San Jose State University

SJSU ScholarWorks

Master's Projects Master's Theses and Graduate Research

Spring 2012

HIDDEN MARKOV MODELS FOR SOFTWARE PIRACY DETECTION

Shabana Kazi
San Jose State University

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

Part of the Computer Sciences Commons

Recommended Citation

Kazi, Shabana, "HIDDEN MARKOV MODELS FOR SOFTWARE PIRACY DETECTION" (2012). *Master's Projects*. 236.
DOI: https://doi.org/10.31979/etd.srkx-n4pf
https://scholarworks.sjsu.edu/etd_projects/236

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.

HIDDEN MARKOV MODELS FOR SOFTWARE PIRACY DETECTION

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

Shabana Kazi

May 2012

© 2012

Shabana Kazi

ALL RIGHTS RESERVED

The Designated Thesis Committee Approves the Thesis Titled

HIDDEN MARKOV MODELS FOR SOFTWARE PIRACY DETECTION

by

Shabana Kazi

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2012

Dr. Mark Stamp	Department of Computer Science
Dr. Chris Pollett	Department of Computer Science
Dr. Sami Khuri	Department of Computer Science

ABSTRACT

Hidden Markov Models for Software Piracy Detection

by Shabana Kazi

The unauthorized copying of software is often referred to as software piracy. Software piracy causes billions of dollars of annual losses for companies and governments worldwide.

In this project, we analyze a method for detecting software piracy. A metamorphic generator is used to create morphed copies of a base piece of software. A hidden Markov Model is trained on the opcode sequences extracted from these morphed copies. The trained model is then used to score suspect software to determine its similarity to the base software. A high score indicates that the suspect software may be a modified version of the base software and, therefore, further investigation is warranted. In contrast, a low score indicates that the suspect software differs significantly from the base software. We show that our approach is robust, in the sense that the base software must be extensively modified before it is not detected.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Mark Stamp for giving me a chance to work on this topic. I would also like to thank him for his valuable guidance. I would like to thank my committee members Dr. Sami Khuri and Dr. Chris Pollett for their feedback and advice. In addition, I would like to thank my parents for their love, support and guidance. I would also like to thank my sister for her advice and support. Special thanks to my husband for his unending moral and emotional support throughout my Masters.

TABLE OF CONTENTS

CHAPTER

1 Introduction 1

2 Background 4

2.1 Metamorphic Software 4

2.1.1 Metamorphic Techniques 4

2.2 Hidden Markov Model 7

3 Design Overview 9

3.1 Training 9

3.2 Detection 9

3.3 Design of Metamorphic Generator 10

4 Implementation 14

4.1 Metamorphic Generator 14

4.2 Training the HMM 16

5 Experiments 18

5.1 Experiment 1 18

6 Results and Observations 23

6.1 Experiment 1: Training by morphing with 1 block 23

6.2 Experiment 6: Training by morphing with 32 blocks 24

7 Conclusion 27

8 Future work 28

APPENDIX **Page**

A Programs used 32

B Pseudocode for morph.java 33

C Experiment Results 34

C.0.1 Experiment 2: Training by morphing with 2 blocks 34

C.0.2 Experiment 3: Training by morphing with 4 blocks 34

C.0.3 Experiment 4: Training by morphing with 8 blocks 34

C.0.4 Experiment 5: Training by morphing with 16 blocks 35

LIST OF TABLES

Table **Page**

1 Example of Dead Code Insertion 5

2 Example of Instruction Replacement 7

LIST OF FIGURES

Figure **Page**

1 Permutation Technique. 6

2 Illustration of insertion of JMP statements [25] 6

3 A Hidden Markov Model 8

4 Training Phase [13] 10

5 Detection Phase 10

6 Metamorphic Generator 12

7 Inserting Blocks 13

8 Training of HMM 17

9 Tampered files generation for Experiment 1 20

10 Detection rate average calculation 21

11 Representation of all experiments 22

12 Experiment 1 Detection Results 24

13 Experiment 6 Detection Results 25

14 Results Summary 26

CHAPTER 1

Introduction

Software piracy is referred to as the unauthorized use of software [19]. It also includes the illegal copying of copyrighted software or the installation of copyrighted software on more computers than permitted under the terms of the software license agreement [19].

Business Software Alliance (BSA) is a leading advocate for the global software industry. BSA serves as the worlds premier anti-piracy organization. According to the 2010 BSA Global Software Piracy Study, the commercial value of software piracy grew 14 percent globally in 2010 to a record high total of $58.8 billion This amount has almost doubled since 2003 [14]. These funds could have been used to spur new jobs or innovation by the software companies. Thus in turn, the software industry incurs huge losses. For every dollar of PC software sold, around $3 to $4 is lost as revenues [20].

Users using pirated copies of software are also vulnerable to virus attacks. Security threats like Trojans, spyware, worms and viruses are built to exploit the vulnerabilities in any software products. This forces software developers to release patches and fixes to counter the emerging malware issues. Consumers using pirated, unlicensed products are usually unable to benefit from the patches and important updates, which would keep their systems, secure [20]. Eventually, this leads to these consumers being more predisposed to attacks over long term. There are other disadvantages of using software piracy. Consumers do not receive technical support, documentation or manuals [20]. Through pirated versions, users can get incomplete

or trial versions of the software [20]. Moreover, a user's computer could be infected with viruses for remote-controlled cyber crime [20].

The goal of this project is to develop and test a tool that can be used to detect pirated software. Our technique can be used if a company suspects that their copyrighted software has been illegally copied. Using a novel technique based on hidden Markov models, the original software is scored against the suspected pirated copy. A high score indicates that further investigation is warranted, while a low score indicates that the two pieces of software are almost certainly distinct. Scores can be computed after the original software has been distributed and no special effort is required during the software development process. Our scoring technique uses executable files only, i.e., no source code is required. In addition, our technique relies only on statistical analysis neither the original nor the suspect code is executed and the scoring technique is fast and efficient. Extensive experimental results provided in this paper indicate that our approach is robust, in the sense that the original software must be extensively modified before we are unable to detect a high degree of similarity to the original code.

We would like to highlight here that our technique differs from the plagiarism detection techniques. When a possible plagiarized document is compared against a registered document, information retrieval techniques are employed until two paragraphs are found which are related highly semantically. The paragraphs are compared minutely, on a per-sentence basis, in order to find out if the paragraphs have common sequences of words [18] [11]. In order to detect plagiarism in short computer programming homework, students writing code for the same assignment may have many common sequences. This is because all the students work on the same homework programming problem [3]. Thus, our technique cannot be used for plagiarism detection,

since most of the students' programs will score high in our detection technique even if they have not copied from each other.

Our technique has been inspired by previous research work done on detection of metamorphic viruses [26]. It is difficult to detect metamorphic viruses. A signature-based scanner may not be able to detect viral code even if there are small changes. Moreover, the signature database needs to be constantly updated to detect newly morphed variations. In this research [26], the Hidden Markov Model (HMM) was trained using viruses and the trained models were able to classify a particular virus family from non-viral programs [26]. In further research work that was done, the HMM was trained to detect a specific copy of software. The experimental results proved that even after the original software is extensively modified; it could be identified [13].

The report is organized as follows. Section 2 discusses background material on metamorphic software and hidden Markov models. In Section 3, we provide an overview of the design of our piracy detection technique. Section 4 contains details on how our approach has been implemented. Experiments are explained in Section 5. Section 6 discusses the results and observations. Finally, Sections 6 and 7 contain our conclusions and a discussion of future work, respectively.

CHAPTER 2

Background

2.1 Metamorphic Software

Metamorphism is used for changing a piece of code into copies that functionally perform the same task, but differ structurally. This method has been used by virus and malware writers to create viruses that would go undetected by the anti-virus detection programs [1].

Metamorphism has positive applications as well. It can also be used to increase the diversity of software [22]. In the paper [23], an analogy has been drawn between software and a biological system. If a biological system is attacked by a disease, a large percentage of population survives [23]. This is partly because of the genetic variety of the population. However, software tends toward a monoculture. Due to this, an attack which is successful on a piece of software, succeeds on every other instance of the software [6]. In metamorphic software, the same attack will not be successful on the different copies of the software [6].

2.1.1 Metamorphic Techniques

This section describes some common techniques of generating metamorphic code. These techniques are discussed below.

2.1.1.1 Garbage Code Insertion (Dead Code Insertion)

The simplest method of morphing used by a metamorphic engine is to change the byte sequence of code by inserting dead code [16]. The inserted instructions do not have any effect on the functionality of the program [9]. Garbage (dead) code is

equivalent to a null operation. The code inserted is never executed, so there is no semantic effect on the software [2]. An example of dead code insertion appears in Table 1.

