, this method is not at all difficult to beat.

However, the second method chooses and reorders the instructions in such a way that they have no impact on one another. This creates new code structure that is independent. This method is difficult to implement because it is not easy to find the independent instructions. An example of this method is illustrated by Figure 7 [15].

3.5 Code Integration

This is a technique established by the Win95/Zmist malware [11](also known as Zmist). In this technique a malware fuses itself to the target program code. The

Figure 5: Instruction Substitution (Figure 2 shows the original code) [15]

execution of this technique occurs through various stages. The first stage is the de-compilation of its resultant program into manageable objects. In the second stage, the objects add itself among themselves and finally it reconstitutes the combined code into a fresh generation in the last stage.

3.6 Dead-Code Insertion

Dead-code insertion is a fundamental strategy technique that adds some garbage instructions which are not effective to a program which changes the way it looks while keeping its behavior the same [16, 18, 19]. These dead instructions do not alter the logic of the original program. The instruction NOP is an example of it. Inserting NOP instructions can easily obfuscate the original code and this is shown in Figures 8

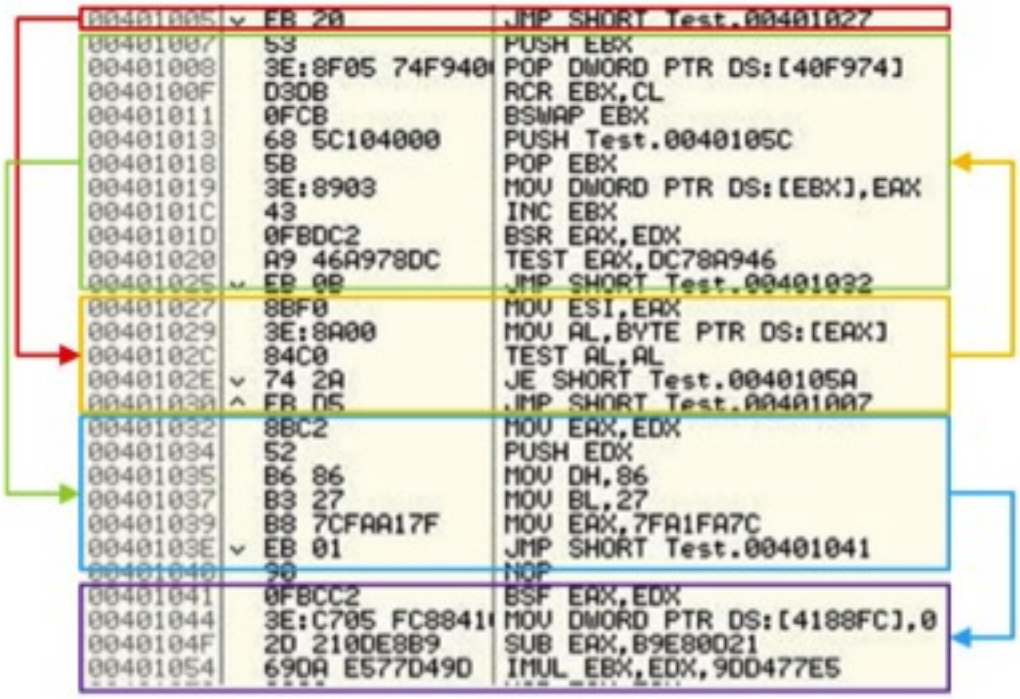

Figure 6: Code Transposition : Unconditional Branches [15]

and 9. However, signature based detection can remove the dead junk lines before the analysis. Figure 9 [15] shows some code that makes the detection more difficult.

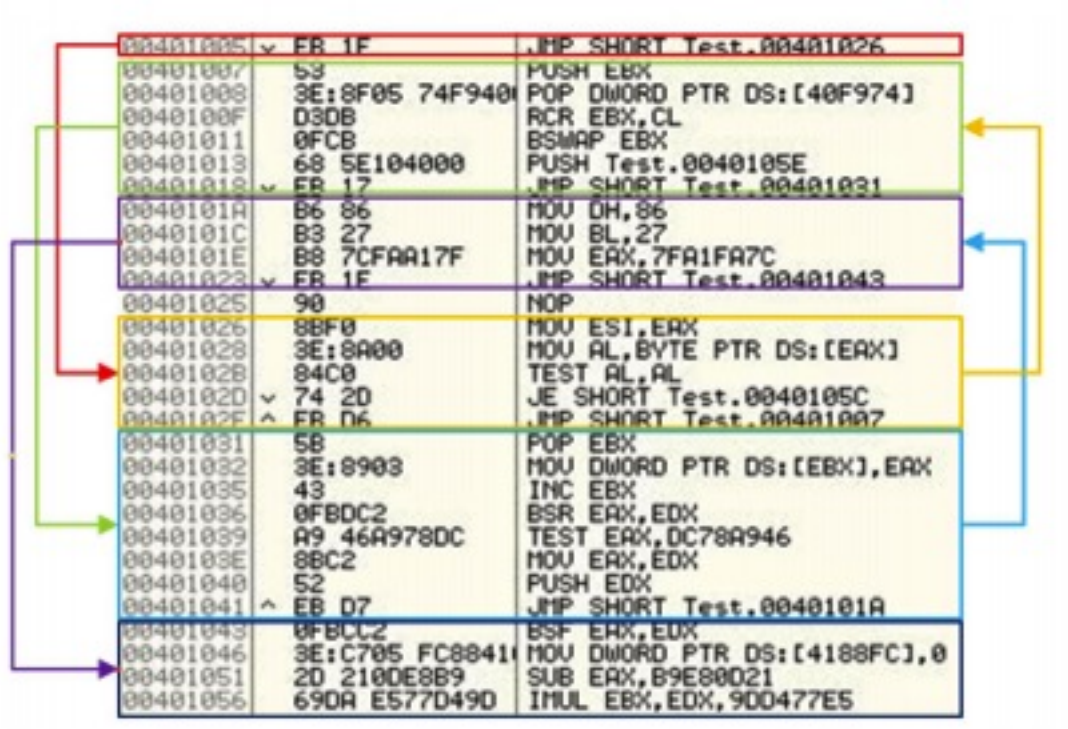

Figure 7: Code Transposition : Independent Instructions [15]

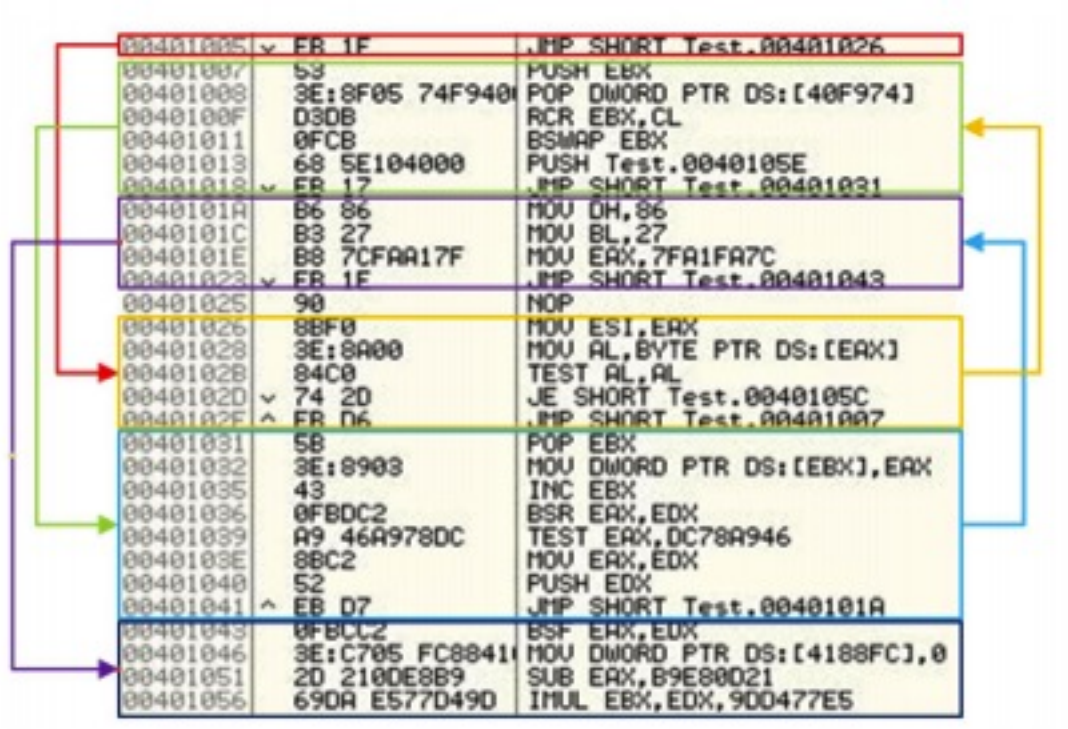

Figure 8: Dead-Code Insertion [15] (Figure 2 shows the original code)

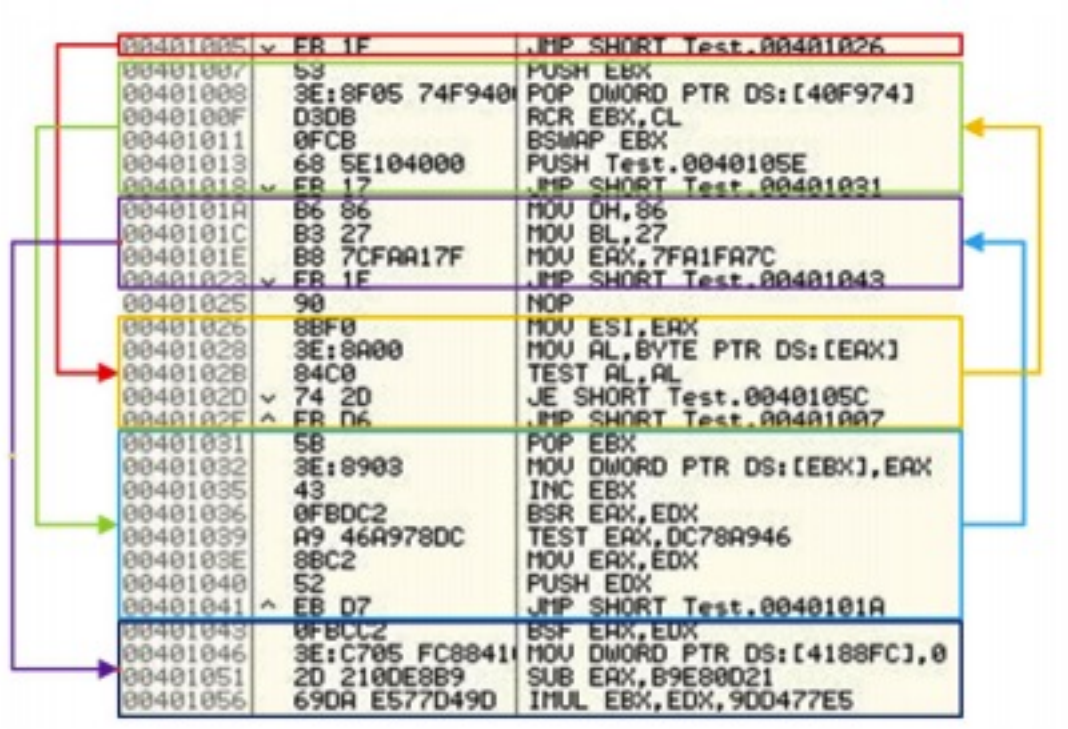

Figure 9: Ineffective Code Sequences [15]

