Spring 2011

Robust Watermarking using Hidden Markov Models

Mausami Mungale
San Jose State University

Robust Watermarking using Hidden Markov Models

A Thesis

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the

Requirements for the Degree

Masters of Science

By

Mausami Mungale

May 2011

The Designated Thesis Committee Approves the Thesis Titled

Robust Watermarking using Hidden Markov Models

by

Mausami Mungale

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

May 2011

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

Table of Contents

List of Figures

List of Tables

Acknowledgements

I am grateful to take this opportunity to sincerely thank my thesis advisor, Dr. Mark Stamp, for his constant support, invaluable guidance, and encouragement. His work ethic and constant endeavour to achieve perfection have been a great source of inspiration. I wish to extend my sincere thanks to Dr. Sami Khuri and Dr. Chris Pollett for consenting to be on my defence committee and for providing invaluable suggestions in my project.

I would like to specially thank my husband, Alok Tongaonkar, for encouraging me to achieve my goals. Without his cooperation, I could not have achieved this milestone. I owe my deepest gratitude to my parents, Madhuri and Mahesh Mungale, for their love and care. I am very grateful to my parents-in-law, Shubhangi and Satish Tongaonkar, for their emotional support and blessings. I wish to thank Prisha, Shreyus, Mayura, and Madhur Bobde, for providing me a home away from home, and making my stay very comfortable and enjoyable. I wish to extend my thanks to Nupura and Alok Kogekar for their unconditional love and support throughout my studies. I am grateful to Aryan, Meghana, and Nitin Sadawarte, for welcoming me into my new family and helping me get settled. Finally, and most importantly, I would like to dedicate my work to my grandfather, Madhukar Kulkarni, and grandmother, Saraswati Mungale, without whose blessings this work would not have been possible.

Abstract

Software piracy is the unauthorized copying or distribution of software. It is a growing problem that results in annual losses in the billions of dollars. Prevention is a difficult problem since digital documents are easy to copy and distribute. Watermarking is a possible defense against software piracy. A software watermark consists of information embedded in the software, which allows it to be identified. A watermark can act as a deterrent to unauthorized copying, since it can be used to provide evidence for legal action against those responsible for piracy.

In this project, we present a novel software watermarking scheme that is inspired by the success of previous research focused on detecting metamorphic viruses. We use a trained hidden Markov model (HMM) to detect a specific copy of software. We give experimental results that show our scheme is robust. That is, we can identify the original software even after it has been extensively modified, as might occur as part of an attack on the watermarking scheme.

1. Introduction

In recent years the software industry has faced a growing problem of piracy. Software piracy can be defined as making illegal copies or using commercial software purchased by someone else. Software piracy has many adverse effects. Each pirated copy of software takes away from profits, reducing funds for further software development activities. In addition to reducing the revenues for local information technology (IT) services and distribution firms, software piracy lowers the tax revenue. Use of pirated software also increases the risk of cyber crimes and security problems as pirated software can be used to install Trojans and malware. In spite of the efforts of the software industry and governments across the world, software piracy has been steadily increasing. A recent study of global personal computer software piracy by Business Software Alliance (BSA) and IDC, IT industry's leading global market research and forecasting firm, shows that software piracy levels rose globally from 41% in 2008 to 43% in 2009 [1]. The impact of software piracy on the software industry just in terms of lost revenue was more than 50 billion US dollars.

A variety of ethical, legal, and technical solutions are employed to prevent software piracy [2]. Ethical and legal techniques like copyright laws focus on making it less desirable for consumers to use pirated software. A number of technical solutions also aid in the fight against software piracy. Techniques like encryption and obfuscation aim to increase the difficulty and thereby, the cost of duplicating software. Another effective technique for piracy prevention is digital watermarking, the process of embedding information into a digital object in a way that is difficult to remove [3]. A digital watermark enables us to determine whether the software is ours and it could also tell us who originally purchased the software. This could reduce software piracy since people would be less likely to illegally share software. Digital watermarks can also aid in

taking legal action against the pirates. Thus, digital watermarks can act as powerful deterrents to software piracy.

In this project, we developed an approach for watermarking that was inspired by previous studies of viruses that change their structure (but not their function) each time they replicate [4]. This technique of transforming code into functionally equivalent but structurally different code is called as metamorphism. It makes it difficult to detect virus since anti-virus software have fixed signatures that do not match the metamorphic copies. Metamorphism can be used in the context of software watermarking to embed a unique watermark in each instance of the software.

The remainder of this report is organized as follows. Background information about digital watermarking, metamorphism, and hidden Markov models are discussed in detail in Section 2. Section 3 contains an overview of the design of our watermarking technique. In Section 4, we discuss our implementation in some detail and experimental results are given in Section 5. Section 6 concludes the report.

2. Background

In this section, we describe three concepts that are essential for understanding the work done in this project:

i) Digital Watermarking

ii) Metamorphism

iii) Hidden Markov Model

2.1 Watermarking

Watermarking is the technique of embedding some special mark in an object to identify the object [3]. This project addresses software watermarking, which we accomplish by embedding some special pieces of code in software to identify the software. These special pieces of code serve as a watermark and enable us to identify the code.

2.1.1 Why Watermark?

In the past, duplicating copyrighted material generally required a significant effort and there was often a loss of quality in the duplication process. However, with digital content, a single click of a mouse is often sufficient to make a perfect digital copy [5]. An embedded watermark can be used to identify the purchaser of the work and thereby detect such copying. This concept is also applicable to all types of digital content, including audio and video. Digital watermarking has many applications which require watermarks that have different properties [6]. We discuss a few of these applications below.

a) Ownership Assertion

Watermarks can be used to prove ownership. Alice can embed some information in a digital object before giving it to Bob. If Bob makes a copy of the object, then the embedded watermark is also copied. Later if Bob distributes this copy of the object, Alice can detect the presence of her watermark and know that it is a copy of the object owned by her. It is essential in these applications that the watermark can be detected even if the copy is modified.

b) Tamper Detection

In some applications it is important just to know if a digital object has been tampered. Watermarks can be used to detect tampering. For tamper detection, we can use watermarks that become undetectable even if small modifications are made to the object. For instance, Alice can embed such a watermark in a digital object. If Bob modifies the object, the watermark will become undetectable revealing the fact that the object has been tampered.

c) Fingerprinting

In some applications where digital content is distributed, the owner of the content may want to identify the person making unauthorized copies by embedding unique watermarks in each copy. When an unauthorized copy is found, the watermark in the copy can be used to identify the person who made the copy. In these applications the watermarking scheme must be such that it is difficult to remove or forge a watermark.

2.1.2 Classification of Digital Watermarking Techniques

As we can see there are various properties of watermarks that are useful in different applications. We can classify digital watermarking techniques in different ways based on these properties. One

way of classifying watermarks is based on the **visibility** [3]. Watermarks can be **visible** or **invisible**. A visible watermark is intended to be known to the user. An example of a digital watermark that is visible is a visible company logo in a classified document. Invisible watermarks, on the other hand, are supposed to be imperceptible to the user. Invisible watermarks have been proposed for digital music. Ideally, such watermarks, which are designed to detect music piracy, must not degrade the quality of the audio and they must be sufficiently robust to withstand attacks by the pirates.