Table 1: Example of Dead Code Insertion

Original Code		Transformed Code		
mov	eax, 1034h		mov	eax, 1034h
sub	eax		jmp	loc
			push	ebp
			pop	ebp
			sub	esp, 18h
		loc:	sub	eax,1

2.1.1.2 Permutation Techniques

In this technique, the metamorphosis is carried out by dividing the code into frames. Then the frames are positioned at random and are connected by branch instructions. This is done in order to maintain the flow of the process. The branch instructions can be jump statements [5]. Figure 1 illustrates the Permutation Technique.

2.1.1.3 Insertion of Jump Instructions

JMP is an assembly language instruction, which carries out an unconditional jump. It takes a memory address as an argument, which is a label in assembly language [16]. The JMP instruction is used to change the targeted instruction address. However, the flow of the program does not change [16]. Figure 2 illustrates the insertion of Jump instructions.

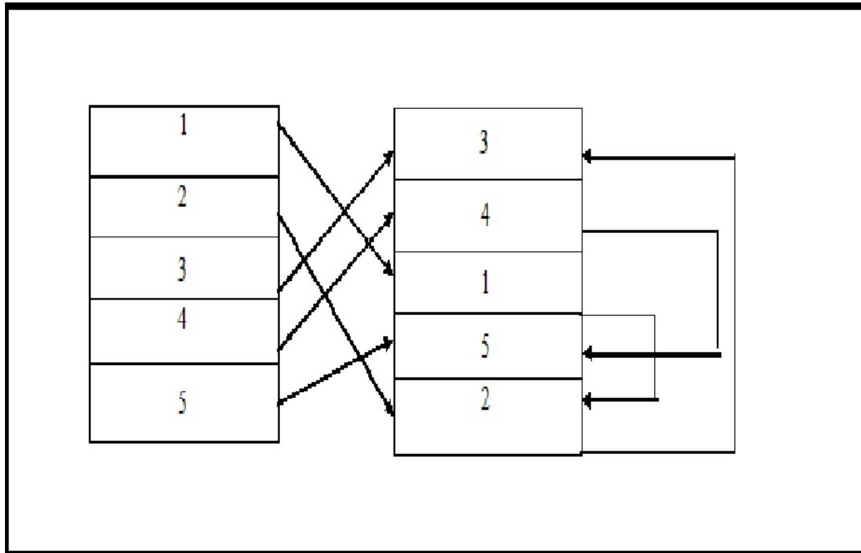

Figure 1: Permutation Technique.

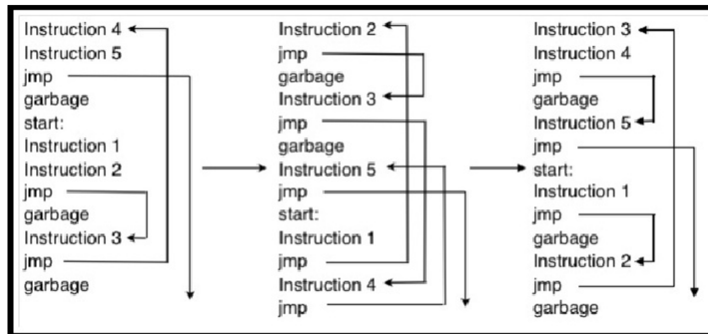

Figure 2: Illustration of insertion of JMP statements [25]

2.1.1.4 Instruction Replacement

In this technique, a particular instruction is replaced by an equivalent instruction or a set of instructions are replaced by an equivalent set of instructions. For example, the instruction xor eax, eax can be replaced with the instruction sub eax, eax [9]. Table 2 illustrates an example.

Table 2: Example of Instruction Replacement

Original Code		Transformed Code	
add	eax, 05H	add	eax, 04H
mov	al, bl	add	eax, 01H
		push	al
		pop	bl

2.2 Hidden Markov Model

A statistical model that has states and known probabilities of the state transitions is called a Markov model [21]. In such a Markov model, the states are visible to the observer. In contrast, a hidden Markov model (HMM) has states that are not directly observable [15]. A hidden Markov model consists of state transition probabilities, a probability distribution for all possible output symbols for each state, and initial state probabilities [21]. A hidden Markov model is a machine learning technique, which uses a discrete hill climb technique [21]. HMMs have been successfully used in speech recognition [17], malware detection [12] and a variety of other problems. It is used because of its efficient algorithm.

The following notation can be used to describe an HMM [21]:

$T \rightarrow$ Length of the observation sequence

$N \rightarrow$ Number of states in the model

$M \rightarrow$ Number of observation symbols

$Q = \{q_0, q_1, \ldots, q_{N-1}\} \rightarrow$ Number of observation symbols

$V = \{0, 1, \ldots, M-1\} \rightarrow$ Set of possible observations

$A \rightarrow$ state transition probabilities

$B \rightarrow$ observation probability matrix

$\pi \rightarrow$ initial state distribution

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

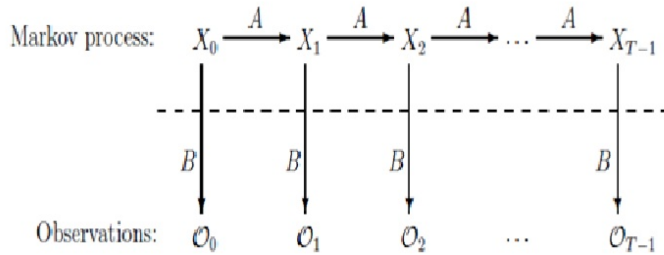

Figure 3: A Hidden Markov Model

A hidden Markov model is defined by the matrices A, B and π. Therefore, we denote an HMM as $\lambda = (A, B, \pi)$. Figure 3 represents a hidden Markov model. The following three problems can be solved using the efficient algorithms of HMM:

Problem 1: Given a model $\lambda = (A, B, \pi)$ and an observation sequence O, we need to find $P(O|\lambda)$. That is, an observation sequence that can be scored to see how well it fits a given model [21].

Problem 2: Given $\lambda = (A, B, \pi)$ and O, we can determine an optimal state sequence for the Markov model. That is, the most likely hidden state sequence can be uncovered [21].

Problem 3: Given O, N, and M, we can find a model λ that maximizes probability of O. This is the training of a model in order to best fit an observation sequence [21].

In this project, the algorithms for Problems 1 and 3 are used. First, we train a model based on a given base piece of software (Problem 3). Then the model obtained can be used to score any other piece of code against the model (Problem 1). A high score would indicate a high degree of similarity with the base code. The next section discusses the overall design overview.

CHAPTER 3

Design Overview

The aim of our project is to design and build a robust software piracy detecting method. We consider a scenario where we suspect that our software has been stolen and modified to avoid being detected. Our goal is to be able to detect that the suspected piece of software has been copied from our software.

Our design has been inspired by research done in previous studies [13]. The system mainly comprises of two main phases. They are the training and detection phases. In the training phase, a hidden Markov model is trained using morphed copies of the base software. In the detection phase, we test the closeness of a piece of software to the original base software.

3.1 Training

In this phase, slightly morphed copies of the base software are created. The opcode sequences from these morphed copies are extracted and appended. A hidden Markov model is trained using the extracted sequence. Morphed copies of the base software is used to avoid having the HMM overfit the training data [24]. The training phase is represented in Figure 4.

3.2 Detection

In this phase, the opcode sequence from a given piece of suspected software is extracted. The sequence is scored against the trained HMM, which was generated in the previous phase. A high score would signify that the suspected software is similar to the original software. A low score would signify that the suspected software is not

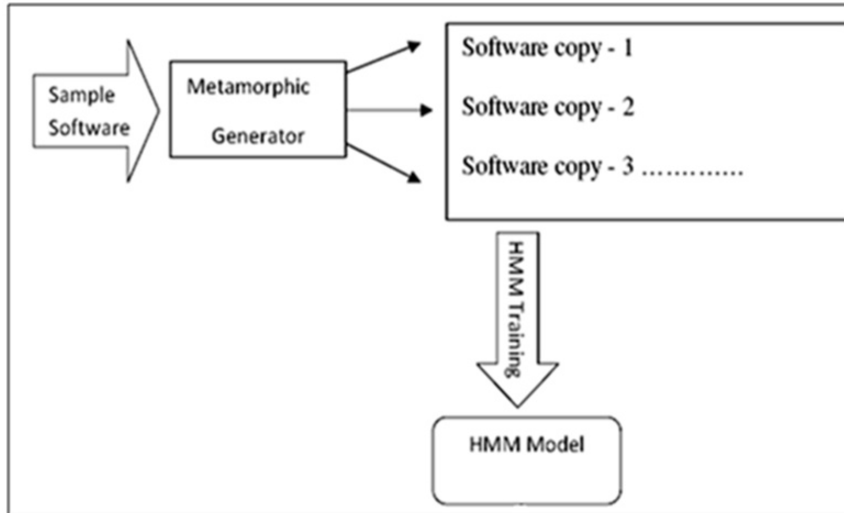

Figure 4: Training Phase [13]

similar to the original software [24]. Figure 5 illustrates the detection phase.