CHAPTER 4

LLVM

4.1 LLVM Introduction

LLVM(Low Level Virtual Machine) is a base responsible for developing and hosting a set of low-level tool components which includes compilers, debuggers, assemblers, etc. These tools are designed in such a way that they match the tools used on UNIX machines. LLVM has several advantages compared to the GCC compiler. The reason for this is its tools which include C, C++ and Objective-C compiler. The unique design in itself is a stand out feature of LLVM when compared to the other compilers [24, 29].

LLVM is now used as a common infrastructure and it supports various known as well as lesser known languages. These languages include Python, Ruby, Java, .NET, Haskell, etc. With LLVM being accepted broadly, it is also recommended over some special compilers which includes Adobe's compiler and Apple's OpenGL stack.

4.2 Classical Compiler Design

The most traditional static compiler with popular design (like the majority C compilers) includes components like the front end, the back end and the optimizer. As mentioned earlier it is known as the three phase design [23]. Figure 10 shows this design. Various tasks such as parsing the source code, and checking the errors is taken care by the front end.

It is the responsibility of the optimizer to perform a wide variety of transformations and improve the code's running time. For instance, to eliminate redundant

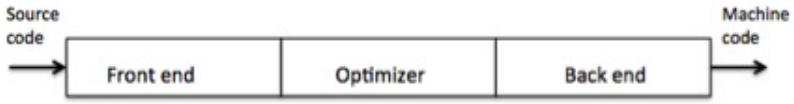

Figure 10: Three components of a compiler

computations and to see if it is not dependant on the language are its role. Mapping the code onto the target instruction set is then done by the back end. It is responsible for generating good code besides making correct code, taking the benefit of unusual features of the supported architecture. Instruction selection, instruction scheduling and register allocation are the common parts of a compiler back end. Figure 11 shows the LLVM design. This model is equally applicable to JIT compilers as well as to interpreters. An implementation of this model is the Java Virtual Machine (JVM).

4.2.1 Advantages of the design

Compiler, with the help of this design, can be used to support a new source language which needs to have a new front end, whereas the other two components in the three phase can be reused. Implementation of a new source language would be required if these parts were not separated, so to support N targets and M source languages N*M compilers would be needed.

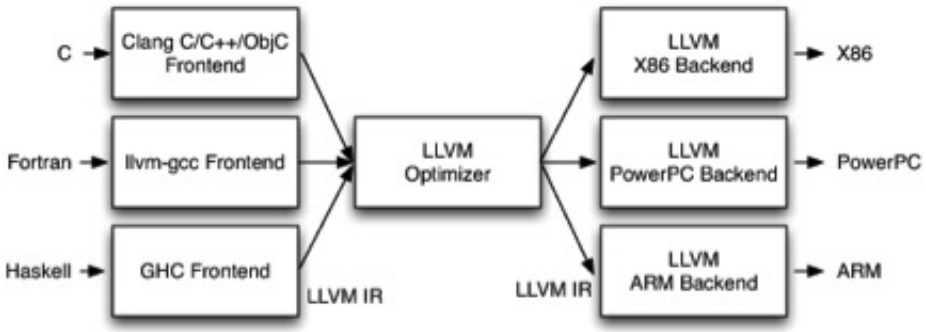

Figure 11: LLVM design [23]

If the compiler had supported only one target and source language then it would not have served a wide bunch of programmers. The skills that are required to implement one component from the three phase is completely different from the remaining components.

4.3 LLVM IR

LLVM Intermediate Representation (IR) is the most important part of LLVM. LLVM IR exists in such a way that it can withstand mid-level transformations and analysis of a compiler. The LLVM IR is responsible for supporting trivial optimizations, cross-functional optimizations, and analysis of program etc. It is known to be a first class language with well structured semantics. An example of .ll file is shown in the Figure 12.

C code is corresponded by this LLVM IR and it provides two different ways to add integers as shown in Figure 13:

LLVM IR is a low-level virtual machine which supports linear sequences of basic instructions like compare, addition, and subtraction. Labels are supported by LLVM IR and looks generally like a weird form of assembly language.

With a simple type system, LLVM is strongly typed (e.g. i32** is a pointer to pointer to 32-bit integer, i32 is a 32-bit integer) unlike most RISC instruction sets and few machine details are hidden. The LLVM IR does not make use of permanent collection of named registers. Whereas it uses a collection of temporaries starting with a % character.

LLVM IR is not only used and executed as a language, but it takes various similar forms like the textual format, dense on-disk ‘‘bit code’’ format and a data structure

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse

recurse:
  %tmp2 = sub i32 %a, 1
  %tmp3 = add i32 %b, 1
  %tmp4 = call i32 @add2(i32 %tmp2, i32 %tmp3)
  ret i32 %tmp4

done:
  ret i32 %b
}
```

Figure 12: Example of .ll file [23]