Digital watermarks can also be classified based on their **robustness** [3]. A **robust** digital watermark can be reliably detected even if an attacker attempts to degrade it. A **fragile** watermark on the other hand is rendered undetectable even if a minor modification is applied to the object. A robust digital watermarking scheme would provide an effective means for copyright protection (ownership assertion/fingerprinting). In contrast, a fragile watermark could be used to detect tampering.

Other ways of classification are also possible such as based on the **embedding method** (**spread spectrum, quantization type**) and the **capacity** (**zero-bit watermarking, multi-bit watermarking**) that are relevant to multimedia content [3]. In the context of software watermarking, we can broadly classify the techniques based on the way that the embedding is done. **Static watermarking** techniques embed the watermark inside the program code [7] [8]. **Dynamic watermarking** techniques on the other hand store the watermark in the execution state of the program [9][10]. In this project, we focus on developing a static watermarking technique.

2.1.3 Challenges in Digital Watermarking

Previous research in this area has shown that developing a robust watermarking scheme is a challenging problem. In September 2000, Secure Digital Music Initiative [11], a consortium of parties interested in preventing piracy of digital music, issued a challenge to test the strength of four watermarking technologies [12]. For each technique, they released a file (File 1) and its watermarked version (File 2). They also provided another watermarked music file (File 3). The challenge was to remove the watermark from the third file without distorting the quality of sound. No information about the watermarking technique was provided. An online "oracle" was the only help that the testers had. Testers could send the file that they created by removing the watermark to the oracle and it would inform them if they had succeeded, i.e., the watermark was not detectable in their new file and it sounded like File 1. Craver et al. [14] showed that all four techniques could be defeated using standard signal processing techniques [13]. Continuing research in this area has shown that newer techniques for digital watermarking are still open to such removal attacks.

Over the years, many static techniques have been developed for software watermarking. Various kinds of attacks are possible against these techniques. Additive attacks involve inserting additional watermark into already watermarked software. The goal of this attack is to make the original watermark undetectable. Distortive attacks involve using semantics preserving transformations like variable renaming and function in-lining to make the watermarks undetectable. Subtractive attacks involve identifying the watermark and removing it without changing the functionality of the program. In this project, we develop a watermarking scheme based on trained hidden Markov models of executables that provides strong protection against many of these attacks.

2.2 Metamorphism

Metamorphism is the process of transforming a piece of code into unique copies that are functionally equivalent but structurally different. Metamorphism has been used widely by virus writers to defeat signature based anti-virus software. Virus writers create metamorphic copies of virus that can not be matched by the existing signatures. In recent years, metamorphism has also been used by security researchers to prevent various attacks. Diversity in software can reduce the impact of many implementation-level attacks like buffer overflow and stack smashing [15]. Metamorphism has also been used previously for generating watermarking schemes [16].

2.2.1 Assembly Language Basics

In this project, we developed a technique for watermarking based on generating metamorphic copies of **x86 assembly code**. Before we discuss the operations performed by our metamorphic generator, we provide some background information about x86 architecture and assembly language here [17]. This will help to understand the transformations that we perform on the code.

2.2.1.1. Central Processing Unit (CPU) Internals

The term x86 refers to a family of instruction set architectures that are based on Intel 8086 processor. It is one of the most popular instruction set architectures for personal computers. There is almost full binary backward compatibility between the Intel 8086 chip through to the current generation of x86 processors like the Pentium series. Here we discuss the 32-bit x86 architecture. The main components of the CPU are –

1. Clock
2. Control Unit

3. Arithmetic and Logic Unit (ALU)
4. Registers

For understanding the assembly language, we need to know the different types of registers. We describe the types of registers below.

General Purpose Registers

The general purpose registers are used for mainly for arithmetic and data movement. These 32-bit registers are:

- EAX, EBX, ECX, EDX – The lower 16-bits of these registers can be accessed by omitting the "E" from the name such "AX". Within the AX register, each individual byte can be accessed as "AH" (high byte) and "AL" (low byte) as shown in Figure 1.
- ESP, EBP, ESI, EDI – These can be used as 32-bit registers or 16-bit registers by removing the "E" in the name. ESP is the extended stack pointer. EBP is used as base pointer to reference local variables in a function. ESI and EDI are used in high-speed memory transfer instructions as source index and destination index.

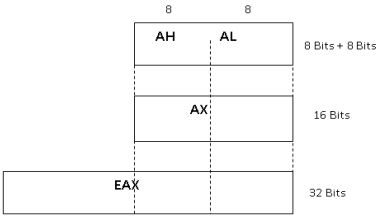

Figure 1: General Purpose Registers

Segment Registers

Segment registers (CS, DS, ES, FS, GS, and SS) are used to hold the base address of pre-assigned areas in memory called as segments. These are 16-bit registers that are sometimes used in conjunction with certain general-purpose registers to access specific memory locations. CS is

called the code segment register. It is used to point to the code within the currently executing program. DS, called the data segment register, points to the segment containing the data of the executing program. ES, called extra segment register, is also used in data operations, usually with the DI register. FS and GS are used to hold extra data pointer similar to ES. SS is called stack segment register and is used to point to the stack region.

Special Purpose Registers

There are two special purpose registers in x86 architecture. EIP, called the extended instruction pointer, stores the address of the next instruction to be executed. EFLAG is used to keep track of the CPU state. The value of flags in EFLAG is set by CPU whenever a mathematical or logical operation is carried out. The value in EFLAG can be used to change the control flow of the program. Figure 2 shows all the x86 registers.