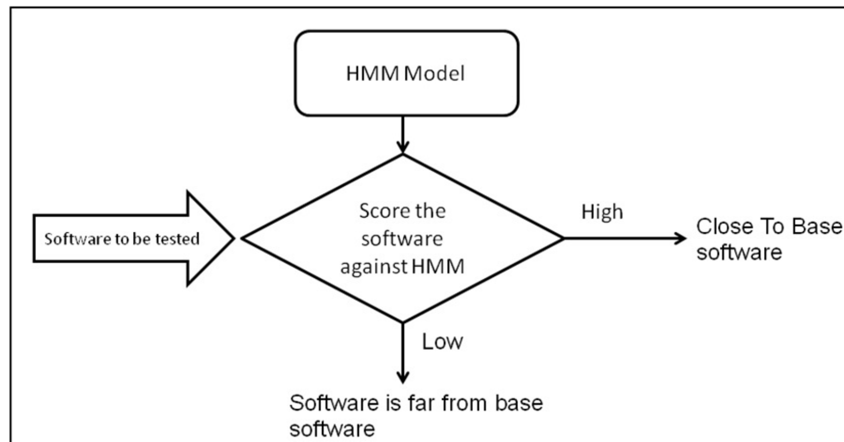

Figure 5: Detection Phase

3.3 Design of Metamorphic Generator

The design of the metamorphic generator was inspired by the research done in [13]. Our metamorphic generator produces morphed copies from a given piece of software. As discussed in Section 2.1.1, there are a number of metamorphic techniques that can be used to generate metamorphic software. In this project, we have used

only dead code insertion. We take the disassembled code of the original base software and apply morphing to it. IDA Pro DisAssembler 5.0 [7] was used to generate the disassembled file.

In our experiments, we have considered percentage of morphing/tampering. The tampering percentage indicates the number of assembly code instructions that is inserted from some other program into the base program. The tampering percentage is a metric in our experiment, by which we can measure the success of our detection rate. We try to simulate how an attacker would attack a piece of software by using the tampering percentage. The best way we could increase the size of our base program, and measure the detection rate against the tampering percentage, is by using dead code insertion. Moreover, with dead code insertion, the control and data flow of the software is not affected.

There are four parameters that are used for the functioning of our metamorphic generator. The parameters are the file that needs to be morphed; the percentage of morphing that is to be done, the number of blocks that need to be inserted into the assembly code and finally the number of morphed copies of the file that need to be generated. The total number of assembly code instructions that need to be inserted into base software are divided into blocks.

For example, let us consider a file xyz.asm to be morphed. Suppose the file has 100 assembly code instruction lines. We consider all the parameters as amount of morphing to be done as 20%, the number of blocks to be inserted as 4 and the number of morphed copies to be generated as 100. Therefore, each morphed copy will comprise of 120 lines. The 20 lines that will be inserted into the xyz.asm are divided into 4 blocks and each block comprises of 5 lines. The 4 blocks are equally distributed into xyz.asm. So, in this case, each block is inserted after every 25 lines

of the file xyz.asm. Finally, 100 morphed copies of xyz.asm are generated. Figure 6 is a representation of a metamorphic generator. The pictographic representation of the morphing of xyz.asm is given in Figure 7.

The next section discusses the implementation of the metamorphic generator and training of the HMM.

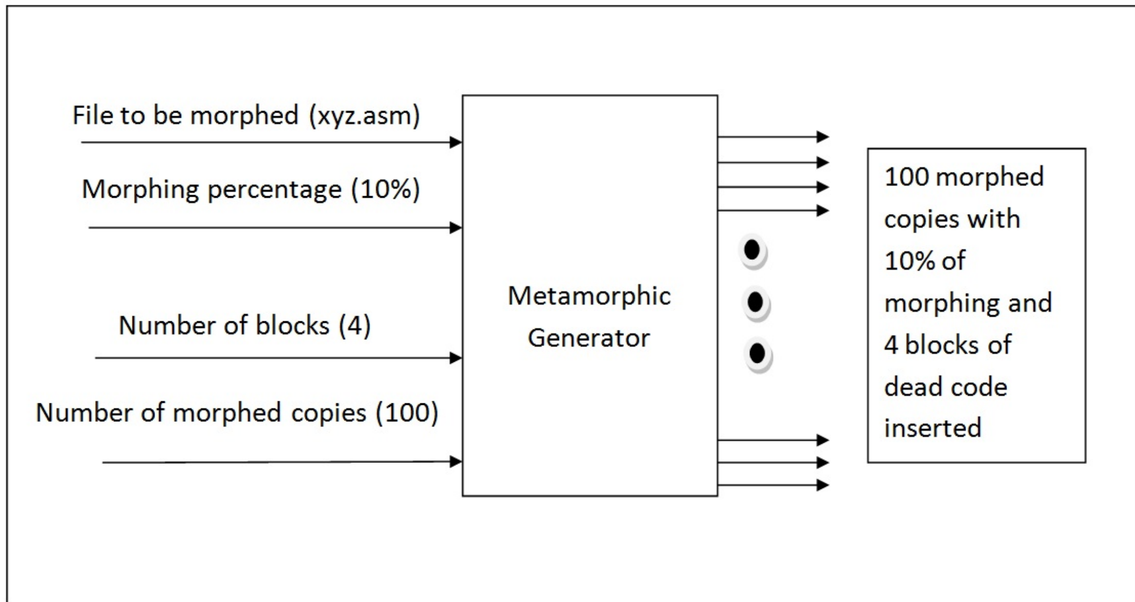

Figure 6: Metamorphic Generator

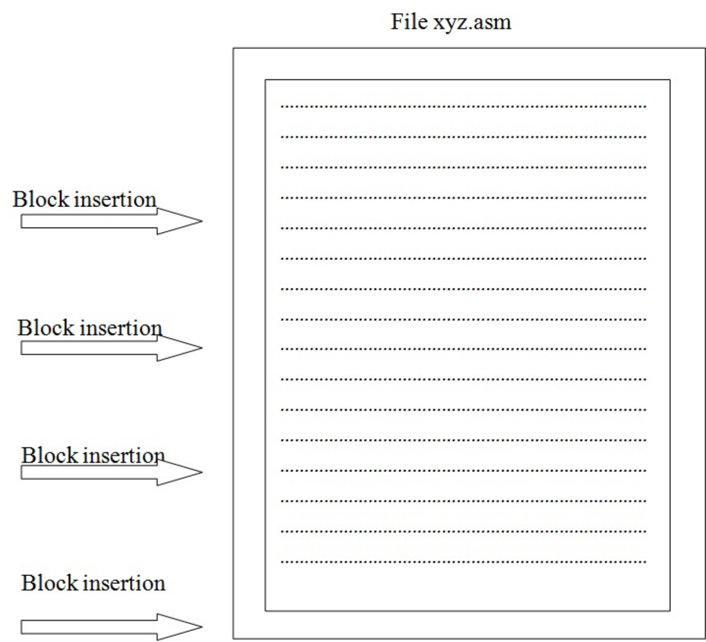

Figure 7: Inserting Blocks

CHAPTER 4

Implementation

In this section, we discuss how the metamorphic generator has been implemented and how the training of the HMM has been conducted. The training of HMM has been inspired by the work done in [26]. The metamorphic generator is used to produce morphed and "tampered" copies of the original software. The HMM is trained using these morphed copies.

4.1 Metamorphic Generator

The metamorphic generator was implemented using Java. The program is named morph.java. In this generator, dead code insertion technique was used. This was done, so that the control and data flow of the software is not changed. The parameters that are given to the program are the morphing percentage, the number of blocks that will be inserted into the base file and the number of morphed files that need to be generated. The final output obtained is morphed files. In all the experiments, 10% of morphing is used. The program inserts dead code into the specified base file from other files. The number of assembly line instructions that need to be inserted into the base file from other files is calculated using the percentage of morphing. The starting points of the lines that are selected from the other file for insertion are selected randomly. Let us consider an example.

Let the disassembled file that needs to be morphed be base.asm. We consider another disassembled file called deadcode.asm. Lines from deadcode.asm will be inserted into base.asm for morphing. Let the number of lines in base.asm be 1000 lines. We consider 10% of morphing and 4 blocks of insertion. Therefore, totally 100

lines will be inserted into base.asm. Also the 100 lines from deadcode.asm will be divided into 4 blocks of 25 lines each. The four blocks will be distributed evenly in the file base.asm. For the first block, a random line is selected in deadcode.asm. Then, the next consecutive 25 lines from deadcode.asm are selected for inserting into base.asm. The first block is inserted after 250 lines of base.asm. The similar process takes place for the second and third block insertion. Finally, the fourth block is inserted at the end of base.asm file. The pseudo code for morph.java is in Appendix B.