```
unsigned add1(unsigned a, unsigned b) {
  return a+b;
}

// Perhaps not the most efficient way to add two numbers.
unsigned add2(unsigned a, unsigned b) {
  if (a == 0) return b;
  return add2(a-1, b+1);
}
```

Figure 13: C code [23]

supported in-memory form.

LLVM IR's structure consists of the following components:

- A module: It holds functions and global variables.
- Functions: They represent the set of instructions.
- Global variables: They are the values accessible by all functions.

4.4 LLVM Tools

There are different tools accessible in LLVM framework. These tools are responsible for compiling a program, generating byte code and running optimizations. Some of those tools are explained in the sections below.

4.4.1 Command for converting (.c → .s) : llvm-gcc

The LLVM IR bytecode is generated with the help of this tool. It is utilized to create the IR bytecode of the source program. The extension used is .ll or .s. An example on how to use this command is as follows:

$$llvm\text{-}gcc\ \text{-}emit\text{-}llvm\ \text{-}S\ test.c$$

$$clang\ \text{-}S\ \text{-}emit\text{-}llvm\ test.c$$

This command would generate test.c in the same directory. By default Clang works just like GCC.

4.4.2 Command for converting (.s → .bc) : llvm-as

This command reads the .s(assembly) file and translates it to LLVM bit code and lastly it writes the result into a standard output or a file.

llvm-as -f test.s

4.4.3 Command for converting (.bc → .s) : llvm-dis

This command disassembles the .bc(LLVM bitcode) file into .s(assembly) file which can be read by a human.

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

4.4.4 Opt and LLVM Passes to call the Dead Code and implement Instruction Substitution

The modular LLVM optimizer and analyzer is the opt command. LLVM source files are taken as an input and it runs the specified analyses or optimizations on it and then it outputs the analysis results or optimized file.

We have developed our own optimizer passes which calls the dead code and implement instruction substitution method respectively. Passes usually perform some interesting tasks like analyzing the results, performing transformations and optimizations which forms the compiler. LLVM has many existing ‘‘Pass classes’’ and some of the passes already exist which can remove the dead code that is prevalent in the file. LLVM's compiler itself is smart enough to remove the dead code from the file and we need a way to overcome this. So we make our metamorphic code generator to use the optimizer pass that we create. This optimizer pass is made to operate on the ModulePass class. The reason for inheriting from the ModulePass class is to use the whole program as a single unit which accesses the functions in no specific order keeping its behavior unknown. Hence while executing, we cannot perform the

optimization.

Once we define our pass to call the dead code, we can implement it by using the opt command. opt makes use of the -load option to load the pass and execute it.

The following is an example of our command:

$$opt < name\ of\ optimizer > < input\ file\ bytecode\ > < output\ file\ bytecode\ >$$

Similarly we have written an optimizer pass for the instruction substitution that operates on the FunctionPass class. It can be implemented in the same way as shown above by using the opt command.

The various instruction substitution techniques that we have covered are as follows :

1. Addition: $a + b$ changes to $a - (-b)$.
2. Subtraction: $a - b$ changes to $a + (-b)$.
3. XOR:
   (a) $a\hat{}a$ changes to $a - a$
   (b) $a\hat{}b$ changes to $(\tilde{}a\&b)|(a\&\tilde{}b)$

The reason for inheriting from the FunctionPass class is that it works independently on each function, does not add or remove the functions and global variables from its current module.

4.4.5 llc

The llc command for a specified architecture compiles LLVM source inputs into assembly language.

llc -f opttest.bc

A file opttest.s is created.

4.4.6 lli

This command takes the input in the bitcode form and executes the result using an interpreter.

lli test.s.bc

test program is executed by this command.

4.4.7 llvm-link

This command links more than one input file into a single file.

For example,

llvm-link -o final.bc addCode.s.bc origin.s.bc

CHAPTER 5

Hidden Markov Models (HMM)

Metamorphism is considered to be an advanced code obfuscation method which is strong enough to escape the signature detection. But it is not difficult today by using construction kits to generate metamorphic variants. It is a challenge to detect metamorphic viruses. If the metamorphic generator is designed in a proper way then detecting metamorphism via signature is difficult [25].

For speech recognition and a variety of other applications Hidden Markov Models (HHMs) is extensively used. They are also considered to be a machine learning method. HMMs are employed with very promising results to detect metamorphic viruses. A method is used over a past period of time where HMM is trained with sequence of opcodes which belong to the same family from viruses. The HMM model that is trained is helpful in determining and scoring binaries and categorize them into a virus or a valid file. Log Likelihood Per Opcode (also known as LLPO) score is computed and threshold is attained for viruses and benign file based on this score. To categorize between benign files and viruses this threshold value is used based on the LLPO score. The following sections describe the detailed application and discussion of HMMs.

5.1 HMM

An HMM is considered to be a statistical model having hidden states within the system that is being modeled. The system referred here is the Markov process. HMM uses the following fundamentals and are as follows [14]:

1. Say N, are the number of finite states in the model; what a state is shall be

rigorously defined.

2. A new state is entered at each clock time, t, based upon a conversion probability, and this depends on the state previously defined.

3. An observation output symbol is produced after each conversion is made on the basis of probability which is dependent on the present state.

HMM model uses the following formal notations and they are as follows [14]:

T = length of the observation series (aggregate number of clock times)

N = number of states

M = number of observation symbols

Q = $q_1, q_2, ..., q_N$ states

V = 0,1,.....,M-1 discrete set of possible symbol observations

A = probability distribution of state transition

B = probability distribution of observation symbol

Pi = starting state distribution

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

Figure 14 explains a generic Hidden Markov Model. X_t and O_t represents the observation and its state at time t respectively. The present state determines the Markov model and it also determines the A matrix. Hidden markov process is controlled by the introductory state X_0 and A matrix. Oi is identified with the process states by the matrices A and B.

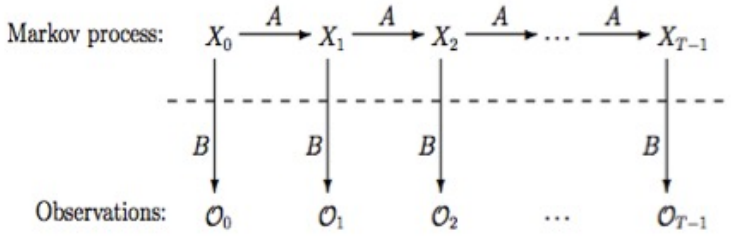

Figure 14: Commonly used HMM [14]

5.1.1 Example of HMM

The inner working of HMM with a simple example is explained in the papers [3,14]. Consider that one has to determine the average temperature over the year for any given year taking into account the perception of diverse tree sizes (S-small, M-medium, L-large). Assumption of the annual temperature being either hot (H) or cold (C) is taken to keep the example simple. In addition to this, lets assume that the behavior of the probabilities of the annual temperature is known. The probability trend numbers are [14] :