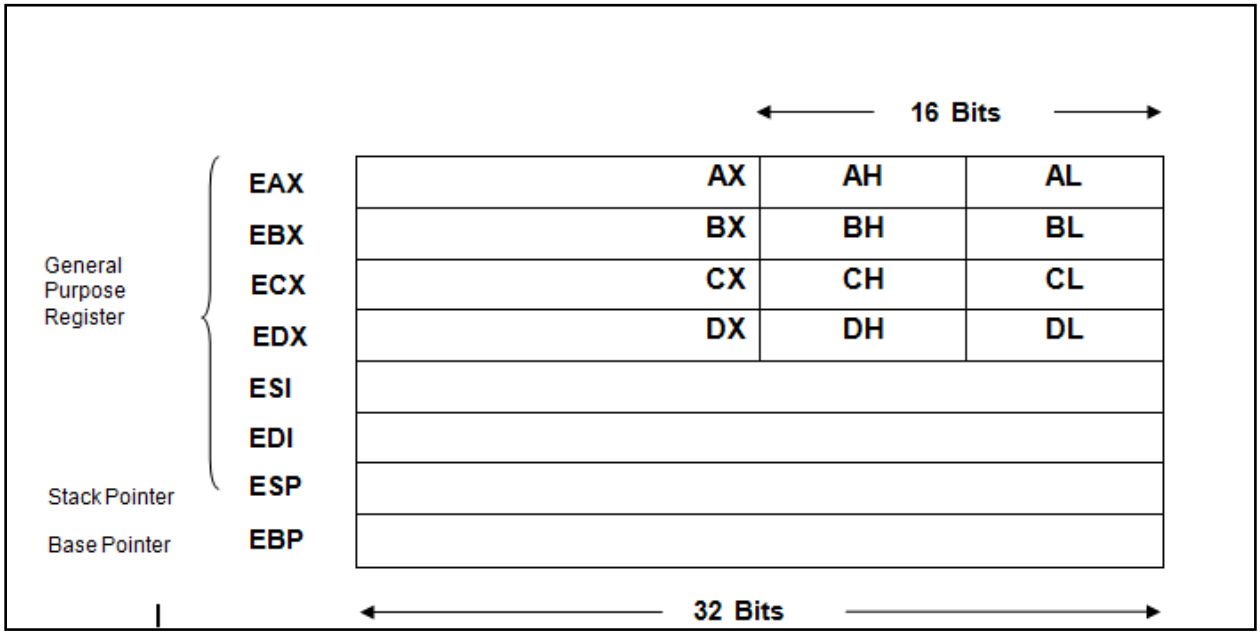

Figure 2: x86 Registers

2.2.1.2 x86 Instruction Set

The full x86 instruction set is large and complex [18]. The instructions consist of an "opcode" followed by one or more operands. The operands can be a constant value, a pointer to a value in memory, or a register. The instructions can broadly be classified as data transfer instructions, arithmetic and logical instructions, and control flow instructions. Table 1 shows some instructions of each type.

Data Transfer Instruction	
MOV	Move byte or word to register or memory
IN, OUT	Input/output byte or word
LEA	Load Effective Address
PUSH, POP	Push/Pop word on/from stack
Arithmetic and Logical Instructions	
NOT	Logical NOT of byte or word
AND	Logical AND of byte or word
OR	Logical OR of byte or word
XOR	Logical XOR of byte or word
ADD, SUB	Add, subtract byte or word
INC, DEC	Increment, decrement byte or word
NEG	Negate byte or word (two's complement)
MUL, DIV	Multiply, divide byte or word (unsigned)
Control Flow Instructions	
JMP	Unconditional jump
JE/JNE	Jump if equal/Jump if not equal
LOOP	Loop unconditional, count in CX, short jump to target address
CALL, RET	Call, return from procedure

Table 1: x86 Instruction Set

2.2.2 Techniques for Generating Metamorphic Code

An important component of our system is the metamorphic code generator. There are a large number of semantics preserving transformations that can be applied to assembly code to get metamorphic copies [19]. Here we discuss some of these techniques.

2.2.2.1 Inserting Data Flow and Control Flow Preserving Instructions

In this transformation we insert single instruction or sequence of instructions that have a combined effect of not changing the data flow and the control flow. We present a few examples here.

1. NOP is a special instruction that has no effect on the execution state. It is simply a "do nothing" instruction. Therefore we can insert nops between instructions as shown below.

Original Code	**Transformed Code**

MOV AL, BL
 	
ADD AL, 05H
	MOV AL, BL
	
	NOP
	
	ADD AL, 05H

2. We can use groups of arithmetic or logical instructions, the net effect of which does not change the value of any registers. Since arithmetic instructions can change the flags, we may need to store and restore the EFLAG register when inserting such code. Below are some examples of such instruction groups.

ADD AX, 05H SUB AX, 05H
XOR AX, 0H
AND AX, FFFFH

3. We can add a label to any instruction and put a “jmp” instruction to that label just before the instruction. This does not change the program behavior in any way.

Original Code	**Transformed Code**
MOV AL, BL ADD AL, 05H	MOV AL, BL JMP LOC1 LOC1: ADD AL, 05H

4. We can push the value of some register on the stack and pop it immediately to preserve the program semantics.

Original Code	**Transformed Code**

MOV AL, BL
ADD AL, 05H
	MOV AL, BL
	PUSH AX
	POP AX
	ADD AL, 05H

2.2.2.2 Dead Code Insertion

We can insert code that is never executed. This transformation is very useful as we can use any combination of instructions within the dead code block. Below is an example of dead code insertion.

Original Code	**Transformed Code**

MOV AL, BL
ADD AL, 05H
	MOV AL, BL
	JMP LOC:
	PUSH AX
	POP AX
	ADD AL, BL
	LOC: ADD AL, 05H

2.2.2.3 Equivalent Code Substitution

We can transform the code by replacing instruction(s) with equivalent instructions. Below are some examples of equivalent code substitution

Original Code	**Transformed Code**
ADD AL, 05H	ADD AL, 04H ADD AL, 01H
MOV AX, BX	PUSH AX POP BX

2.3 Hidden Markov Model

Markov model is a statistical model in which there are states and the probabilities of state transitions are known [20]. In Markov models, the states are visible to the observer. A hidden Markov model differs from a Markov model in this respect. The states in a hidden Markov model are not visible to the observer, but the output, which is dependent on the state, is visible [22]. Therefore, hidden Markov models have state transition probabilities as well as a probability distribution over all possible output symbols in each state. Formally we can describe an HMM model λ using the following parameters.

T: the length of the observation sequence

N: the number of states in the model

M: the number of observation symbols

$X = \{x_0, x_1, \ldots, x_{N-1}\}$: the states of the Markov process

$O = \{O_0, O_1, \ldots, O_{M-1}\}$: set of possible observations

A: the state transition probabilities

B: the observation probability matrix

π: the initial state distribution

Figure 3 shows an example of HMM where X_0, X_1, .., X_{T-1} are the hidden states and O_0, O_1, .., O_{T-1} are the observed symbols in each state. A and B represent the state transition probabilities and observation probabilities respectively.

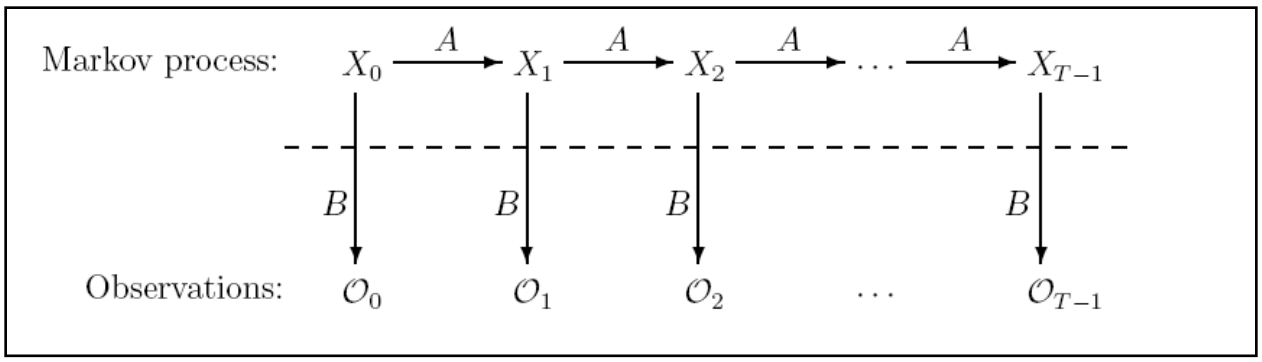

Figure 3: Hidden Markov Model

To understand how we can model a system as a hidden Markov model, consider the following example [21]. A person has three coins and he is in a room tossing them in some sequence. The room is closed and what we are shown is only the outcomes of his tossing, say, *{T,T,H,T,H,H,T,T,..}.* This will be called observation sequence. We do not know the sequence in which he is tossing the different coins, and also the bias of the various coins. We can see that the output sequence depends on (i) the individual bias, (ii) transition probabilities between various states, and (iii) which state is chosen to begin the observations. The above three parameters characterize the hidden Markov model for this coin tossing experiment.