We have also implemented another metamorphic generator using Java. This program is called tamper.java. This program was used to test the robustness of our approach. We use the term "tampering" instead of morphing for the detection phase. We tamper the base file by inserting dead code from other files. The amount of dead code to be inserted is determined by the percentage of tampering. This program uses the similar logic as morph.java. It generates tampered (morphed) files by inserting 1 block, 2 blocks, 4 blocks, 8 blocks, 16 blocks, and 32 blocks into the base file. For each block insertion, tampering percentage of 10% to 100% is considered. A hundred files are generated for each tampering percentage. The process of selection and insertion of lines from other files is similar to the process followed in morph.java.

Let us consider an example for tampering. We consider a disassembled file called deadcode2.asm. Suppose the file base.asm has 1000 assembly code instruction lines. We consider 60% of tampering and 4 blocks of insertion. Therefore, totally 600 lines will be inserted into base.asm. Also the 600 lines from deadcode2.asm will be divided into 4 blocks of 150 lines each. The four blocks will be distributed evenly in the file base.asm. For the first block, a random line is selected in deadcode2.asm. Then, the next consecutive 150 lines from deadcode2.asm are selected for inserting into base.asm. The first block is inserted after 250 lines of base.asm. The similar

process takes place for the second and third block insertion. Finally, the fourth block is inserted at the end of base.asm file.

4.2 Training the HMM

The morphed copies that are generated using the metamorphic generator are used as dataset for training the HMM [26]. The opcode sequences are extracted from each morphed copy and concatenated to obtain a long observation sequence [13] [26]. The cross-validation technique was used for the process of training [8].

A five-fold cross-validation was used in the experiments for this project. In order to train a model, one of the subsets is selected as test data. The remaining four subsets are selected as training data for HMM [13]. The test data along with the other normal files are used for scoring. This process is done over five iterations. In each iteration, the test data subset and training data subset is altered [13] [26].

We observe that the test data files score high. On the other hand, normal files score low. The threshold is set by considering the highest scores of the low scoring files (normal files) [13]. The reason why this happens is because the test data is similar to the training files that have been used to train the hidden Markov model. Since the normal files, that have been selected to set the threshold, are different from the training set files, therefore they score low.

During the detection phase, a given piece of software is scored against the trained model. If the score is above the set threshold, then the piece of software is similar to the original software. If the score is lower than the threshold, then this means that the piece of software is not similar to the base software [24]. Figure 8 illustrates the training phase.

The next section discusses the experiments conducted.

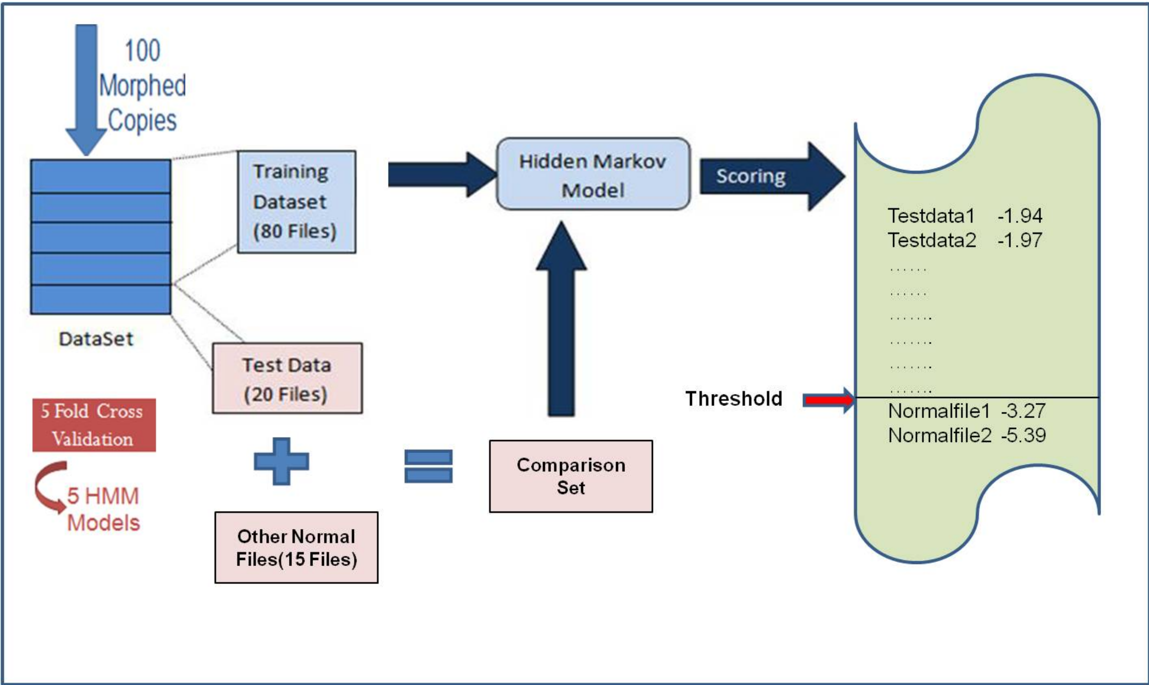

Figure 8: Training of HMM

CHAPTER 5

Experiments

We carried out a number of experiments to test the robustness of our approach. In each experiment, we considered 10 Cygwin utility files as base files. The sizes of the files were within the range of 80 to 110 KB. IDA Pro DisAssembler 5.0 [7] was used to generate the disassembled file. We will describe below how the first experiment was conducted.

5.1 Experiment 1

We considered a Cygwin utility file as the base file. We used the metamorphic generator to generate 100 morphed copies of the base file. Morphing was done by considering slight morphing percent of 10% and by inserting 1 block of dead code. The morphing and block insertion was done as explained in Section 3.3. The hidden Markov model was trained using the 100 morphed copies. A five-fold cross validation technique was applied. Eighty files from the morphed copies were used to train the model and the 20 files were used to test. The normal files that were used as the test data to set the threshold for the experiment consisted of 15 executables from Cygwin version 1.5.19 [26]; see Figure 8. Morphed copies of the base software are used to avoid having the HMM overfit the training data [24].

In order to determine the robustness of the approach, the base software was tampered for detection phase. The insertion of code from other files was done in order to make the tampered files look more like other files. The reason for doing this is if someone steals and copies the base software, they will tend to make the original software look more like some other program rather than the original program.

There were a number of parameters considered for tampering. First was the number of blocks of dead code that were inserted into the base file. For each block insertion, 10% to 100% of tampering was carried out. For each tampering percentage, 100 tampered files were generated. Blocks of 1, 2, 4, 8, 16 and 32 were inserted into the base file. In total, there were 6000 tampered files that were generated for the first Cygwin base file. These tampered files were scored against the trained model as described in Section 3.3. The HMM based similarity scores were generated and we developed a script to print the number of files that scored above the threshold for each of the 100 files. Figure 9 illustrates the representation of the generation of the tampered files and scoring against the trained model.

This experiment was repeated nine more times by considering nine more different base files. Totally 60,000 tampered files were generated for the ten different base files. The number of files that score above the threshold were averaged for corresponding block and percentage of tampering. For example, for all 10 Cygwin files for 1 block of 10% tampering, the number of files that scored above the threshold is averaged. This gives the detection rate. Figure 10 illustrates how the average is calculated. This process was carried out for all the percentage of tampering for each corresponding block of tampering.

In experiments 2, 3, 4, 5 and 6, the HMM was trained by inserting 2, 4, 8, 16, 32 blocks respectively. For each experiment, tampering was carried out by inserting 1, 2, 4, 8, 16 and 32 blocks of code and for each block tampering, 10% to 100% of tampering was considered. The approach followed in the experiments was similar to the experiment procedure carried out in Experiment 1. Figure 11 represents all the experiments carried out.