- 0.7 is the probability of the hot year followed by another hot year (HH)
- 0.3 is the probability of the hot year followed by a cold year (HC)
- 0.4 is the probability of the cold year followed by a hot year (CH)
- 0.6 is the probability of the cold year followed by another cold year

The above information about the probabilities is represented in the form of a matrix shown below:

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

Below are the assumed points that explain the correlation between tree sizes and temperature [14]:

- In a hot year, the probability of a tree size,
  - 0.1 is the probability of being small
  - 0.4 is the probability of being medium
  - 0.5 is the probability of being large
- In a cold year, the probability of a tree size,
  - 0.7 is the probability of being small
  - 0.2 is the probability of being medium
  - 0.1 is the probability of being large

The temperature and the tree sizes are related by their probabilities which is as follows:

$$\begin{array}{c} \\ H \\ C \end{array} \begin{array}{c} \begin{array}{ccc} S & M & L \end{array} \\ \begin{bmatrix} 0.1 & 0.4 & 0.5 \\ 0.7 & 0.2 & 0.1 \end{bmatrix} \end{array}$$

All the information stated above can be associated with the HMM notations. The annual temperatures can be considered as the states, and tree sizes as the observable symbols. The states H and C are considered to be hidden since we do not have any information about the temperatures in the past. Whereas, the observation symbols can be known. We can build the HMM model based on the above know information as shown in Figure 15 [3].

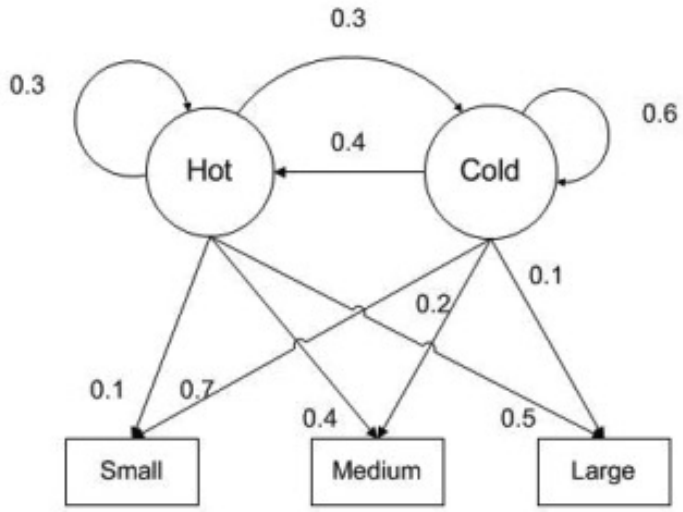

Figure 15: HMM model [17]

Assuming that the data of tree size groupings (observation symbols) for four continuous years i.e. (S, M, S, L) is known, we can find the yearly temperature series.

In order to tackle this issue using HMM, we can define the HMM parameters as follows [14]:

1. Matrix for state transition probability

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

2. Matrix for observation probability distribution

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

3. N = 2 is the number of states

4. Number of unique observation symbols (M=3, {S, M, L})

5. Given initial state distribution matrix

$$\pi = \begin{bmatrix} 0.6 & 0.4 \end{bmatrix}$$

Following are the HMM steps to calculate the state transition having length T = 4 for a given observation (S, M, S, L) [14, 32]:

1. Determining all state transitions that are possible (NT)

2. With the given observation sequence calculate the probability for each state transition (Figure 16). The formula for calculating the probability of sequence HHCC is as follows:

$$P(HHCC) = \pi_H b_H(S) a_{H,H} b_H(M) a_{H,C} b_C(S) a_{C,C} b_C(L)$$
$$= 0.000212$$

3. The yearly temperature series has the highest probability. According to Figure 16 [14], CCCH has the highest probability and hence that will be the answer.

The above process of generating the HMM results needs extensive work to be done which is not possible. This method is known as the brute force method.

5.2 Discussion of Three problems and its solution

In this section we will discuss three problems in which we are interested and the solution provided by three efficient algorithms included in the HMM [14, 32].

Problem 1: Considering a model $\lambda = (A, B, \pi)$ and an observation sequence O to be known, find P(O|λ). This is nothing but the chance of getting the sequence O when a model is given.

Problem 2: Considering the above model, find an ideal state series for the Markov process. This means to find the hidden part of HMM.

state sequence	probability
HHHH	0.000412
HHHC	0.000035
HHCH	0.000706
HHCC	0.000212
HCHH	0.000050
HCHC	0.000004
HCCH	0.000302
HCCC	0.000091
CHHH	0.001098
CHHC	0.000094
CHCH	0.001882
CHCC	0.000564
CCHH	0.000470
CCHC	0.000040
CCCH	0.002822
CCCC	0.000847
Σ probability	0.009629
max probability	0.002822

Figure 16: List of probabilities in observing (S, M, S, L) possibilities of state sequence [14]

Problem 3: Considering an observation sequence O, values of N and M to be known, find the model $\lambda = (A, B, \pi)$ that maximizes the probability of O. This means to train a model in order to best fit the data seen.

In our paper, we have trained a model on the opcode sequence derived from the base software/files(Problem 3). Once the model is trained, we use it to score the morphed versions/file of this base software (Problem 1). The research carried out earlier proves that HMMs are effective enough to detect if a metamorphic malware is present or not [26].

5.2.1 α pass or Forward Algorithm

This algorithm also known as α pass resolves the problem of finding P(O|λ) [14].

For $t = 0, 1, ..., T-1$ and $i = 0, 1, ..., N-1$, define

$$\alpha_t(i) = P(O_0, O_1, ..., O_t, x_t = q_i|\lambda)$$

The calculation of $\alpha_t(i)$ and P(O|λ) is as follows [14]:

1. Let $\alpha_0(i) = \pi_i b_i(O_0)$, for $i = 0, 1, ..., N-1$

2. For $t = 1, 2, ..., T-1$ and $i = 0, 1, ...N-1$, compute [14]

$$\alpha_t(i) = \left(\sum_{j=0}^{N-1} \alpha_{t-1}(j) a_{ij} \right) b_i(O_t)$$

3. P(O|λ) = $\sum_{i=0}^{N-1} \alpha_{T-1(i))}$

Hence the forward algorithm discussed above is better than the naive approach since it requires only N^2T multiplications compared to $2TN^T$ for the latter.

5.2.2 Backward Algorithm

This algorithm resolves the problem of finding the most likely state sequence [14, 32].

For $t = 0, 1, ..., T-1$ and $i = 0, 1, ..., N-1$, define [14,32]

$$\beta_t(i) = P(O_{t+1}, O_{t+2}, ..., O_{T-1}|x_t = q_i, \lambda)$$

$\beta_t(i)$ can be calculated efficiently as follows [14, 32]:

1. Let $\beta_{T-1}(i) = 1$, for $i = 0, 1, ...N-1$

2. For $t = T-2, T-3, ..., 0$ and $i = 0, 1, ...N-1$, compute [14]

$$\beta_t(i) = \sum_{j=0}^{N-1} a_{ij} b_j(O_{t+1}) \beta_{t+1}(j)$$

For $t = 0, 1, ..., T-2$ and $i = 0, 1, ..., N-1$, define

$$\gamma_t(i) = P(x_t = q_i|O, \lambda)$$

$\alpha_t(i)$ is used to measure the relevant probability up to time t and $\beta_t(i)$ is used to measure probability after time t [14],

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{P(O|\lambda)}$$

A possible state at any time t is the state when $\gamma_t(i)$ is maximum.

5.2.3 Baum-Welch Algorithm

This algorithm solves the problem of assessing the model parameters to best fit the observations. The values of the parameters N and M are fixed, whereas the values of A, B and π are unknown and needs to be found. The steps to re-estimate the model are as follows [14]:

1. Initialize $\lambda = (A, B, \pi)$ with some random values. For instance, $\pi = 1/N, A_{ij} = 1/N, B_{ij} = 1/M$,

2. Find the values of $\alpha_t(i), \beta_t(i), \gamma_t(i)$ and $\gamma_t(i, j)$ where $\gamma_t(i, j)$ is di-gamma. Di-gamma is defined as follows [14]:

$$\gamma_t(i) = \frac{\alpha_t(i)a_{ij}b_j(O_{t+1})\beta_{t+1}(j)}{P(O|\lambda)}$$

$\gamma_t(i)$ and $\gamma_t(i, j)$ are in relation as follows:

$$\gamma_t(i) = \sum_{j=0}^{N-1} \gamma_t(i, j)$$

3. Re-estimate the model as follows:

   For $i = 0, 1, ..., N-1$ let $\pi_i = \gamma_0(i)$

   For $i = 0, 1, ..., N-1$ and $j = 0, 1, ..., N-1$, compute [14]

$$a_{ij} = \frac{\sum_{t=0}^{T-2} \gamma_t(i,j)}{\sum_{t=0}^{T-2} \gamma_t(i)}$$

   For $j = 0, 1, ..., N-1$ and $k = 0, 1, ..., M-1$, compute [14]

$$b_j(k) = \frac{\sum_{t\epsilon(0,1,..,T-1),O_t=k} \gamma_t(j)}{\sum_{t=0}^{T-2} \gamma_t(j)}$$

4. If $P(O|\lambda)$ increase then go to the 3rd step