HMMs are used in many applications like speech recognition, alignment of bio sequences, and cryptanalysis. HMMs are very useful in these applications as following three problems associated with HMMs can be solved very efficiently:

1. For a hidden Markov model and a sequence of observations, find the likelihood that the observed sequence can be generated by the process that is modeled.

2. For a hidden Markov model and an observation sequence, find an optimal sequence of states in the model that can generate the observation sequence.

3. For an observation sequence, known number of states, and known number of observation symbols, find the model that maximizes the probability of observing the given sequence.

In this project, we require the algorithms for problem (1) and (3) above. The solution for (3) is used for training the model and the solution for (1) is used for detecting if a piece of code has been watermarked by our system or not. We explain the details in the next two sections.

3. Design Overview

The goal of this project is to design a robust watermarking scheme. The requirements for our system can be understood by considering the following scenario. We have some software that we want to protect from unauthorized copying. We watermark our software and sell it to some person, say Trudy. She makes an illegal copy of this software and tries to destroy the watermark in the copy. Our watermarking scheme should be able to detect that the copy made by Trudy was made from our software

3.1 System Overview

Our system has two phases. In the first phase, we use a metamorphic generator to create morphed copies of the software to be protected from piracy. We use the morphed copies for training and generating a hidden Markov model. The learnt model acts as the watermark. The use of morphed copies ensures that the watermark is more robust, i.e., more resistant to attacks. Another advantage of using morphed software is that it increases the diversity and therefore makes it difficult to attack any vulnerability in the software. Figure 4 shows the training phase of our system.

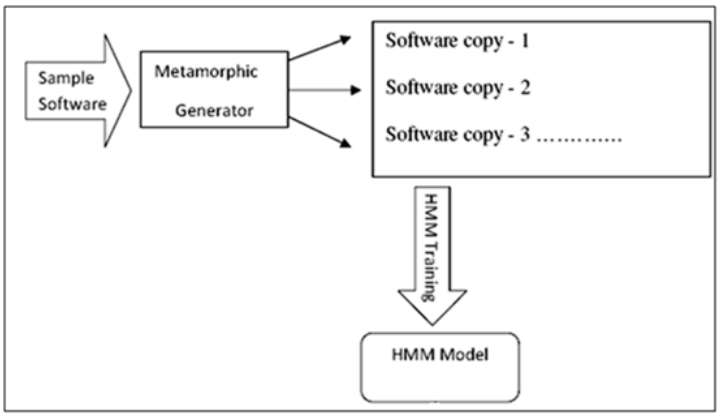

Figure 4: Training Phase

The second phase is the detection phase. In this phase we find out the likelihood that given software belongs to the hidden Markov model generated in the first phase. A high score indicates that the software is a modified copy of the morphed copies used for training. Figure 5 shows the detection phase.

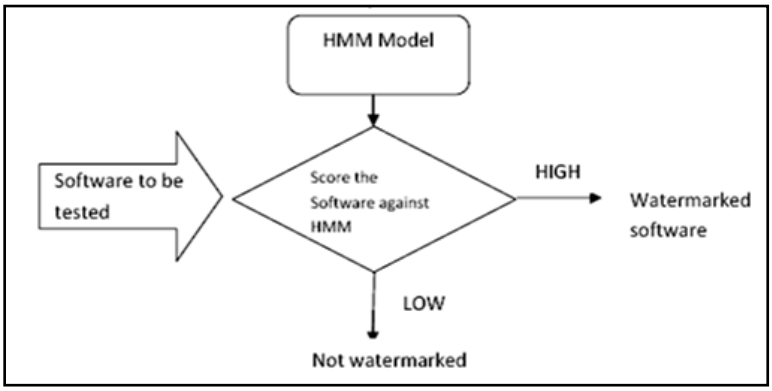

Figure 5: Detection Phase

3.1.1 Design of Metamorphic Generator

Our metamorphic generator makes morphed copies from given software. For generating these morphed copies, various techniques like the dead code insertion and the equivalent code substitution transformations (explained in Section 2) can be used. The operation of the metamorphic generator is driven by three parameters that a user has to set. The first parameter is the name of the file to be morphed. The second parameter is the amount of morphing to be done (in percentage). This number indicates by what percentage of the number of lines of code in the original software, the number of lines of code in the morphed copy can increase. For example, if the number of lines in original .asm file is 1000 and we want to perform 20% of morphing, then after morphing, the maximum number of lines of code in a morphed copy will be 1200. The third parameter is the number of copies to be generated. Figure 6 illustrates the design of the metamorphic generator.

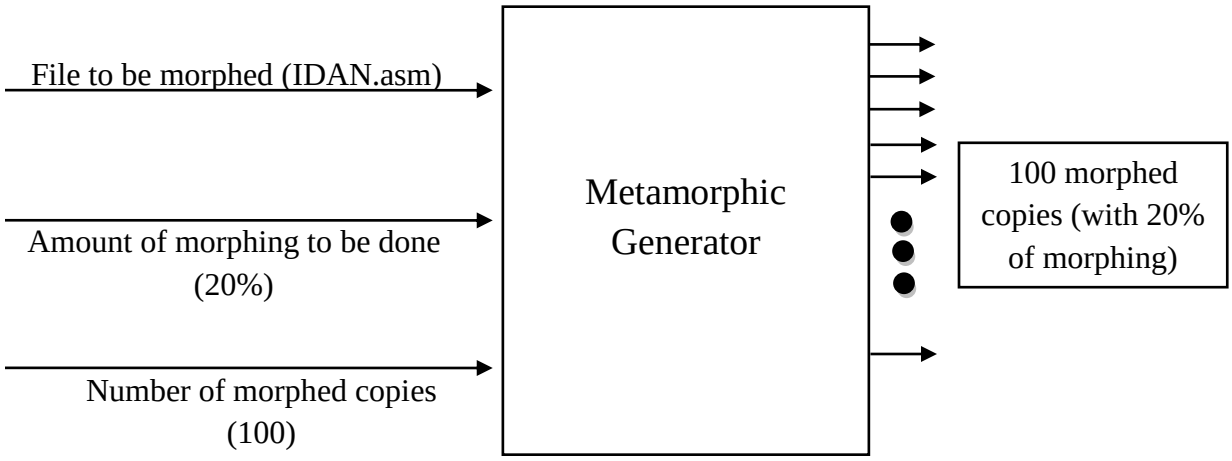

Figure 6: Metamorphic Generator

3.1.2 Robustness of HMM Based Watermarking Scheme

In our system, the hidden Markov model that is learnt in the training phase is the watermark. The files used for training can be considered to be watermarked. This is actually a very unusual way to watermark, since the watermark is not anything that we add to the code. To read the watermark, an HMM will be used and it will score the software. If it scores high, then the watermark was created with our metamorphic generator, and if not, it was not created by the generator. From here on we will use the term "normal file" for any software that has not been created by our metamorphic generator and hence is scored low by the HMM. Details of HMM training and how HMM scores the software are explained in Section 4. Here we are developing a model so that we can detect our software after it has been tampered with by an attacker (who is, presumably, trying to remove or damage the "watermark".).