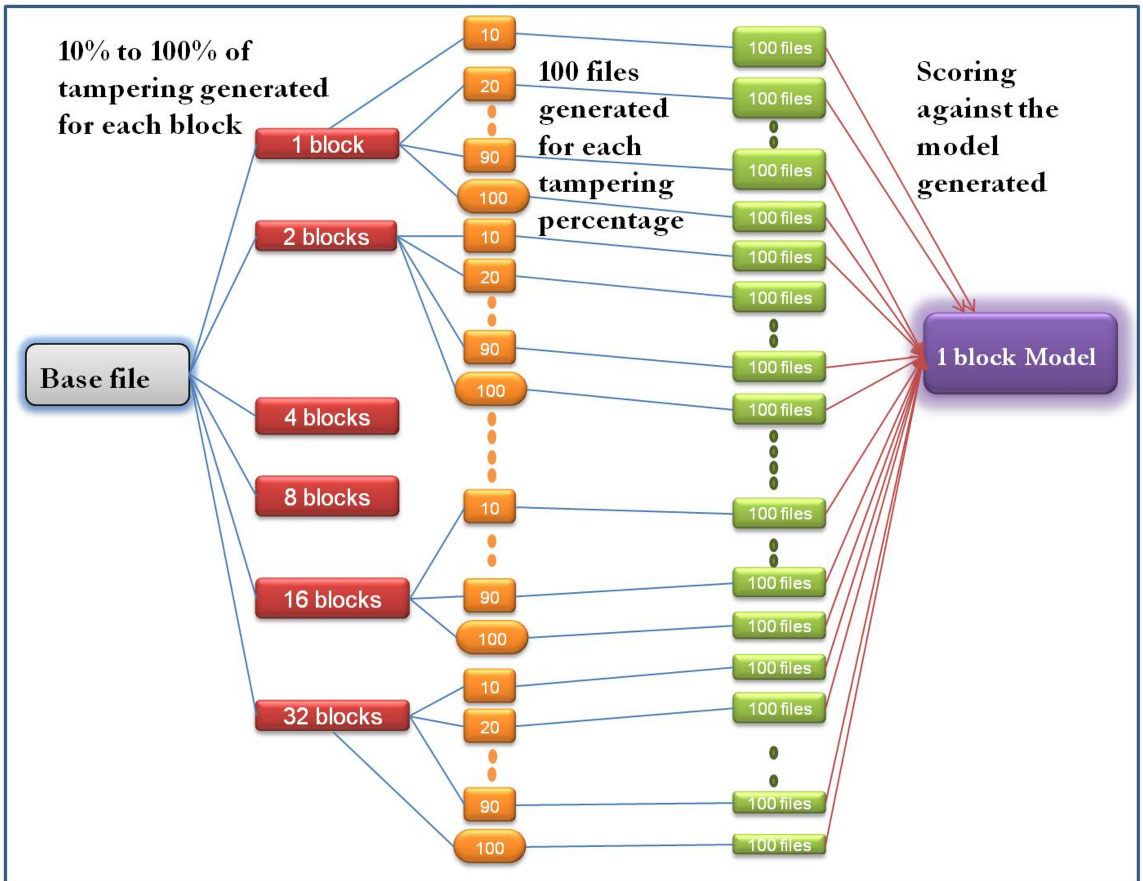

Figure 9: Tampered files generation for Experiment 1

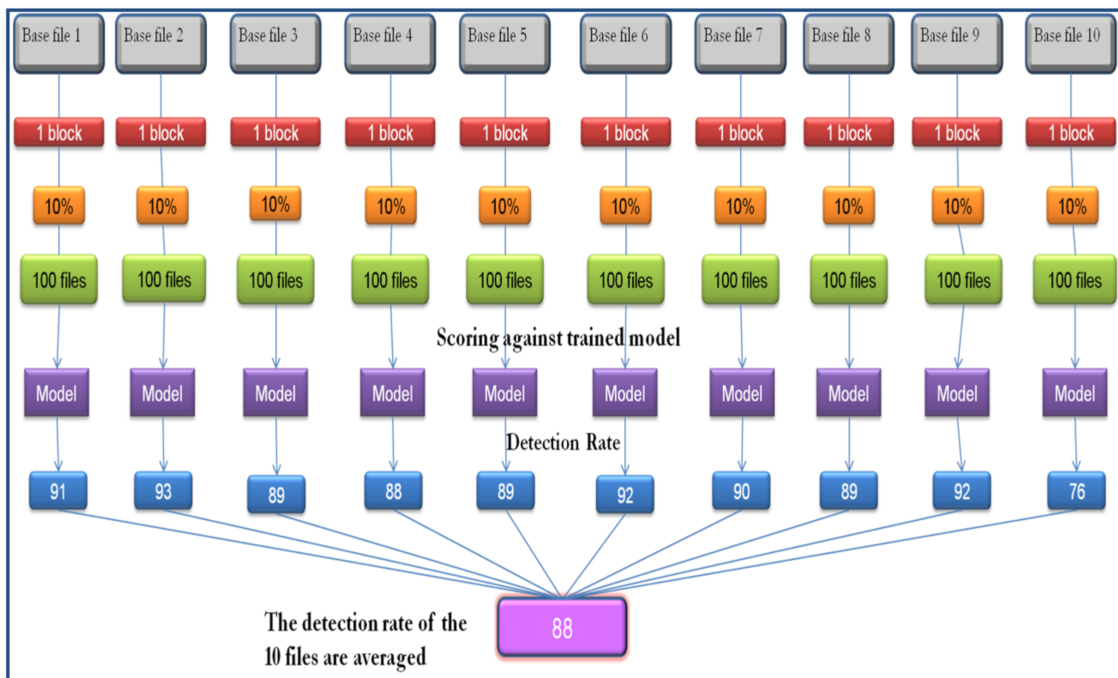

Figure 10: Detection rate average calculation

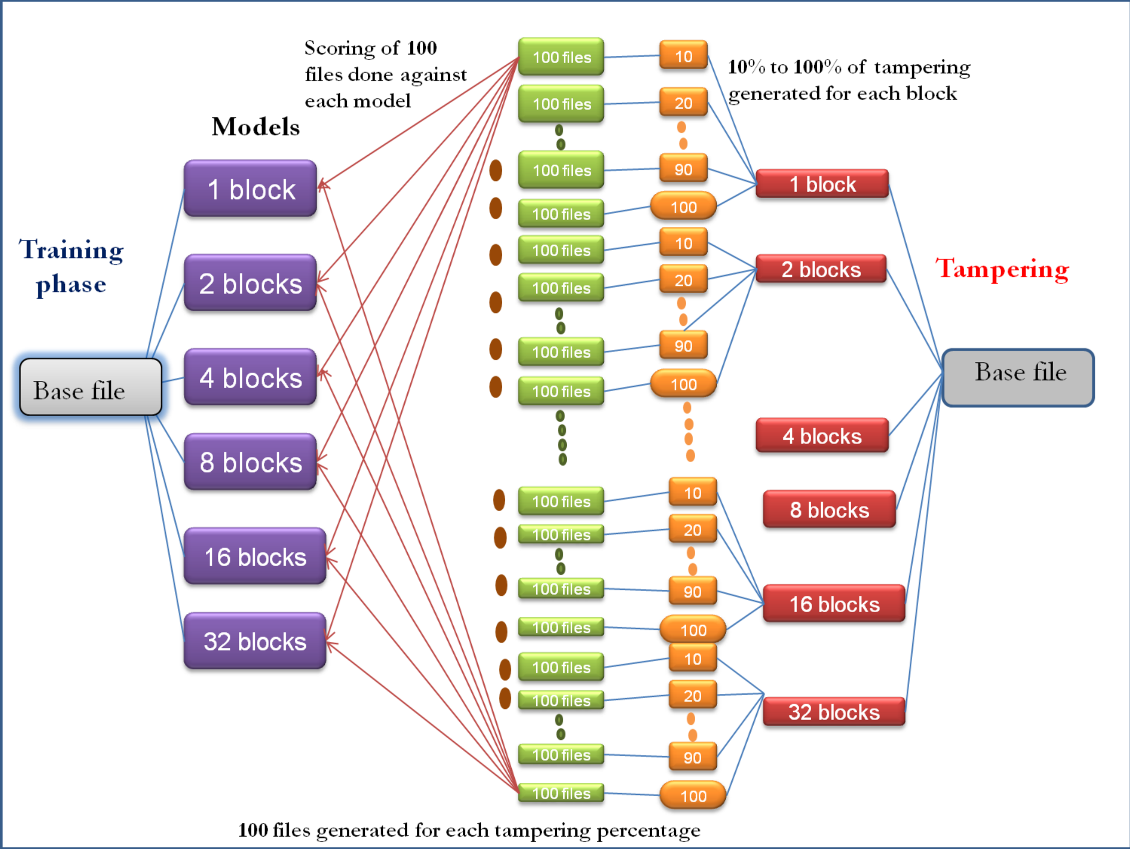

Figure 11: Representation of all experiments

CHAPTER 6

Results and Observations

This section describes the results obtained and observations made in each experiment. Since the results are almost the same for the different trained models, we have described the results obtained in Experiments 1 and 6. For the rest of the experiments, the results are in the Appendix C.

6.1 Experiment 1: Training by morphing with 1 block

There were a number of observations that were made in this experiment. Firstly, for 1 block of tampering with up to 40% of tampering, more than 50% of files are classified as the original base file. Beyond, 60% and 1 block of tampering, no files are classified as the original base file. This happens because as more and more dead code is inserted at the end of the base program, it looks less similar to the original base file. Secondly, for 2, 4, 8 and 16 blocks of tampering, with up to 50% of tampering more than 50% of files are classified as base file. For 32 blocks of tampering, with up to 30% of tampering, more than 50% of files are classified as base file. The detection rate for 32 blocks of tampering is lower than the other blocks of tampering. This happens because with 32 blocks of tampering, a high number of segments of dead code are inserted into the base file. Due to this, the tampered files looks more like some other file rather than the original file. Figure 12 gives the detection results.