5.3 Behavior of HMM as a Virus Detection Tool

HMM needs some training data as an input to generate a training model so that it can behave as a virus detection tool. The detailed discussion about its effectiveness is discussed in [3]. The mathematical and analytical properties of the virus group are represented by trained HMM.

Once the model is trained, it can be used to distinguish whether a given/incoming file is similar to the virus group that was trained in the training set. In order to have the HMM trained, a bunch of virus files falling into the same group are disassembled. From each of these files, unique opcode sequences are extracted which represent the HMM symbols. Figure 17 shows the extracted opcode sequence.

A lengthy observation series is created by merging all the opcode series from all the infected files belonging to the same group. HMM is then trained using this concatenated sequence. All the unique opcodes found in this sequence constitute a set of distinct observation symbols.

Figure 17: Extracted opcode sequence

CHAPTER 6

Planning and Execution

6.1 Overview

The metamorphic techniques discussed earlier are carried out at LLVM IR level and not at the assembly level. In the past it has been seen that by writing LLVM optimizer passes, such obfuscation techniques based on some conditions can be executed at the LLVM IR level [17]. Implementation of ‘‘shadow attack’’ by the malware writers is also tried in [34].

To produce multiple base software copies is the aim of the project which are considerably different from each other and are hard to detect. It considerably generates different morphed copy of the base software when a program is complied with the optimizer. HMM detector which is mentioned in [18] is able to categorize between benign and virus files properly even after the implementation of all metamorphic techniques. To escape from HMM-based detector an unsuccessful effort was made hence proving HMM detection to be powerful [11]. So, our goal is to beat this and develop a metamorphic code generator that escapes HMM detection.

6.2 Challenge faced and New Transformation

Following are the advantages [2] that are included in morphing code at IR level :

- Initially the LLVM was developed only for a few languages like C and C++. Later on, due to the unique design of LLVM, it led to the acceptance of many more languages like Haskell, Action Script, FORTRAN, Java, Rust, Python, GLSL,D, etc. The code generator can use any of these languages to produce

metamorphic copies.

- The intermediary stage not dependent on the platform.
- Virtual addresses get allocated at the bitcode level and not at LLVM IR level. There are some problems if we morph at the assembly level and in order to avoid that we morph at the LLVM IR level.

6.3 Goals

The purpose of executing this design is to have the morphed copies function the same way as the base files. Additionally, increasing the modified code in different percentage values should result in a variance of the morphed files from the base files. If the opcode sequences and counts fall more close to the morphing files compared to the base files, then the base file which is morphed will appear more towards a morphing file. As disscussed earlier, HMMs are well known for their detection [2]. Our aim is to develop a metamorphic generator that confuses the HMM and proves it to inefficient.

6.4 Metamorphic Techniques Used

It is not so easy to accomplish some techniques since we morph at the LLVM IR level and this is already described in Chapter 3. For example, it is comparatively difficult to implement code transposition and register swapping at the IR level. Therefore, the code morphing techniques that we consider are subroutine reordering merged with dead code insertion and instruction substitution. The first two morphing strategies will be accomplished by us by randomly inserting selected functions of dead code which are complete from different program files. The functions inserted are then randomized and its order is changed. Hence, a good amount of variation is created

using the above mixture between different morphed copies. Additionally, to all the dead code subroutines we insert CALL statements in order to avoid them from getting identified as the dead code. Following sections discuss all the techniques:

6.4.1 Instruction Substitution

As discussed in Chapter 3, instruction substitution is a technique of replacing an original code with some equivalent instructions. This obfuscation technique makes use of the fact that there can be more than one way of representing an instruction/operation. We make the use of an LLVM pass which substitutes the existing instructions in the code with equivalent instructions.

6.4.2 Dead Code Technique

This technique is used for appending instructions which behave as dead. The results of this instructions would not interfere in other computation. To increase the range of different opcodes is the main goal of adding this code. It becomes complicated inserting the dead instructions because this needs to be done at the LLVM IR level which has a structure similar to RISC. However, by using linker tool llvm-link functions can be easily inserted. We need IR bytecode of morphing files to insert dead code.

Files from core-util Linux command files [24] have been used for the dead code insertion. The selected files behave similar to the operations of the system level.

In Section 6.5.1. a detailed algorithm is explained to optimize and remove dead code option by LLVM. Code extracts which are actually not getting executed gets identified by anti-virus software's in a smart way. They can detect the virus and track to execution sequence. We CALL the inserted dead code to make the metamorphic code generator smarter. Section 6.5.2. explains a detailed algorithm.

6.4.3 Subroutine Reordering

This technique is reponsible for obfuscating the original code by changing the function order. By changing the sequence of the functions, function permutations can be implemented easily as the IR bytecode is in the textual form. This technique helps in escaping from the detector that is based on matching the patterns. A Python code is run on LLVM IR level to produce a different text file with a different layout which has same functions . Section 6.5.3. explains a detailed algorithm.

6.5 Implementation

Four stages have been developed in this project. Following subsections gives a detailed description of them. Figure 18 explains the working of morphing engine.

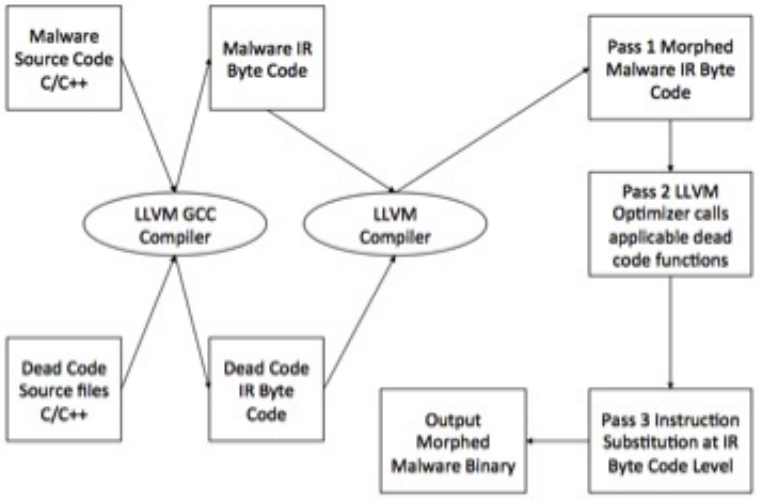

Figure 18: Architecture diagram of a metamorphic code generator [32]

6.5.1 Stage 1 : Dead Code Insertion

The Stage 1 is responsible for inserting the dead code. In order to accomplish this, three parameters need to be specified. They are : a base file, a dead code(%),

and a morphing file (unwanted code is taken from here). The number of lines to be inserted is calculated based on the amount of dead code(%) mentioned. Once the dead code(%) is mentioned, complete functions are copied from the morphing file to the base file. The reason for inserting complete functions into the base file is to match the approximate number of lines that is to be inserted. These modules are merged with the base record amid the connecting stage. The output displays the inserted module names. The dead subroutines can be called in pass 2 and its shown in the output as well.

Following points mention the details of the first pass of our morphing techniques [30]:

- Using llvm-gcc command, compile selected morphing files and generate its IR bytecode.
- Determine the function dependencies from this IR bytecode.
- Calculate its number of lines for each function.
- Decide the functions to be inserted based on the acquisitive strategy. This strategy approximates the total lines to be added based on dead code(%).
- Make an IR byte code file by copying the selected functions. This file is considered to be a temporary file.
- Using llvm-as create temporary IR bytecode files. Bitcode files are also generated for the base file.
- Using llvm-link combine the above two files.
- Finally, the temporary IR file that we created needs to be deleted.

6.5.2 Stage 2 : Calling of Dead Functions

Stage 1 identifies the functions which are dead and they can be called in Stage 2. Here we write an LLVM pass to explicitly CALL the dead code [32]. We run this pass by using the optimizer which inserts the CALL instruction. The opt takes a string as the input which is the function name. It parses the IR bytecode file and searches for the function main. As soon as it finds the main, it inserts a CALL command after every LOAD command. The pass that we have written is based on the ModulePass class. This pass implementation only supports single pointers. The following steps are performed for each dead code subroutine.

- Parse the IR bytecode and find the main function.
- Parse the instructions in the main function and insert the CALL instruction after an instruction of type LOAD is encountered.
- Calculate its number of lines for each function.
- Iterate over its function parameters to insert the CALL instruction. Initialize with a random value and allocate memory for each parameter.
- Insert a CALL instruction, finally.

6.5.3 Stage 3 : Subroutine Reordering

Stage 3 performs a simple task of reordering the functions in LLVM IR and is known as the subroutine reordering or function permutation. A python code runs to implement this stage. The explanation of algorithm is as follows :

- Read the LLVM IR file.

- In temporary file, write all global variables.
- Total number of functions are calculated and assigned to a variable. A random number is generated between 1 to the assigned variable by using random class generator.
- Generate a temporary LLVM IR file and copy the function in it only if it is not added.
- Keep repeating the steps from generating random number and stop only when all the functions are written to the temporary file created in the above step.

6.5.4 Stage 4 : Instruction Substitution

For each function in the IR bytecode file, it checks whether any instruction is replaceable with the other equivalent instructions from the list of techniques mentioned in Section 4.4.4. If it finds one, then it just replaces that instruction with the equivalent instruction. We use the LLVM optimizer in this pass to perform instruction substitution. This optimizer operates on the Function class. The steps for executing the above technique is as follows :

- Iterate over every function.
- Check for every instruction in the function.
- If the instruction is from the list of instruction substitution techniques implemented in the pass (Addition, Subtraction, XOR) then replace it with the corresponding equivalent instructions.

6.6 LLVM configuration

We use the 3.6 version of LLVM source code [33]. LLVM-GCC version 4.2.1 is used for converting the .c files to .s files. MAC operating system already has this version inbuilt. Alternately we can also use clang-602 version for the above conversion.

CHAPTER 7

Results of experiments

The configuration of the system used in implementing the obfuscation techniques and testing the results is as follows :

- Operating System : OS X Yosemite
- Software version : Version 10.10.5
- CPU : 2.3 GHz Intel Core i7
- RAM : 8 GB 1333 MHz DDR3
- Harddrive : 500 GB

This section describes the HMM technique which is used to test the performance of our generator. The process is to keep adding the dead code by increasing percentage value until the HMM detector begins to fail. The point at which it fails indicates a threshold value. Our experiments indicate that after inserting 30% or more dead code, the HMM detector begins to fail and the metamorphic code generator escapes this detection. We can at least conclude from the results that the code generator developed by us is really effective and better than some other metamorphism techniques discussed in the previous research [27].

7.1 Base File

We use a sample spike fuzzer as our base file [35]. Fuzzing is a method of finding bugs using malformed data injection. The process of fuzzing and the use of spike fuzzer

is also explained in [36]. The metamorphic code generator has different executable format because we have written it within the LLVM compiler. Hence standard tools like IDA-PRO should not be used for disassembling the binary. A sample disassembler should be used to disassemble the base files and the morphing files.

The standard coreutil Linux command files are used as our morphing files [33]. Once the morphing files are selected, we insert the dead code from these files and generate around 50 copies of morphed files.

The HMM scoring technique mentioned in [27] is then used to score the morphed files. The results of the classification do not change much by varying the number of hidden states. This has been already proved in the previous research [10, 27]. Therefore, for our experiments we have considered the hidden state of N = 2.

The first step of HMM is to train the model. The model should be trained on the base software. But as discussed in [3, 26], the base software considered for training should have minimal amount of morphing[10%] from the morphing files(Linux command files). The purpose of training the HMM on these slightly morphed base files is to avoid the HMM from overfitting the data which exists in the base file. 50 base files that are slightly morphed, having 10% of morphing from the 50 morphing files are generated. So, the training of HMM is carried out on these 50 slightly morphed base files. This model is referred as the ''base HMM''.

After the training phase is over, we need to score the files. Using the base HMM discussed above 50 morphing files are scored. These 50 morphing files are none other than the coreutil Linux commands.

HMM categorizes each incoming file as ''base'' or ''morphing'' based on its score. If the morphing file score is greater than the base file score, the HMM considers it as

a part of the morphing family. Whereas, if the morphing file score is not higher than the base file, then HMM buckets it as a base family file. For determining if a given file is base or morphing, threshold value comes into picture. Threshold value actually determines the category of a given file. If the score of this given file is lower than the threshold, then it is considered as a morphing file. While, the given document is thought to be a base record if its score is higher than the limit. The score of the base files should be greater than the majority of the morphing files in order to have a good threshold value. If this is the case, then it is easy to categorize the new file.

The result of the scores of 50 morphing files against 50 base (slightly morphed) is as shown in Figure 29. The scores of all the base files is higher compared to the scores of all morphing files. Hence, the base files generated having 10% of morphing are detectable by the HMM model.