If someone wants to remove the watermark from the software, they have to trick the HMM. So for this, the attacker has to tamper the watermarked file such that it will start looking like a

normal file. One possible attack that can be carried out is to insert some amount of dead code from normal file in the morphed (watermarked) copy. If our HMM model scores this modified (tampered) file closer to normal file, then the attack can be considered to be successful. Previous research has shown that it is very difficult to trick such an HMM and, therefore, this watermarking scheme is likely to be robust. We have tested our watermarking scheme to verify that this is the case and have presented the results in Section 5.

4. Implementation

In this section we describe how we have implemented each component in our system.

4.1. Metamorphic Generator

We have implemented metamorphic generator as a Perl script. We have used dead code insertion and insertion of code that does not affect the control flow and data flow of the program to transform the input assembly code. The metamorphic generator takes as input the name of the file to be morphed, the percentage of morphing, and the number of morphed copies to be generated. For example, we can give the file name as "Sample.asm", percentage of morphing as "20%", and number of copies as "100". We provide some files containing different number of instructions (J1.asm, J2.asm, …, J5.asm) to the metamorphic generator. The code in these files can be used as dead code. Following are the steps performed in the metamorphic generator for each of the morphed copies to be generated.

a. Compute the number of lines of code in the original file
b. Compute the number of lines of code that can be inserted, say *icount*, based on the percentage of morphing and the number computed in (a).
c. Select 5 random locations where dead code is to be inserted in the morphed copy.
d. Make a copy of the original ".asm" file.
e. If *icount* is less than the number of lines of code in J1.asm then insert *icount* instructions from J1.asm at the first random location selected in (c) and stop.
f. If *icount* is more than the number of lines of code in J1.asm then insert all the instructions in J1.asm at the first random location selected in (c), decrement *icount* by the number of lines in J1.asm and repeat steps (e) and (f) using the next random location and the next junk file in the list.

We want to make it difficult for the attacker to identify any transformations that we have done. So add additional code in each morphed copy that preserves the semantics but makes the file structurally even more different. For this purpose, we insert the five code blocks shown in Table 2 at random locations in the morphed copies. To avoid detection we insert these code blocks in very small percentage of locations. We do this by generating a random number between 1 and 800 for each line of code in the morphed copy. If the number is 1, we insert the first block shown in the Table 2 before the current instruction being considered in the morphed copy. If it is 2, then we select the second block, and so on.

NOP
JMP LOC_100 LOC_100:
PUSH EAX POP EAX
ADD EAX, 0H
SUB EAX, 0H

Table 2: Instruction Blocks for Increasing Diversity

4.2 Training HMM

We collect all morphed files and use them as *dataset* for training HMM. For training and testing we used cross-validation techniques [4]. Cross-validation is a technique for assessing how the results of a statistical analysis will generalize to an independent data set. It is used where one wants to estimate how accurately a predictive model will perform in future. One round of cross-validation involves partitioning a sample of data into complementary subsets, performing the

analysis on one subset (training set), and validating the analysis on the other subset (testing set) . To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are averaged over the rounds.

We use 5-fold cross-validation. For training a model, each time, we select one of the subsets as test data and remaining four subsets as training data for HMM. We train our model using the assembly opcode sequences of the morphed files. For this we extract the sequences of opcodes from each morphed copy to be used for training. We concatenated the opcode sequences to yield one long observation sequence [4]. We used test data along with normal files for scoring purpose. This process was repeated five times. Each time, the test data subset was changed. In this way we got five HMM models. We see that the scores for the test data files are high while those for the normal files are low. We use the highest score amongst normal files as a threshold. Figure 7 shows the training phase.

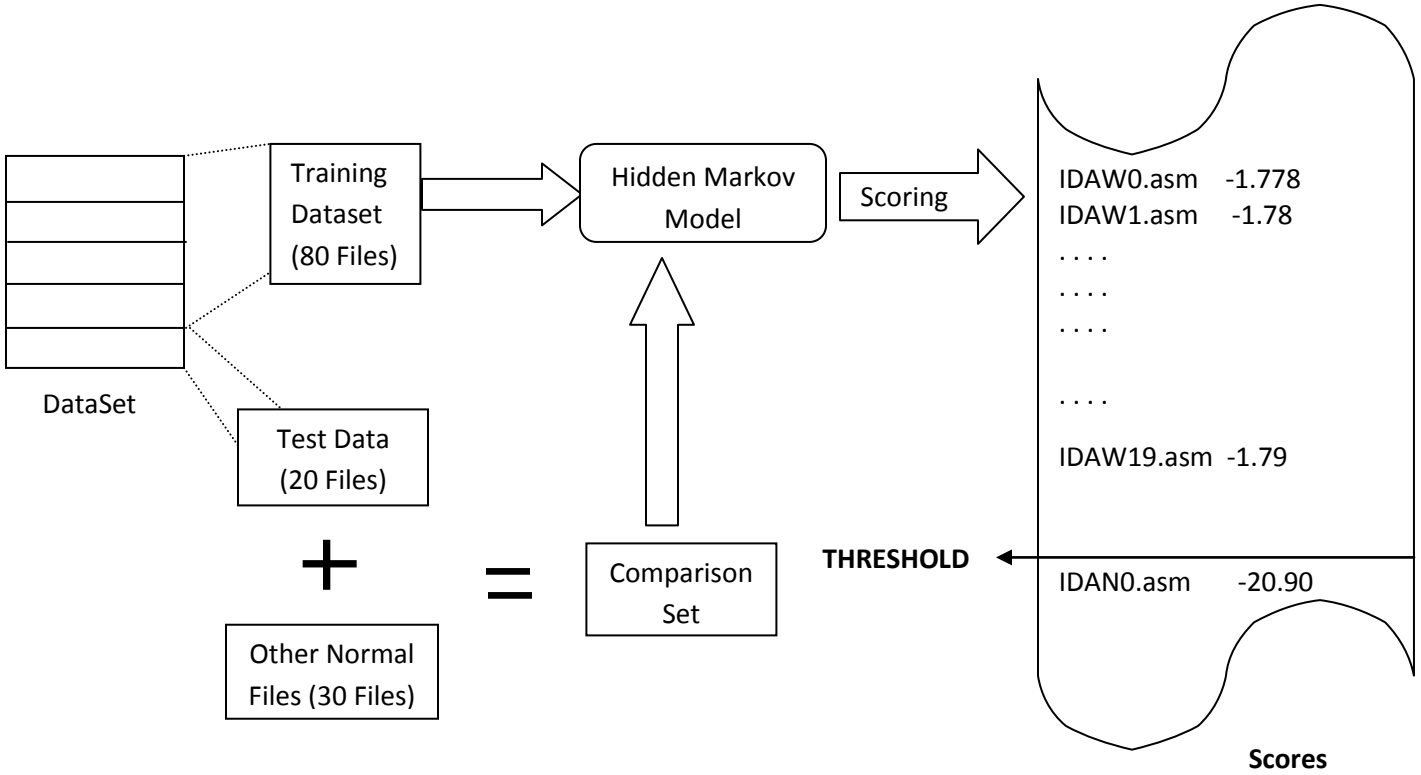

Figure 7: HMM Training

In the detection phase, we score the file provided to us. If the score is higher than the threshold, it means that the file is a watermarked file. If the score is higher than the threshold, but closer to the normal files, that means that it is a copy of a watermarked file that has been tampered.