The detection results for experiments 2, 3, 4 and 5 are given in Appendix C.

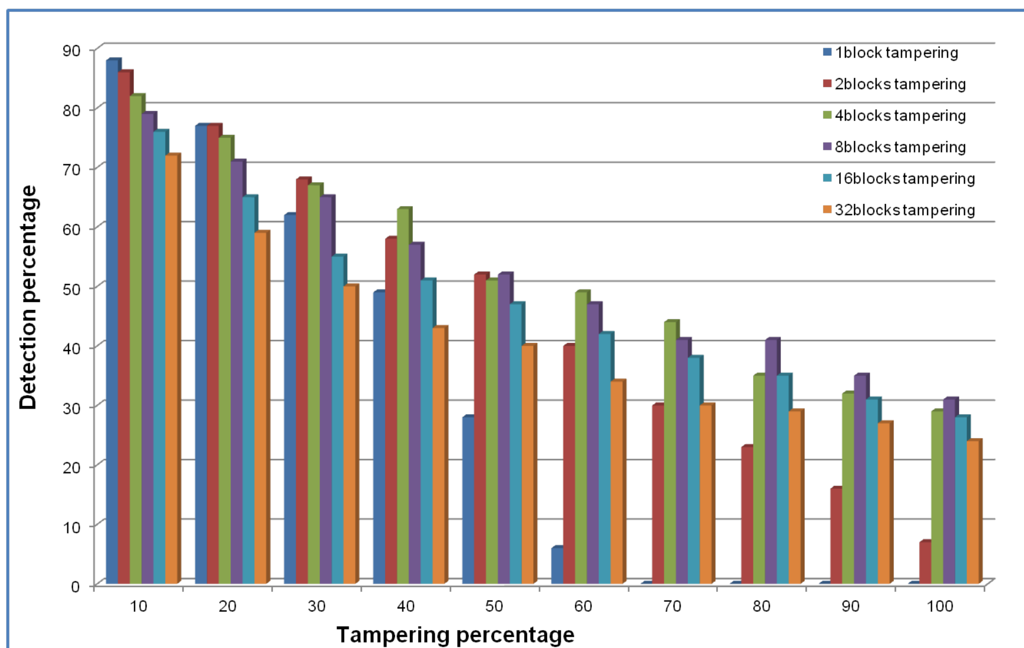

Figure 12: Experiment 1 Detection Results

6.2 Experiment 6: Training by morphing with 32 blocks

In this experiment, the hidden Markov model was generated by training with morphed files which had 32 blocks of insertion. Firstly, for 1 and 16 blocks of tampering, with up to 40% of tampering, more than 50% of files are classified as the original base file. Beyond, 60% and 1 block of tampering, no files are classified as the original base file. This happens because as more and more dead code is inserted at the end of the base program, it looks less similar to the original base file. Secondly, for 2 and 8 blocks of tampering, with up to 50% of tampering more than 50% of files are classified as base file. For 4 blocks of tampering, with up to 60% of tampering more than 50% of files are classified as base file. For 32 blocks of tampering, with up to 30% of tampering, more than 50% of files are classified as base file. Figure 13 represents the detection results.

Figure 14 is a 3-dimensional representation of the detection rate, tampering rate

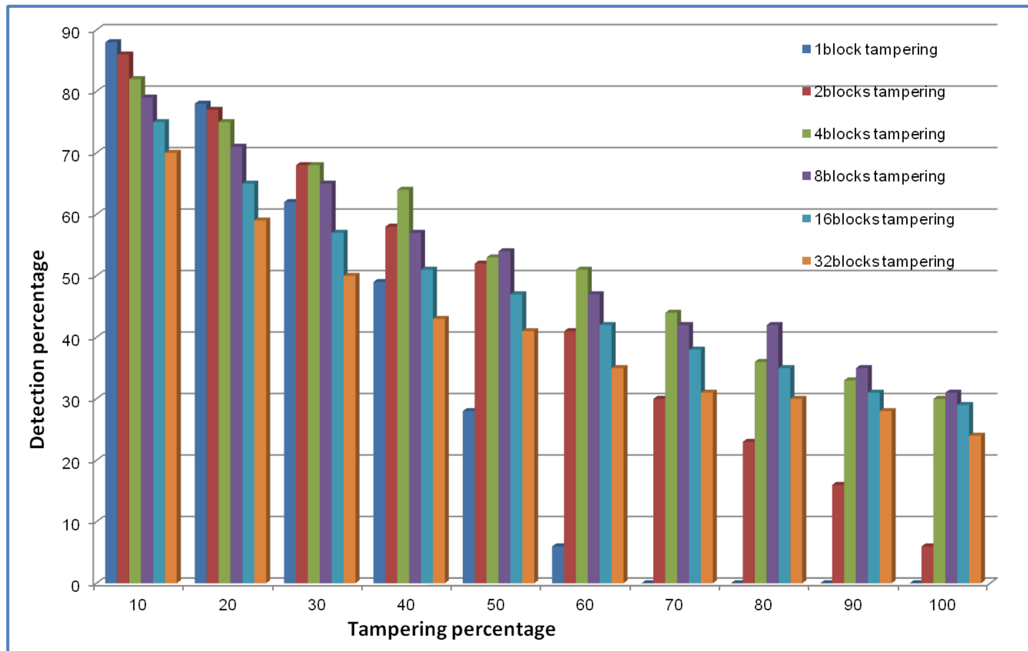

Figure 13: Experiment 6 Detection Results

and number of blocks inserted for tampering. From the figure, we can observe that as the tampering rate increases, the detection rate reduces. The detection rate is higher at lower tampering rates. The graph indicates that if an attacker tampers the original software by more than 60%, then the chances that the tampered software would go undetected is higher. From Figure 12 and Figure 13, we can conclude the best way an attacker can tamper the code. If an attacker inserts a block of code at the end of the program and uses a tampering percentage of 60% or more, the tampered file would go undetected.

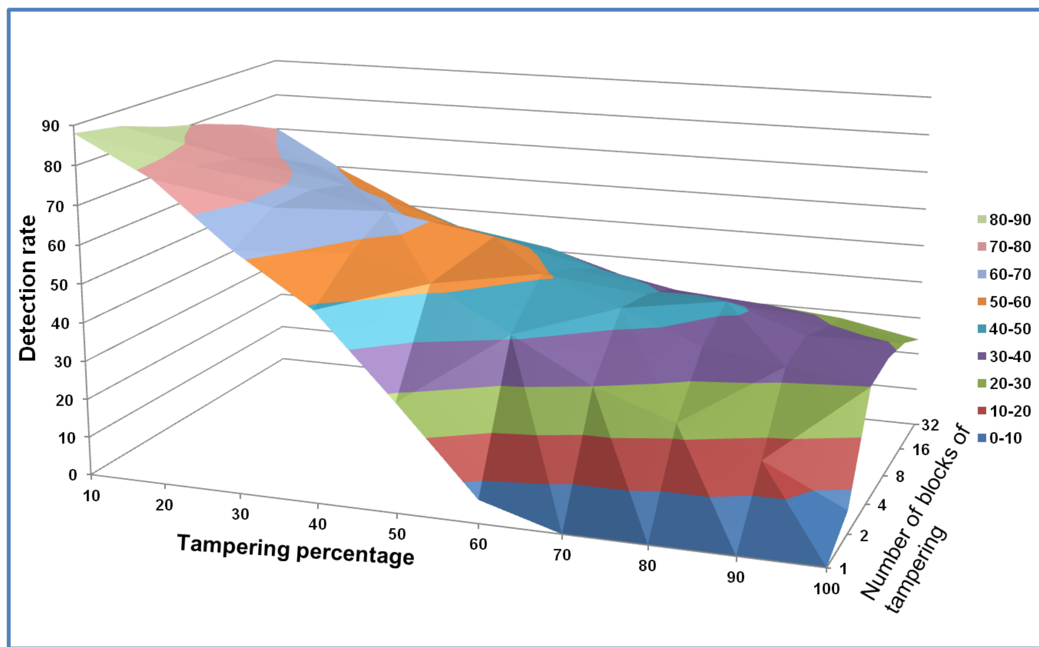

Figure 14: Results Summary

CHAPTER 7

Conclusion

The experimental results show that the scheme is robust.As the tampering rates increase, the detection rate of base files decreases.

We observed that models that were generated by inserting different blocks of code in the base file performed almost the same. For one block of insertion, no tampered files were classified as base file beyond 60% of tampering. This happens because as the percentage of tampering increases, more and more lines of code from some other file is inserted at the end of the base file. Detection rates of 4 blocks, 8 blocks and 16 blocks of tampering showed the best results compared to other blocks of insertion. Overall, on an average more than 50% of files were detected with up to 50% of tampering. In other words, if an attacker tampers a piece of software by up to 50% of dead code insertion, our approach would have a high chance of detecting the tampered software.