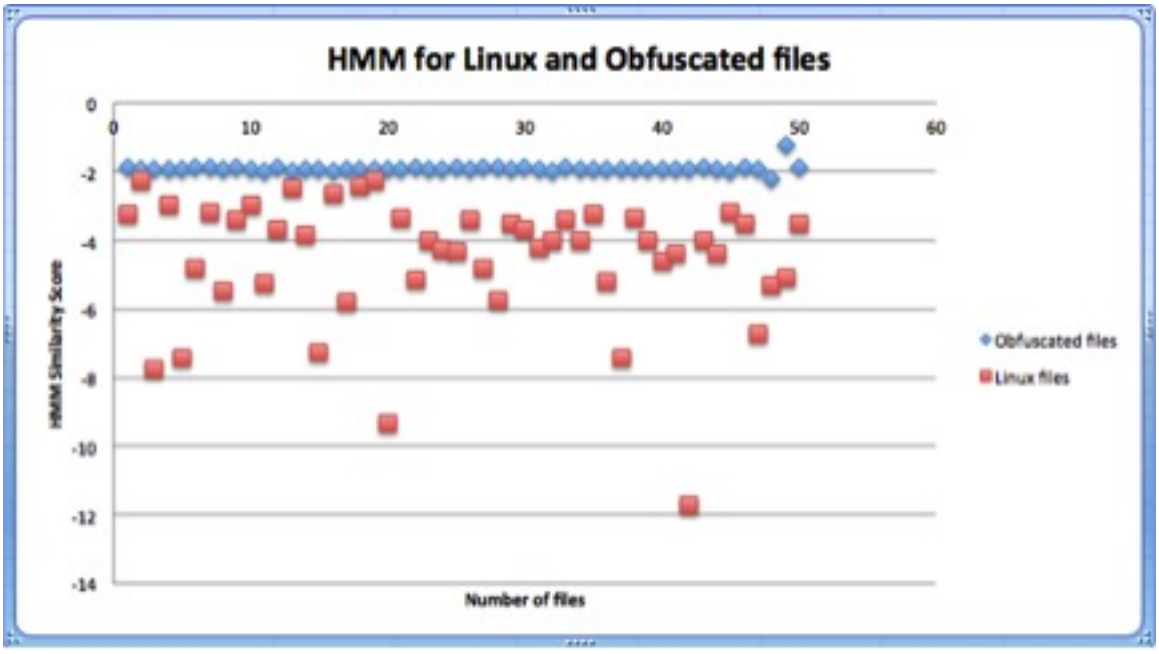

Figure 19: HMM output for obfuscated and linux utilities files

7.2 Different experiments with HMM

To find the percentage at which the HMM fails to detect the base files, we tried different combinations of morphing the base file at various percentages (20%, 30%, 40%, and 50%). At each different value of the percentage mentioned above, we produced 50 copies of base file having that percentage amount of morphing. Hence, we produced 50 copies of base morphed file at 20%, 30%, 40% and 50% respectively. Once the base morphed files are generated, we calculate the scores for these files and compare them with the scores of the morphing files(Linux utilities).The calculation of the dead code lines to be added is in approximation with the percentage of lines in the base file.

The base morphed files considered here consists of the dead code insertion technique. Hence they are the files that we get after executing the first two stages.

Below are the figures displaying the scores after adding 20%, 30%, 40% and 50% of dead code respectively. Figure 20 shows the graph displaying the scores after adding 20% dead code.

It is seen from the Figure 20, that after adding 20% of the dead code, the base morphed file's scores have become little better but majority of them have the scores falling in the range of the base files.

Figure 21 displays the HMM scores after adding 30% of the dead code to the base file. The figure indicates that the scores are better compared to 20% dead code insertion yet majority of the files have the scores similar to the base files.

The HMM scores after adding 40% and 50% dead code are represented by Figure 22 and Figure 23 respectively.

Figure 20: HMM scores having 20% dead code

Both these figures (40% and 50% dead code insertion) indicate that the scores have improved significantly compared to the previous figures (20% and 30% dead code insertion). Figure 23 indicates that after adding 50% of dead code the scores are almost in level with the morphing files.

From all the results displayed above, we can observe that after adding 30% of dead code, the base morphed file's scores start to merge with morphing files. This makes it difficult for the HMM model to differentiate between a base file and a morphing file. This is exactly what we want as our results from the code morphing technique.

All the above results are summarized in the form of ROC curves in Figure 24. ROC curves plot the false positive rate versus the true positive rate for different cut-off points of the score range. The area under the ROC cureve (AUC) is equal to

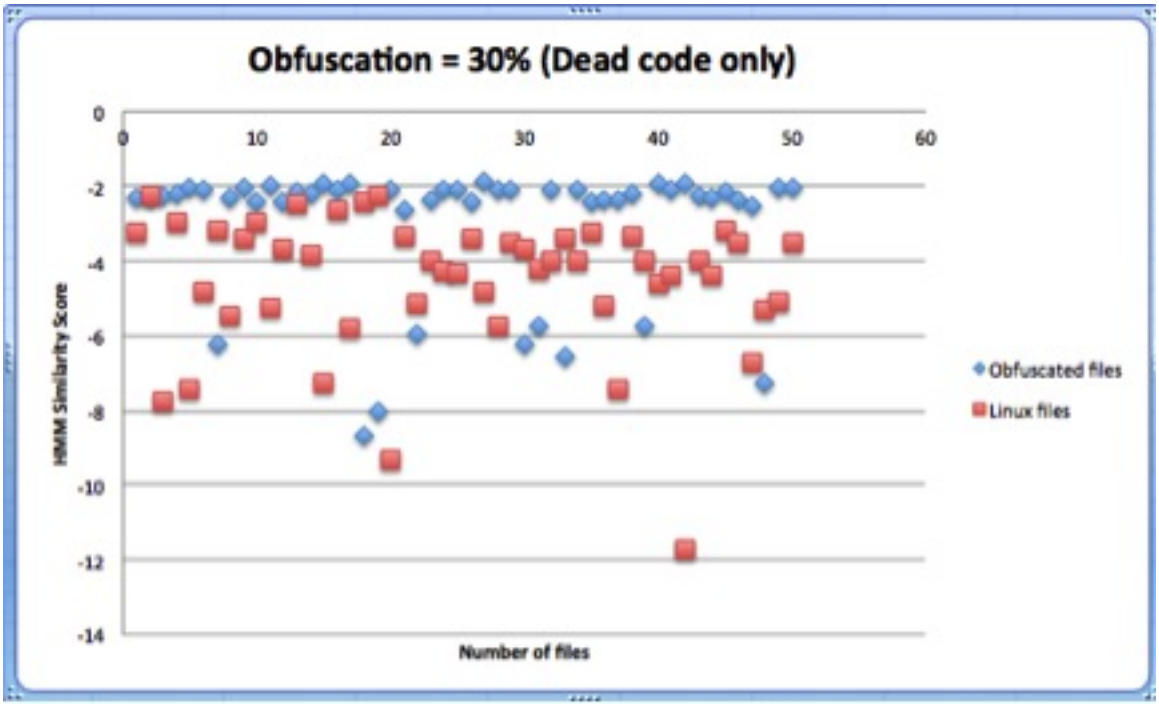

Figure 21: HMM scores having 30% dead code

Figure 22: HMM scores having 40% dead code

Figure 23: HMM scores having 50% dead code

the probability that a classifier ranks a randomly chosen positive instance higher than a randomly chosen negative value. The AUC values for the ROC curves shown in Figure 24 are given in Table 1.

Table 1: AUC values for dead code insertion

Dead code insertion range (%)	AUC
10	1.000
20	0.8828
30	0.8276
50	0.5984

Keeping the base model same, all the above tests are similarly performed for the subroutine permutation combined with dead code insertion technique. The base morphed files considered in this case would be the files having the implementation of the first three stages (dead code insertion + subroutine reordering).

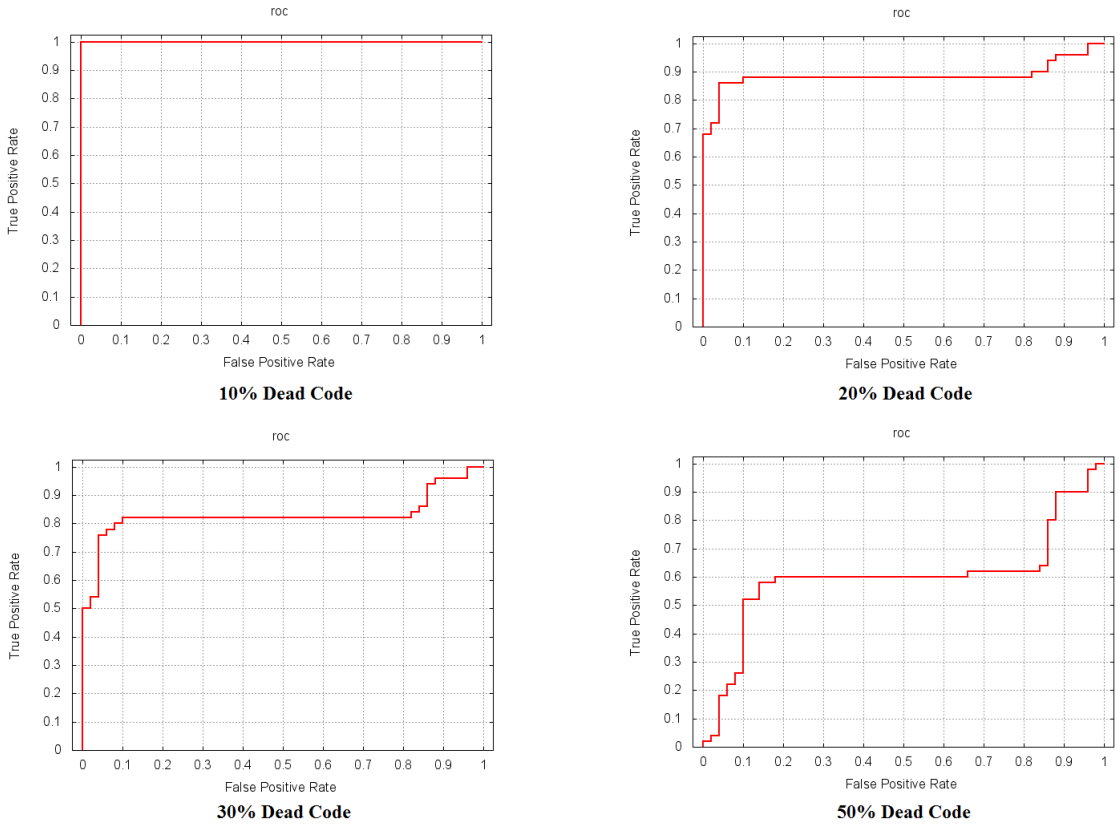

Figure 24: ROC curves for increasing rate of dead code insertion

The results of this test produce similar HMM scores compared to the previous set of tests (first 2 stages included). In fact for majority of the scores, the values are less compared to the dead code insertion technique. Graphs of this test are shown below after Figure 23. The results for the dead code insertion combined with subroutine permutation (20%,30%,40%,50%) are shown in Figure 25, Figure 26, Figure 27, Figure 28 respectively:

This shows that the dead code insertion technique executed by our code morphing generator is strong enough to escape HMM detection beyond 30% on its own and does not depend solely on function permutation technique.

The comparison of ROC AUC values for dead code insertion and the combination

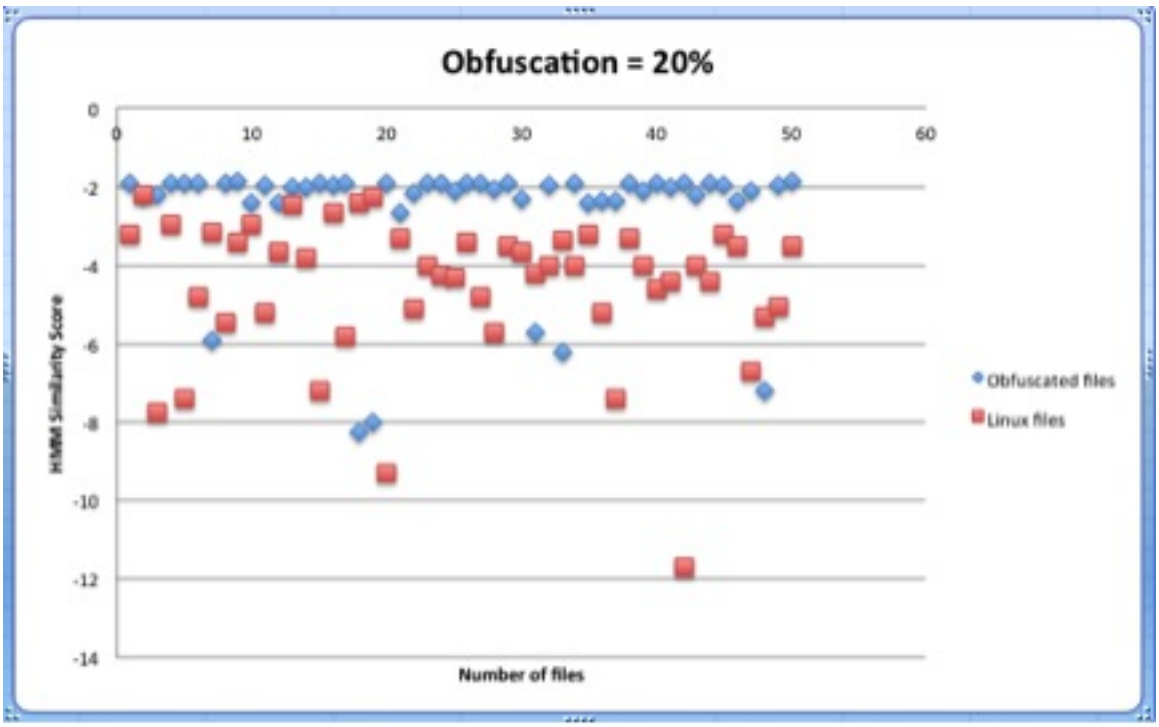

Figure 25: HMM scores with 20% dead code insertion only

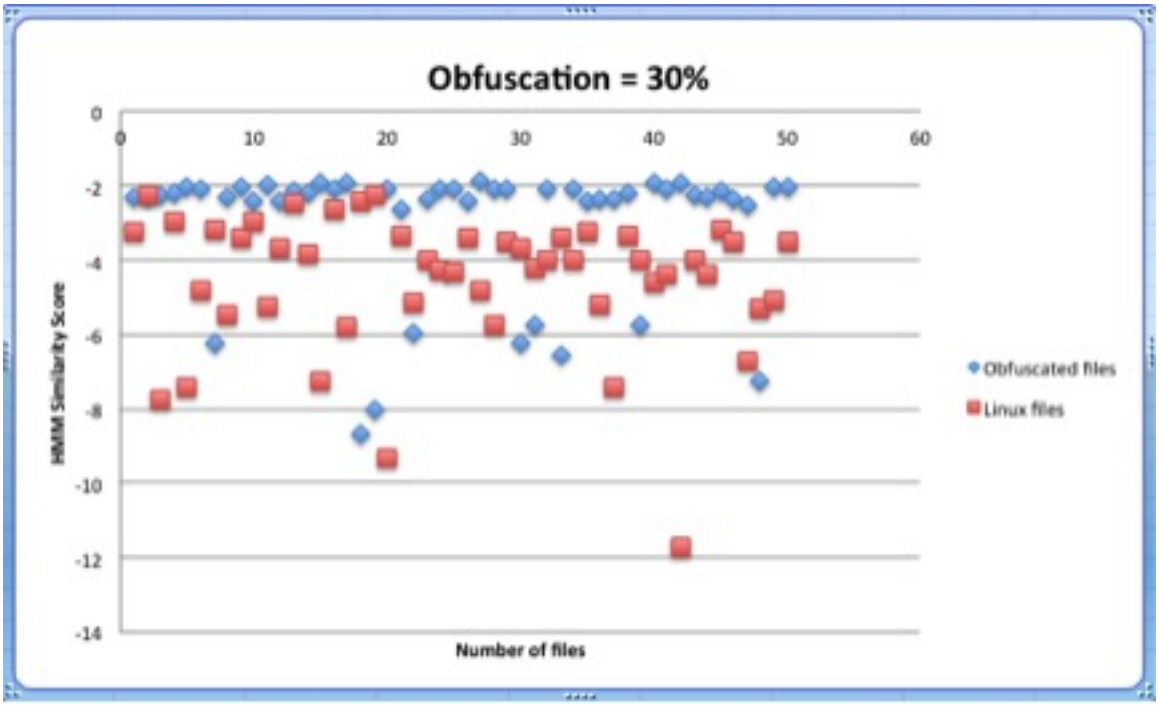

Figure 26: HMM scores with 30% dead code insertion only

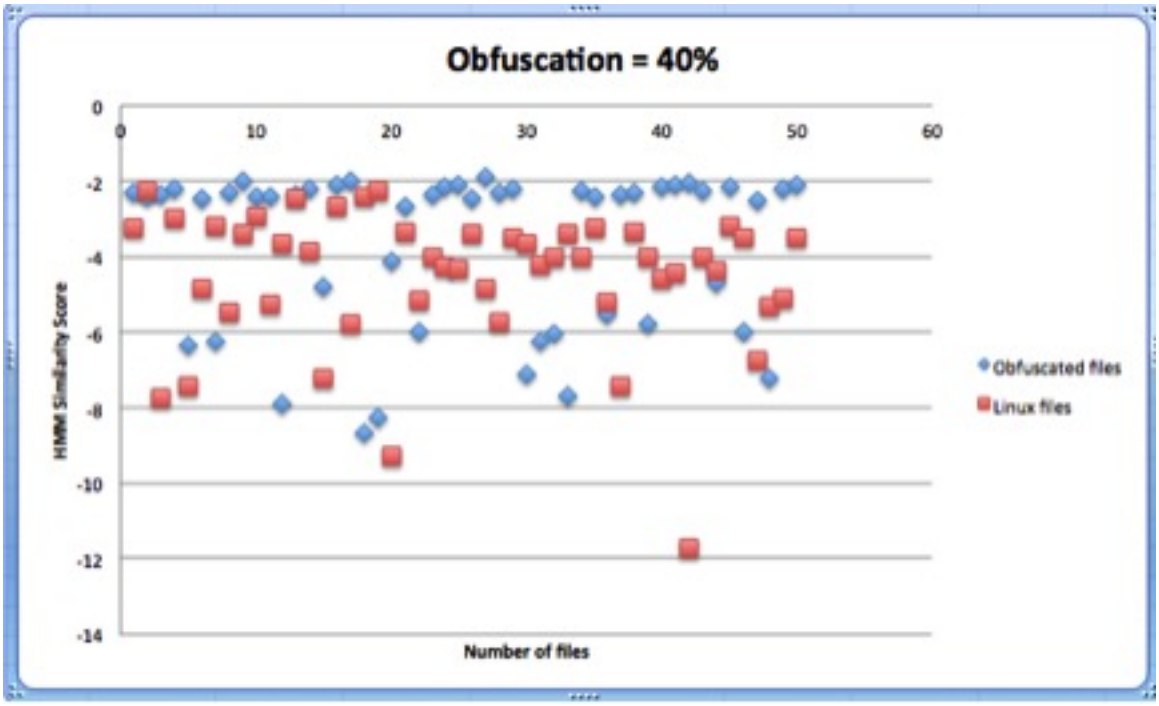

Figure 27: HMM scores with 40% dead code insertion only

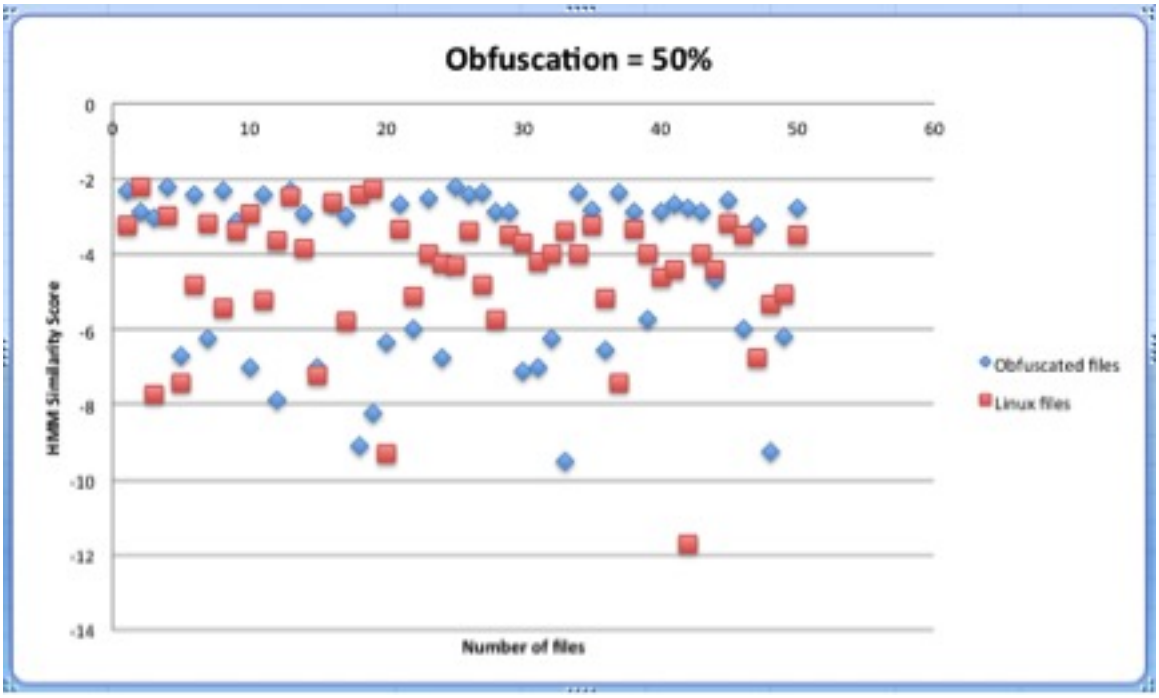

Figure 28: HMM scores with 50% dead code insertion only

of dead code and subroutine permutation is shown in Table 2. The Table 2 shows that the AUC values do not change except for 50%. Hence considering Dead code insertion technique alone is a good choice.

Table 2: Comparision of AUC values

Dead code insertion range (%)	AUC(Dead code)	AUC(Combination)
10	1.000	1.000
20	0.8828	0.8828
30	0.8276	0.8276
50	0.5984	0.5996

Further, we also tried to test the effectiveness of the Instruction substitution technique. We trained the HMM on the base file having 10% of the morphed code from the linux files. The graph for this base HMM is as shown in Figure 29. Using this trained model we tested it on the base morphed files which are now instruction substituted versus the linux utilities files. The result for this is depicted in Figure 30. It is clearly seen that the HMM is able to detect the obfuscation files(instruction substitution) from the linux files.

Hence, we then tried to combine the dead code insertion technique with the instruction substitution. We used the same base HMM model having 10% of the dead code from the linux files. We then tested it on the base morphed files having 50% of the dead code along with the instruction substitution implemented versus the linux files. We saw a slight increase in the values of the scores compared to the dead code insertion technique alone. Table 3 shows the ROC AUC values for the dead code insertion at 50% and the dead code insertion at 50% combined with the instruction substitution technique. The reason for the increase in the value is due to the frequency count of the opcodes in the base file compared to the linux files.

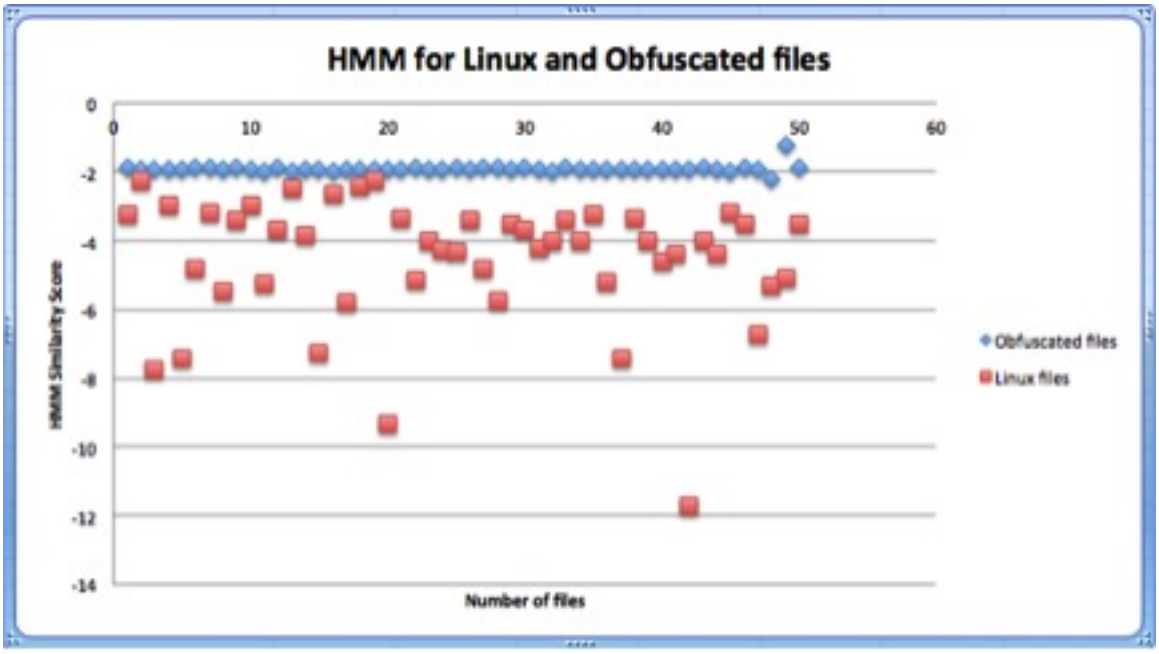

Figure 29: HMM for obfuscated and linux utilities files

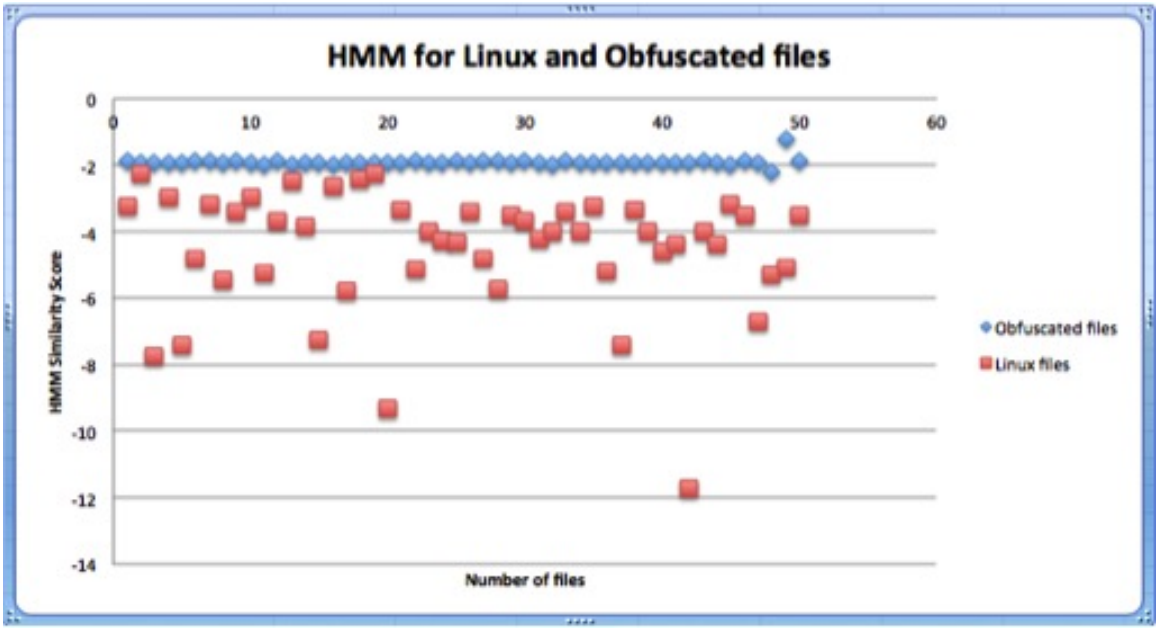

Figure 30: HMM scores for instruction substituted and linux utilities files

Analysing the opcode counts for both the files, we found that the frequency of XOR was more in base file compared to the linux files whereas the frequency of AND

instruction and OR instruction in base file was less compared to the linux files. Hence changing the XOR instruction to AND and OR in the base morphed file will balance the opcode counts for both the files.

Table 3: AUC values for dead code and combination of dead code with instruction substitution

Dead code insertion range (%)	AUC(Dead code)	AUC(Combination)
50	0.5984	0.5620

The values clearly indicates that the combination of dead code insertion technique with the instruction substitution performs better than any other technique. Hence the metamorphic generator we have developed can make use of this combination to evade HMM detection. This combination also gives us better results compared to the previous metamorphic generator developed.

CHAPTER 8

Conclusion