5. Evaluation

We evaluated the robustness of our watermarking using on many experiments. The test data that we used for our experiments consisted of 30 executables from Cygwin version 1.5.19 [4]. These executables were disassembled using the IDA Pro Disassembler version 4.6.0. These 30 Cygwin utilities files were named N0.EXE to N39.EXE. We added the prefix "IDA" to the respective file names and changed the suffix to ".ASM" from ".EXE" to denote that the files were disassembled ASM files created by IDA Pro. For example, the file disassembled from N0.EXE was named IDAN0.ASM. We used another executable from Cygwin version 1.5.19 as the input code to be protected. We named the disassembled version of the file as IDAW.ASM. We named the 100 morphed copies that we generate in each for experiments as IDAW0.ASM to IDAW100.ASM

5.1 Tampering Using Dead Code Insertion

To evaluate the ability of our watermarking scheme to withstand distortion attacks, we developed a tampering scheme to modify the morphed (watermarked) copies. The key idea here is to make the tampered file look similar to a normal file so that the watermark is rendered undetectable. We used our metamorphic generator for tampering the watermarked file. We copied instructions from normal files and inserted them as dead code in the watermarked files. We called such modified files as "tampered" files and named them as "IDAT0.ASM", "IDAT1.ASM" and so on.

As more and more code is inserted from the normal file, the tampered file becomes similar to normal file. The score of the tampered file moves closer to normal files within the increase in tampering and beyond certain point the score becomes less than the threshold. We fail to detect the watermark in such cases. We evaluated the ability of our watermarking scheme to withstand such an attack.

In the first experiment, we used 0% morphing. This means that all the morphed files were the same. We generated 100 morphed files named IDAW0.ASM to IDAW99.ASM. We used 5-fold cross validation technique, i.e., 80 files were used for training the model and 20 for generating testing. We used the 30 normal files (IDAN0.ASM - IDAN29.ASM) for obtaining the threshold. We generated tampered files using just dead code insertion as explained above. We varied the percentage of additional code that is added from 2% to 90%. Figure 8 shows the scores of the tampered files for 2%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, and 90% additional code. We can clearly see in the graph that the normal files have a very low score while the watermarked files have a very high score. Since, we are doing 0% morphing, all the morphed files have the same score. For 2% tampered file the score is very close to the watermarked files. As the tampering is increased, the scores become closer to the normal files. We can still detect the files as being tampered versions of the watermarked files as the scores are more than the normal file till 60% tampering. After 70% tampering, the files become indistinguishable from the normal files.

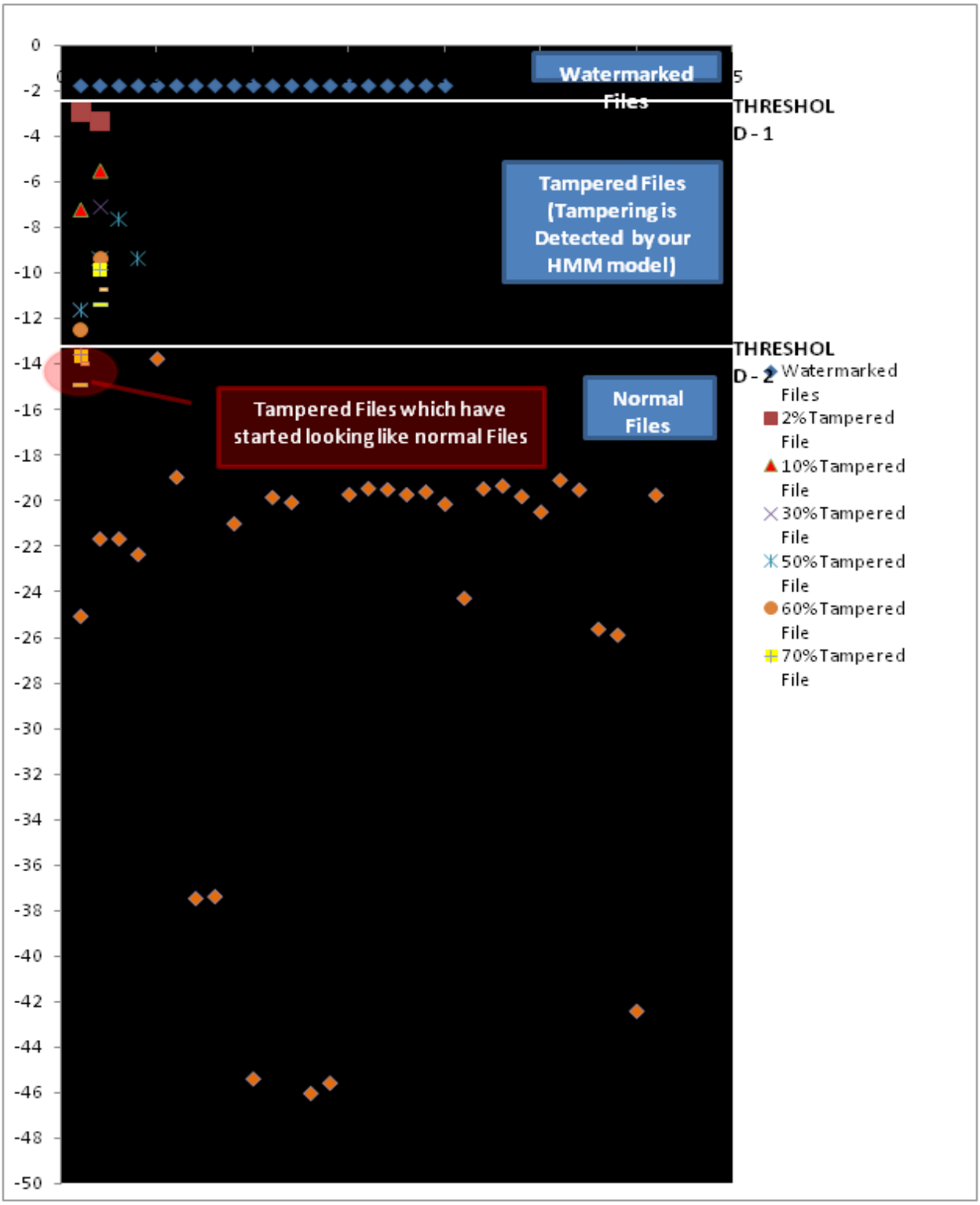

Figure 8: Detecting Tampering (Dead Code Insertion) using 0% Morphing

Table 3 shows the scores of some watermarked, tampered and normal files. Table 4 shows the maximum, minimum, and the average scores for these files.