CHAPTER 8

Future work

The future work would be to try metamorphism in different ways. Other methods of morphing such as code substitution should be experimented with. More experiments on other types of files should also be carried out to test this approach. Since inserting large blocks of data at the end of base file goes undetected, ways to handle this form of tampering is worth researching. More research on the way to train HMM should be done in order to be able to detect files that have been tampered by inserting code at the end of the base program. Also, for more than 50% of tampering, the success rate of classification of base file decreases. Therefore, more research needs to be done in order to detect software that was tampered by more than 50%. Moreover, models with different percentage of morphing can also be generated and scored against to test the results.

LIST OF REFERENCES

[1] Birrer B. D., Raines R. A., Baldwin R. O., Mullins B. E. & Bennington R.W. (2007). Program Fragmentation as a Metamorphic Software Protection. In *Proceedings of the Third International Symposium on Information Assurance and Security* (pp. 369–374). Washington DC, USA: IEEE Computer Society

[2] Cesare, S. (2010, May). *Fast Automated Unpacking and Classification of Malware*. Masters Thesis, Central Queensland University. Retrieved on April 1, 2012 from
`http://www.scribd.com/doc/43697483/Fast-Automated-Unpacking-and--Classification-of-Malware/`

[3] Costello F., Bleakley C.& Aliefendic S.(n.d). *Using whitespace patterns to detect plagiarism in program code*. Retrieved on April 1, 2012 from
`http://www.csi.ucd.ie/content/using-whitespace-patterns-detect-plagiarism-program-code`

[4] Cronin G.(2002). *A Taxonomy of Methods for Software Piracy Prevention*. Technical Report, University of Auckland, New Zealand. Retrieved on March 25, 2012 from
`http://www.croninsolutions.com/writing/piracytaxonomy.pdf`

[5] Finones R.G. and Fernandez R.T(2006, March). Solving the metamorphic puzzle. *Virus Bulletin*, pp. 14–19.

[6] Gao X. & Stamp M. (2005). Metamorphic Software for Buffer Overflow Mitigation. In *Proceedings of the 2005 Conference on Computer Science and its Applications*. Retrieved on March 27, 2012, from
`http://www.cs.sjsu.edu/faculty/stamp/papers/BufferOverflow.doc`

[7] IDA Pro DisAssembler
`http://www.hex-rays.com/index.shtml/`

[8] Kohavi R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (pp. 1137–1143).

[9] Konstantinou E. (2008, January). *Metamorphic Virus: Analysis and Detection.* Technical Report, Royal Holloway, University of London. Retrieved on March 15, 2012, from
`http://www.ma.rhul.ac.uk/static/techrep/2008/RHUL-MA-2008-02.pdf/`

[10] Lin D. & Stamp M. (2011). Hunting for undetectable metamorphic virus. *Journal in Computer Virology*, 7(3), pp. 201–214.

[11] Lukashenko R., Graudina V. & Grundspenkis J. (2007). Computer-Based Plagiarism Detection Methods and Tools: An Overview. In *Proceedings of the 2007 international conference on Computer systems and technologies* (pp. 1–6). New York, NY, USA: ACM.

[12] Muhaya F. B., Khan M. K. & Y. Xiang (2011). Polymorphic Malware Detection Using Hierarchical Hidden Markov Model. In *Proceedings of the 2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing* (pp. 151–155). Washington DC, USA: IEEE Computer Society

[13] Mungale M. (2011, May). *Robust watermarking using hidden Markov models.* Masters Thesis, Department of Computer Science, San Jose State University. Retrieved on September 1, 2011, from
`http://www.cs.sjsu.edu/faculty/stamp/students/mungale_mausami.pdf/`

[14] 2010 Piracy Study. Retrieved on March 1, 2011, from
`http://portal.bsa.org/globalpiracy2010/downloads/study_pdf/2010_BSA_Piracy_Study-Standard.pdf/`

[15] Rabiner L.R. (1989). A tutorial on hidden Markov models and selected applications in speech recognition. In *Proceedings of the IEEE* (pp. 257–286).

[16] Rad B.B. & Masrom M. Metamorphic virus variants classification using opcode frequency histogram (2010). In *Proceedings of the 14th WSEAS international conference on Computers* (pp. 147–155). Stevens Point, Wisconsin, USA: WSEAS.

[17] Rigoll G. (1994). Maximum Mutual Information Neural Networks for Hybrid Connectionist-HMM Speech Recognition Systems. *IEEE Transactions on Speech and Audio processing*, 2(1), pp. 175–184.

[18] Si A., Leong H. V. & Lau R. W. H. (1997). CHECK: A Document Plagiarism Detection System. In *Proceedings of the 1997 ACM Symposium on Applied Computing* (pp. 70–77). New York, NY, USA: ACM.

[19] Software Piracy. Retrieved on March 1, 2011, from
`http://www.bsa.org/country/Anti-Piracy/What-is-Software-Piracy.aspx/`

[20] Software Piracy on the Internet: A threat to the security. Retrieved on March 1, 2011, from
`http://portal.bsa.org/internetreport2009/2009internetpiracyreport.pdf/`

[21] Stamp M (2004). *A Revealing Introduction to Hidden Markov Models.* Retrieved on November 1, 2011, from
`http://www.cs.sjsu.edu/~stamp/RUA/HMM.pdf/`

[22] Stamp M. (2010). *Information Security: Principles and Practice*, (2nd edition). Hoboken: Wiley.

[23] Stamp M.(2004, March). Risks of Monoculture. *Communications of the ACM — Homeland Security*, 47(3), p. 120.

[24] Stamp M. & Mungale M. (2011). Software Similarity and Metamorphic Detection. In *Proceedings of 2012 International Conference on Security & Management* (SAM '12).

[25] Ször P. & Ferrie P. (2001, September). Hunting for metamorphic. *Virus Bulletin Conference.*

[26] Wong W. & Stamp M. (2006) Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3), pp. 211–229.

APPENDIX A

Programs used

Table A.3: Programs used

Program Name	Functionality
morph.java	This program is used to generate 100 morphed copies with 10% of morphing.
tamper.java	This program is used to generate tampered copies by inserting 1 block, 2 blocks, 4 blocks, 8 blocks, 16 blocks , 32 blocks and tampering percentage from 10% to 100%
scoring.sh	This script is used to score all the tampered files against the corresponding model and count the number of files that score above the threshold.
average_scoring.sh	This script is used to average the count of files that are above the threshold for 10 Cygwin files
cross-validate.rb [5]	Performs k-fold cross validation
compareAsm.rb [5]	Reads and compares two assembly programs specified by filename1 and filename2
hmm_score.rb [5]	Used by read-train-test to find the log Likelihood per opcode of a file
read-train-test.rb [5]	Perform one round of K-fold cross validation. Read virus assembly files from DataSet, excluding viruses in the selected test set. Write data file (.in) and alphabet file (.alphabet) for HMM training to directory 'TrainFile'. Train a HMM with files in training set and write model to file (.model)in directory
score.rb [5]	Score input ASM file using the model stored in model file

APPENDIX B

Pseudocode for morph.java

```
numoflines_base = countlines(basefile)  //The number of lines of base file is
                                        //calculated by calling the countlines
                                        //function

numoflines_normal = countlines(normalfile) //The number of lines of normal file
                                           //is calculated by calling the
                                           //countlines function

linestoinsert = morphingpercent/100 * numoflines_base  //The lines to insert for
                                                       // morphing is calculated

partition = (numoflines_base / block); //This determines after how many lines
                                       //in the base file the blocks need to
                                       //be inserted
blockinsert = linestoinsert / block;  // This determines the number of lines in
                                      // each block
 for int i = 0 to 100
  begin for
   for int j=0 to block
   begin for

/* In this function the source file to copy from, the starting line of source
file,ending line of source file and the destination file are given as parameters*/

   copylines(basefile, startpoint, startpoint+partition-1, IDANi)
   if(blockinsert> numoflines_normal)
     print Error: There are more lines to print that present in normal file
     else begin
       randomrange = numoflines_normal  blockinsert
       randomnumber = rand(randomrange)
       copylines(normalfile, randomnumber, randomnumber+blockinsert, IDANi)
       startpoint = startpoint + partition
     end else
    end for
   end for
 end for
```

APPENDIX C

Experiment Results

C.0.1 Experiment 2: Training by morphing with 2 blocks

Figure C.15 represents the detection results.