This paper discusses a metamorphic code morphing approach based on LLVM IR bytecode. The code morphing engine developed in this approach is made available as a compile-time option which does not require additional efforts on the part of a software developer.

The code generator developed here makes use of dead code insertion, instruction substitution and subroutine reordering obfuscation techniques. The dead code insertion basically copies functions from other programs. To make the detection of the dead code insertion difficult, these functions are called inside the program. The effectiveness of our morphing technique has been tested by using an HMM model that has proved to be efficient in malware detection and other acts of privacy. The results conclude that the code generator developed by us is effective to a great extent since the morphed code cannot be distinguished from the base code by HMM at a relatively low morphing rate. The technique that performs the best is the combination of dead code insertion along with the instruction substitution.

According to the results shown above, the base files starts looking similar to the morphing files when the percentage rate of dead code insertion increases. The Log Likelihood score per opcode of these files start looking similar.

The HMM model can detect upto 30% of dead code insertion on the data set considered. When the percentage value increases beyond 30%, the HMM begins to misclassify the files as shown in the above figures (Figure 21, Figure 22, Figure 23).

The metamorphic generator presented here can be improved in many ways.

Currently the dead code insertion is dependent on complete subroutines but there should be a way to insert the dead code at the basic block level. More powerful obfuscation technique like Register swap and Code transposition can be executed. In addition, the control of morphing handled by user at the time of compilation would make it even better. More variation techniques of instruction substitution can be added (AND, OR, etc). The inclusion of more variations in instruction substitution technique would make it even stronger.

The HMM model that we used to detect the effectiveness of our morphing generator was not able to distinguish all the virus files correctly. Hence, research in this field would really be helpful.

LIST OF REFERENCES

[1] V. Adve and C. Lattner, Architecture for a next Generation GCC. *First GCC Annual Developers Summit*, May 26, 2003,
http://llvm.org/pubs/2003-05-01-GCCSummit2003pres.pdf

[2] Prithi Desai, A highly metamorphic virus generator,Multimedia intelligence and security Vol.1, No 4, 2010,
https://www.truststc.org/pubs/779/IJMIS010407\%20STAMP.pdf

[3] Da Lin, Hunting for undetectable metamorphic viruses, 2009,
http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1017&context=etd_projects

[4] Wing Wong, Analysis and detection of metamorphic computer viruses, May, 2006,
http://www.cs.sjsu.edu/faculty/stamp/students/Report.pdf

[5] Paloalto networks, What is malware, August 2015,
https://www.paloaltonetworks.com/resources/learning-center/what-is-malware.html

[6] Computer virus, the free encyclopedia, August 2015
https://en.wikipedia.org/wiki/Computer_virus

[7] The truth about malware, List of common malware types, August 2015,
http://www.malwaretruth.com/the-list-of-malware-types/

[8] Security news, What is a computer worm, August 2015,
http://www.pctools.com/security-news/what-is-a-computer-worm/

[9] Peter Szor and Peter Ferrie, Hunting for metamorphic, Symantec Security Response, June 2015,
https://www.symantec.com/avcenter/reference/hunting.for.metamorphic.pdf

[10] D. Lin and M. Stamp. Hunting for undetectable metamorphic viruses.*Journel in Computer Virology*, 7:201--214, Aug. 2011

[11] Peter Szor, Advanced Code Evolution Techniques and Computer Virus Generator Kits, March 25, 2005,
http://www.informit.com/articles/article.aspx?p=366890&seqNum=6

[12] Panda Security (n.d.), Virus, worms, trojans and backdoors: Other harmful relatives of viruses, 2011,
http://www.pandasecurity.com/homeusers-cms3/security-info/about-malware/general-concepts/concept-9.htm?wbc_purpose=Basicdefault.htm/

[13] Sudarshan Madenur Sridhara, Metamorphic worm that carries Its own morphing engine, SJSU scholar works, Spring 2012,
http://scholarworks.sjsu.edu/etd_projects/240/

[14] Mark Stamp, A Revealing Introduction to Hidden Markov Models, SJSU, August 31, 2015 ,
http://www.cs.sjsu.edu/faculty/stamp/RUA/HMM.pdf

[15] Ilsun You and Kangbin Yim, Malware Obfuscation Techniques: A Brief Survey, 2010,
http://isyou.hosting.paran.com/papers/bwcca10-2.pdf

[16] Arini Balakrishnan and Chloe Schulze, Code Obfuscation Literature Survey, December 19th, 2005,
http://pages.cs.wisc.edu/~arinib/writeup.pdf

[17] Lenny Zeltser, How antivirus software works: Virus detection techniques, July 2015,
http://searchsecurity.techtarget.com/tip/
How-antivirus-software-works-Virus-detection-techniques

[18] W. Wong and M. Stamp, Hunting for Metamorphic Engines.*Journal in Computer Virology,* vol. 2, no. 3, pp. 211-229, Dec. 2006

[19] E. Konstantinou, Metamorphic Virus: Analysis and Detection, RHUL-MA-2008-02.*Technical Report of University of London*, Jan. 2008

[20] Evgenios Konstantinou and Stephen Wolthusen, Metamorphic Virus: Analysis and Detection, July 2015
http://cdn.ttgtmedia.com/searchSecurityUK/downloads/RH5_Evgenios.pdf

[21] Virus Construction Kits, May 2015,
http://computervirus.uw.hu/ch07lev1sec7.html

[22] Taylor Thomas, What is Heuristic Antivirus Detection, May 22, 2009
http://internet-security-suite-review.toptenreviews.com/
premium-security-suites/what-is-heuristic-antivirus-detection.html

[23] Chris Lattner, The Architecture Of Open Source Applications, Elegance, Evolution and a few fearless hacks, May 2015,
http://www.aosabook.org/en/llvm.html

[24] Jakob Praher, A Change Framework based on the Low Level Virtual Machine Compiler Infrastructure, April 2007,
http://llvm.org/pubs/2007-04-PraherMSThesis.pdf

[25] Srilatha Attaluri, Scott McGhee, and Mark Stamp , Profile Hidden Markov Models and Metamorphic Virus Detection, May 2015
http://www.cs.sjsu.edu/faculty/stamp/papers/profileHMM4.pdf

[26] Shabana Kazi and Mark Stamp, Hidden Markov Models for Software Piracy Detection, SJSU , September 2015,
http://www.cs.sjsu.edu/faculty/stamp/papers/topics/topic10/shabana.pdf

[27] Wing Wong and Mark Stamp, Hunting for metamorphic engines, November 2006,
https://vxheaven.org/lib/pdf/Hunting\%20for\%20metamorphic\%20engines.pdf

[28] Coreutils source code, September 2015,
http://ftp.gnu.org/gnu/coreutils/

[29] LLVM Compiler Infrastructure, LLVM Command Guide, June 2015,
http://llvm.org/docs/CommandGuide/

[30] Adam Berger, Statistical machine learning for information retrieval, Carnegie Mellon University, April, 2001,
http://www.cs.cmu.edu/~aberger/pdf/thesis.pdf

[31] Virus Creation Tools : VX Heavens, August 2015,
http://oktridarmadi.blogspot.com/2009/09/virus-creation-tools-vx-heavens.html

[32] Teja Tamboli, Metamorphic Code Generation from LLVM IR Bytecode (2013),
http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1305&context=etd_projects

[33] LLVM source code, August 2015,
http://llvm.org/releases/download.html

[34] W. Ma, P. Duan, S. Liu, G. Gu and J. Liu. Shadow Attacks: Automatically evading system-call behavior, July, 2015,
http://faculty.cs.tamu.edu/guofei/paper/ShadowAttacks_final-onecolumn.pdf

[35] Spize fuzzer source code,
http://www.immunitysec.com/resources-freesoftware.shtml

[36] Introduction to spike fuzzer,
http://resources.infosecinstitute.com/intro-to-fuzzing/