1	Watermarked Files (0% Morphing)		23	Tampered Files (2%-90%)		49	Normal Files	
2	IDAW0	-1.77829	24	IDAT1	-1.77829	50	IDAN0	-25.118
3	IDAW1	-1.77829	25	IDAT11	-1.77829	51	IDAN1	-21.7181
4	IDAW2	-1.77829	26	IDAT9	-2.35854	52	IDAN2	-21.7181
5	IDAW3	-1.77829	27	IDAT15	-2.39115	53	IDAN3	-22.401
6	IDAW4	-1.77829	28	IDAT4	-2.97128	54	IDAN4	-13.8061
7	IDAW5	-1.77829	29	IDAT18	-3.296	55	IDAN5	-19.0073
8	IDAW6	-1.77829	30	IDAT3	-3.66833	56	IDAN6	-37.5169
9	IDAW7	-1.77829	31	IDAT12	-4.20353	57	IDAN7	-37.4333
10	IDAW8	-1.77829	32	IDAT13	-4.25444	58	IDAN8	-21.0517
11	IDAW9	-1.77829	33	IDAT2	-4.77143	59	IDAN9	-45.443
12	IDAW10	-1.77829	34	IDAT6	-5.37998	60	IDAN10	-19.8953
13	IDAW11	-1.77829	35	IDAT5	-5.90857	61	IDAN11	-20.1161
14	IDAW12	-1.77829	36	IDAT8	-5.91894	62	IDAN12	-46.0812
15	IDAW13	-1.77829	37	IDAT14	-6.27577	63	IDAN13	-45.6286
16	IDAW14	-1.77829	38	IDAT7	-8.286	64	IDAN14	-19.7677
17	IDAW15	-1.77829	39	IDAT0	-10.1517	65	IDAN15	-19.5055
18	IDAW16	-1.77829	40	IDAT10	-10.7101	66	IDAN16	-19.5503
19	IDAW17	-1.77829	41	IDAT22	-15.0969	67	IDAN17	-19.7725
20	IDAW18	-1.77829	42	IDAT19	-15.248	68	IDAN18	-19.6513
21	IDAW19	-1.77829	43	IDAT20	-15.3831	69	IDAN19	-20.19

Table 3: Sample Scores of Watermarked, Tampered, and Normal Files (0 % Morphing)

1	Watermarked Files (10% Morphing)		23	Tampered Files (2%-90%)		49	Normal Files	
2	IDAN0	-1.875424093	24	IDAR0	-1.795316795	50	IDAR10	-22.43776683
3	IDAN1	-1.894162628	25	IDAR1	-1.800754499	51	IDAR11	-14.97724653
4	IDAN2	-1.875497311	26	IDAR2	-1.815625475	52	IDAR12	-14.97724653
5	IDAN3	-1.878042727	27	IDAR3	-1.854477792	53	IDAR13	-21.11368146
6	IDAN4	-1.87885201	28	IDAR4	-1.87358674	54	IDAR14	-10.74107218
7	IDAN5	-1.875691427	29	IDAR5	-2.323843809	55	IDAR15	-17.29351763
8	IDAN6	-1.876453179	30	IDAR6	-2.439801625	56	IDAR16	-27.75389289
9	IDAN7	-2.374626063	31	IDAR7	-3.136182652	57	IDAR17	-28.4030914
10	IDAN8	-2.365077653	32	IDAR8	-3.801740907	58	IDAR18	-16.18621466
11	IDAN9	-1.889510832	33	IDAR9	-3.995460827	59	IDAR19	-41.89153923
12	IDAN10	-1.885152854	34	IDAR10	-4.175567442	60	IDAR20	-18.66338825
13	IDAN11	-1.871452212	35	IDAR11	-4.188493634	61	IDAR21	-14.82174246
14	IDAN12	-1.874302117	36	IDAR12	-4.771510208	62	IDAR22	-41.63772885
15	IDAN13	-1.87457392	37	IDAR13	-4.778173453	63	IDAR23	-42.37592219
16	IDAN14	-1.889280041	38	IDAR14	-8.728824252	64	IDAR24	-14.62062856
17	IDAN15	-1.87895805	39	IDAR15	-10.3883321	65	IDAR25	-14.34505669
18	IDAN16	-1.884413815	40	IDAR16	-10.71233973	66	IDAR26	-14.42036049
19	IDAN17	-1.864063183	41	IDAR17	-11.36451828	67	IDAR27	-14.51835707
20	IDAN18	-1.875355547	42	IDAR18	-12.40486271	68	IDAR28	-14.53510193
21	IDAN19	-1.86300812	43	IDAR19	-12.65003951	69	IDAR29	-14.94528756

Table 4: Sample Scores of Watermarked, Tampered, and Normal Files (10% Morphing)

	Maximum Score	**Minimum Score**	**Average**
Watermarked File	-1.77829	-1.77829	-1.77829
Tampered Files	-2.9506	-14.0354	-16.986
Normal Files	-13.806	-46.081	-59.887

Table 5: Max, Min, Average Scores

	Maximum Score	**Minimum Score**	**Average**
Watermarked Files	-1.86300812	-2.374626063	- 2.1188170915
Tampered Files	-3.325264237	-11.40799065	- 7.3666274435
Normal Files	-13.806	-46.081	-59.887

Table 6: Max, Min, Average Scores (10% Morphing)

Figure 9 shows the size of the tampered files in term of the lines of code.