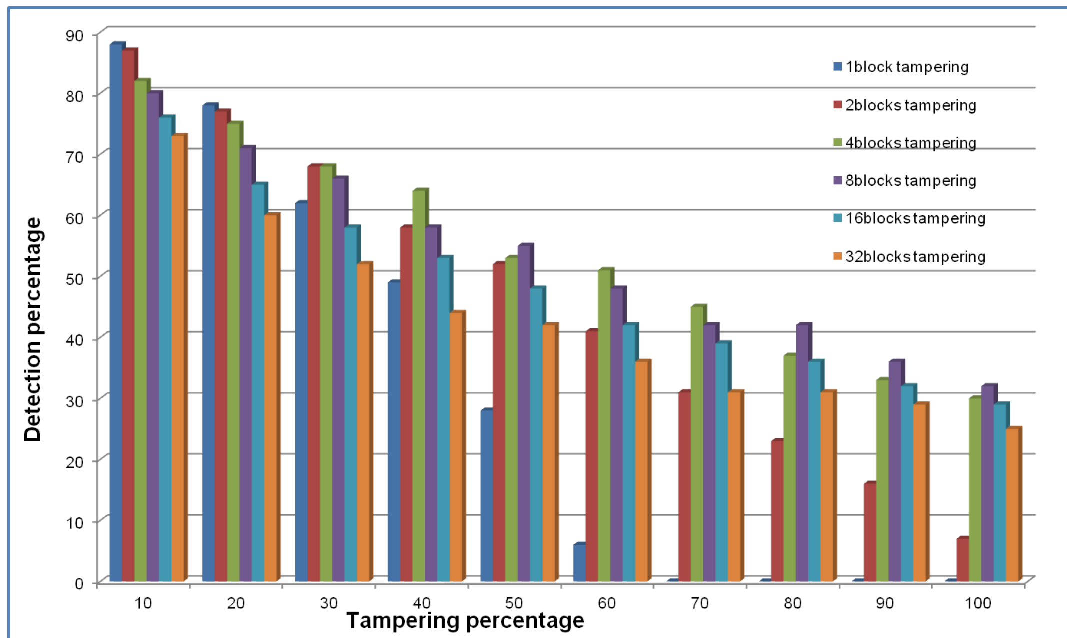

Figure C.15: Experiment 2 Detection Results

C.0.2 Experiment 3: Training by morphing with 4 blocks

Figure C.16 represents the detection results.

C.0.3 Experiment 4: Training by morphing with 8 blocks

Figure C.17 represents the detection results.

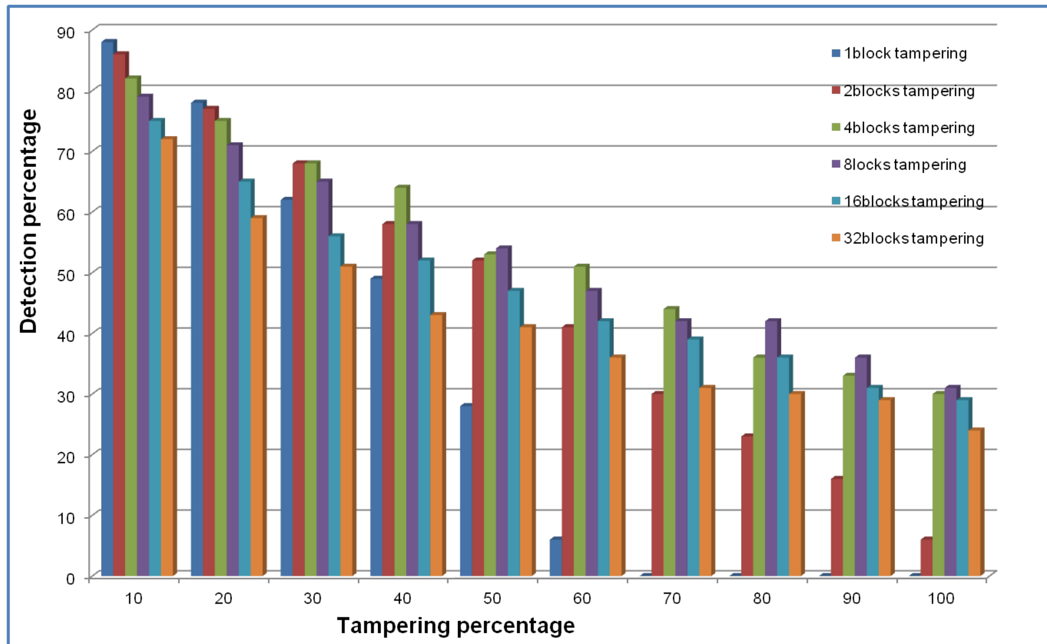

Figure C.16: Experiment 3 Detection Results

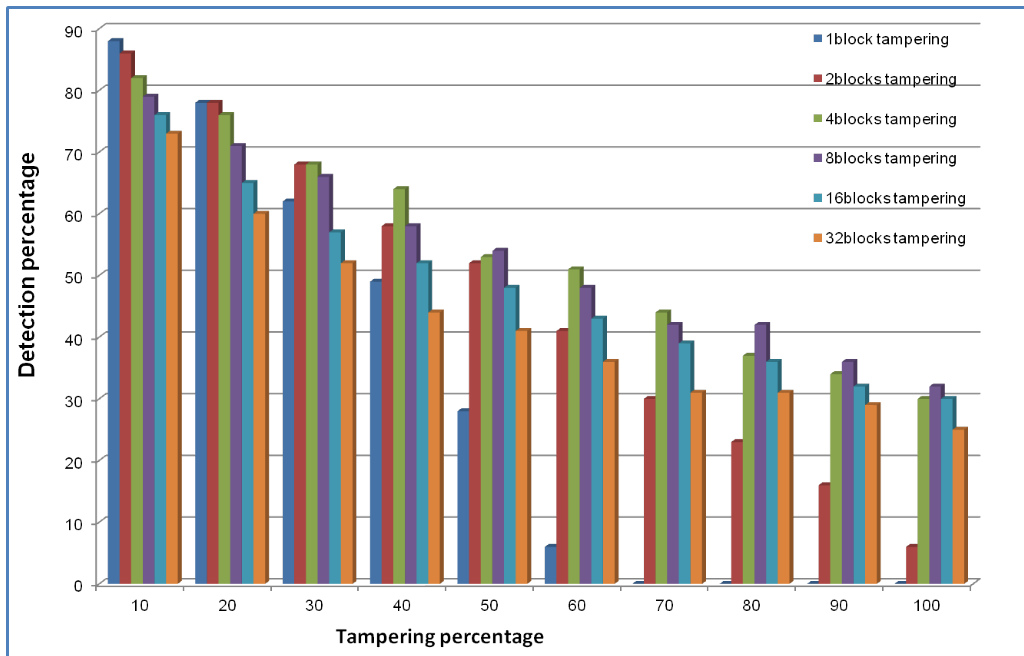

Figure C.17: Experiment 4 Detection Results

C.0.4 Experiment 5: Training by morphing with 16 blocks

Figure C.18 represents the detection results.

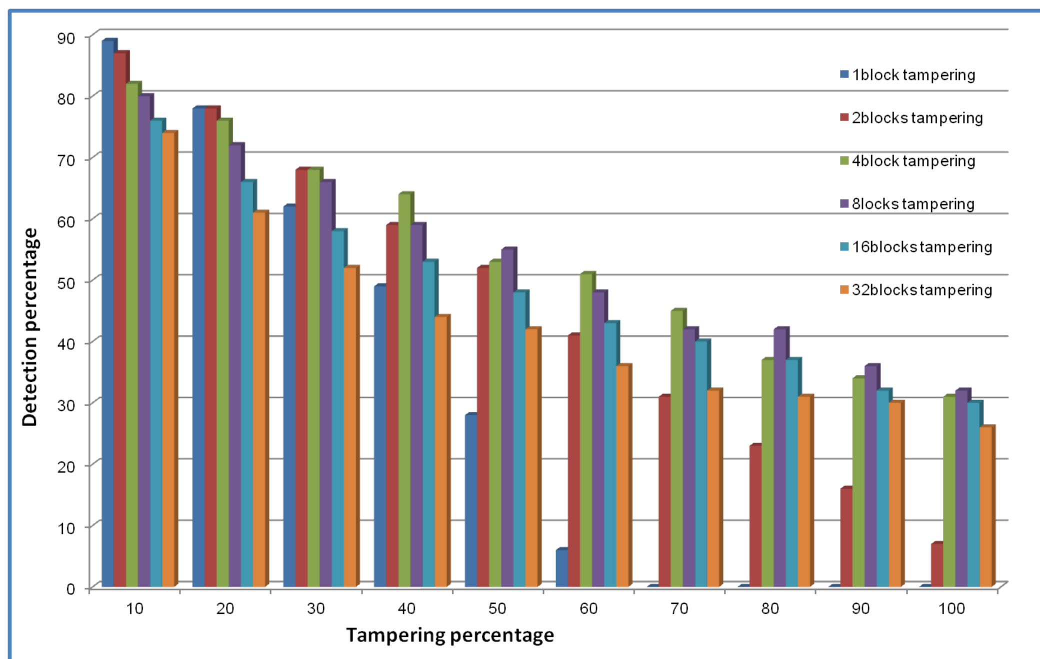

Figure C.18: Experiment 5 Detection Results