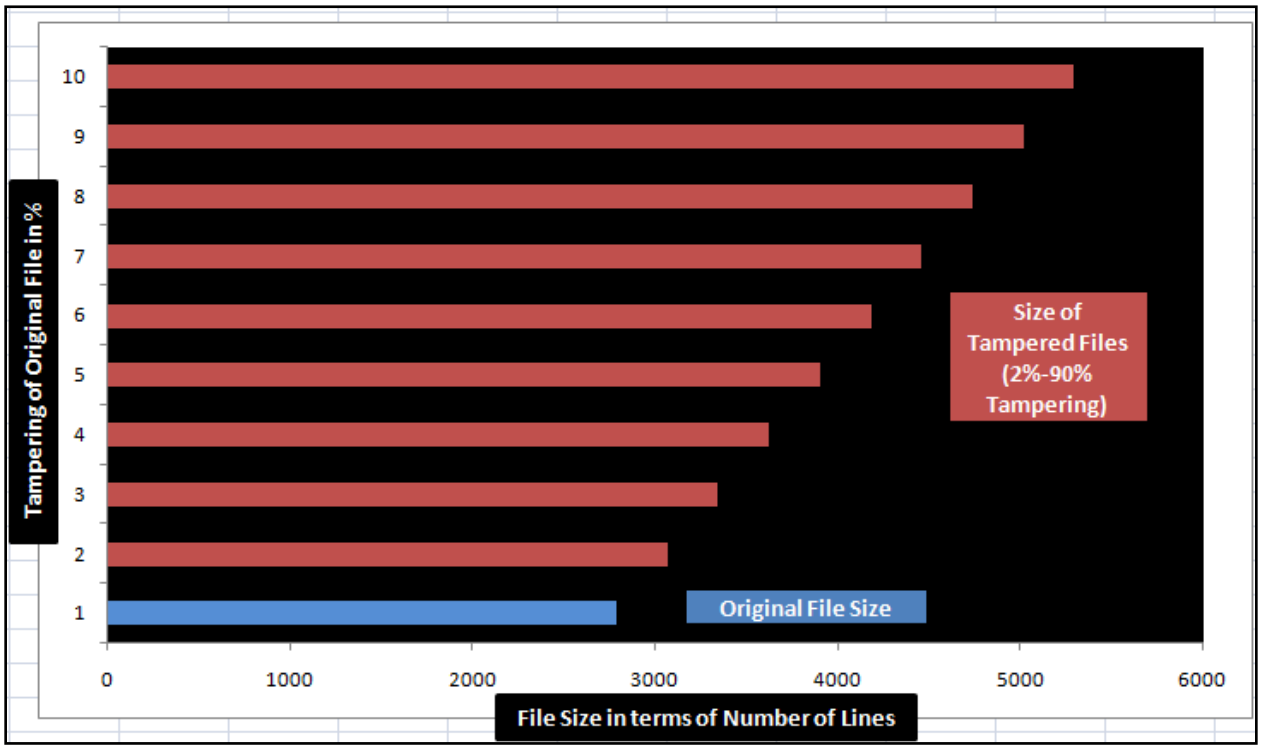

Figure 9: Sizes of Tampered Files and Original File

For the second experiment, we used 10% morphing in the metamorphic generator. Figure 10 shows the results for tampering from 2%-90%. Here we can see that see the scores for the morphed copies vary slightly. Using morphing the watermarking scheme becomes more robust and is able to detect even 70% tampered files. But the files that are tampered more than 80% can not be detected.

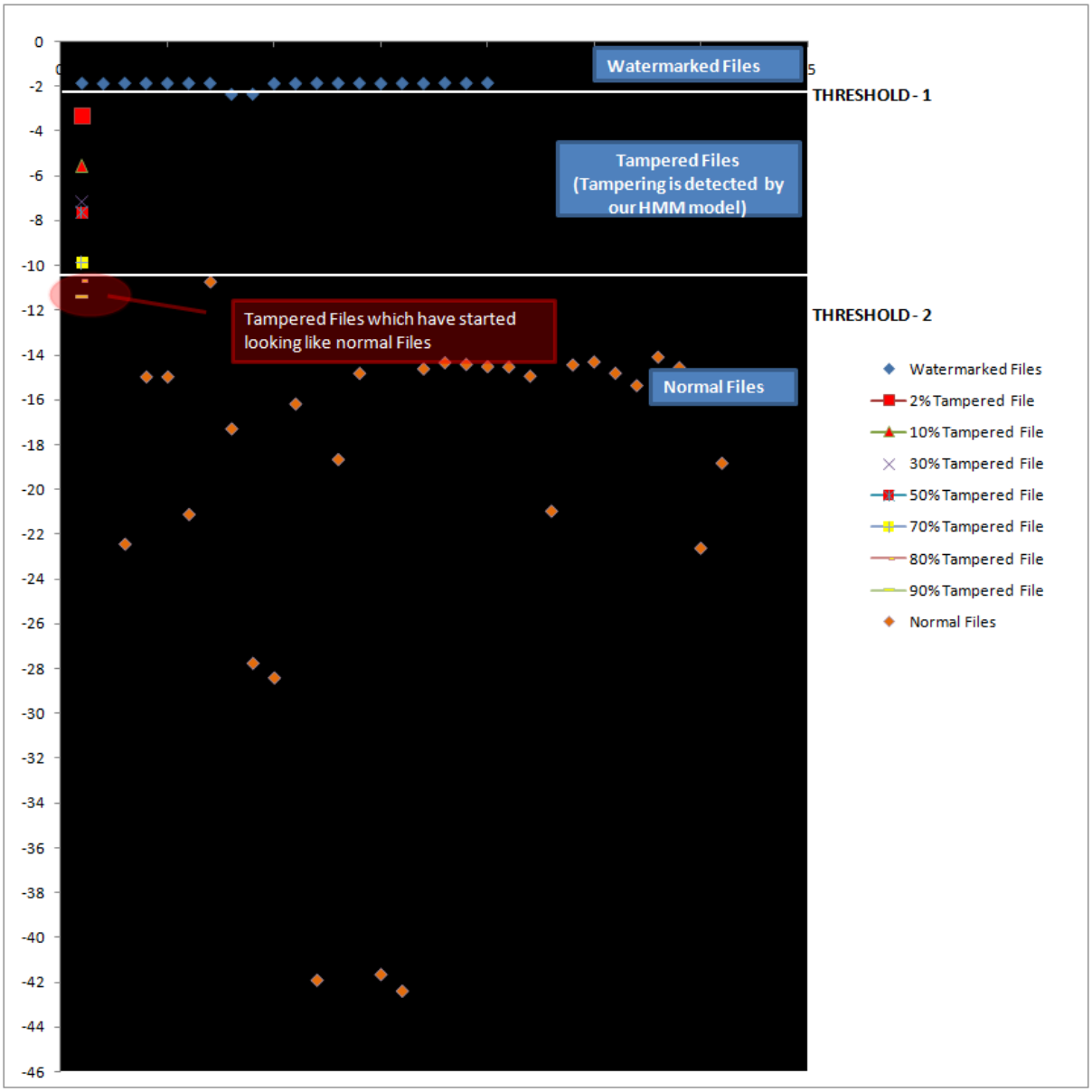

Figure 10: Detecting Tampering (Dead Code Insertion) using 10% Morphing

Table 7 summarizes the results for 0% and 10% morphing.

Morphing (in %)	Tampering (in %)	Is tampering detected? (Yes/No)	Morphing (in %)	Tampering (in %)	Is tampering detected? (Yes/No)
0%	2%	Yes	10%	2%	Yes
0%	5%	Yes	10%	5%	Yes
0%	10%	Yes	10%	10%	Yes
0%	20%	Yes	10%	20%	Yes
0%	30%	Yes	10%	30%	Yes
0%	40%	Yes	10%	40%	Yes
0%	50%	Yes	10%	50%	Yes
0%	60%	Yes	10%	60%	Yes
0%	70%	Yes	10%	70%	Yes
0%	80%	No	10%	80%	Yes
0%	90%	No	10%	90%	No

Table 7: Detection Results for 0% and 10% Morphing

5.2 Tampering Using Dead Code Insertion and Code Substitution

We performed another experiment to evaluate the robustness of our watermarking scheme against distortion attacks. We used equivalent code substitution along with dead code insertion to tamper the files. The equivalent code substitution is as explained in Section 2. Here the percentage of tampering indicates both the amount of dead code inserted and the percentage of code substituted. Figure 11 shows the results for 0% morphed files and tampered files from 2% - 90%. Once gain we see that we are able to detect 2% - 60% tampered files.

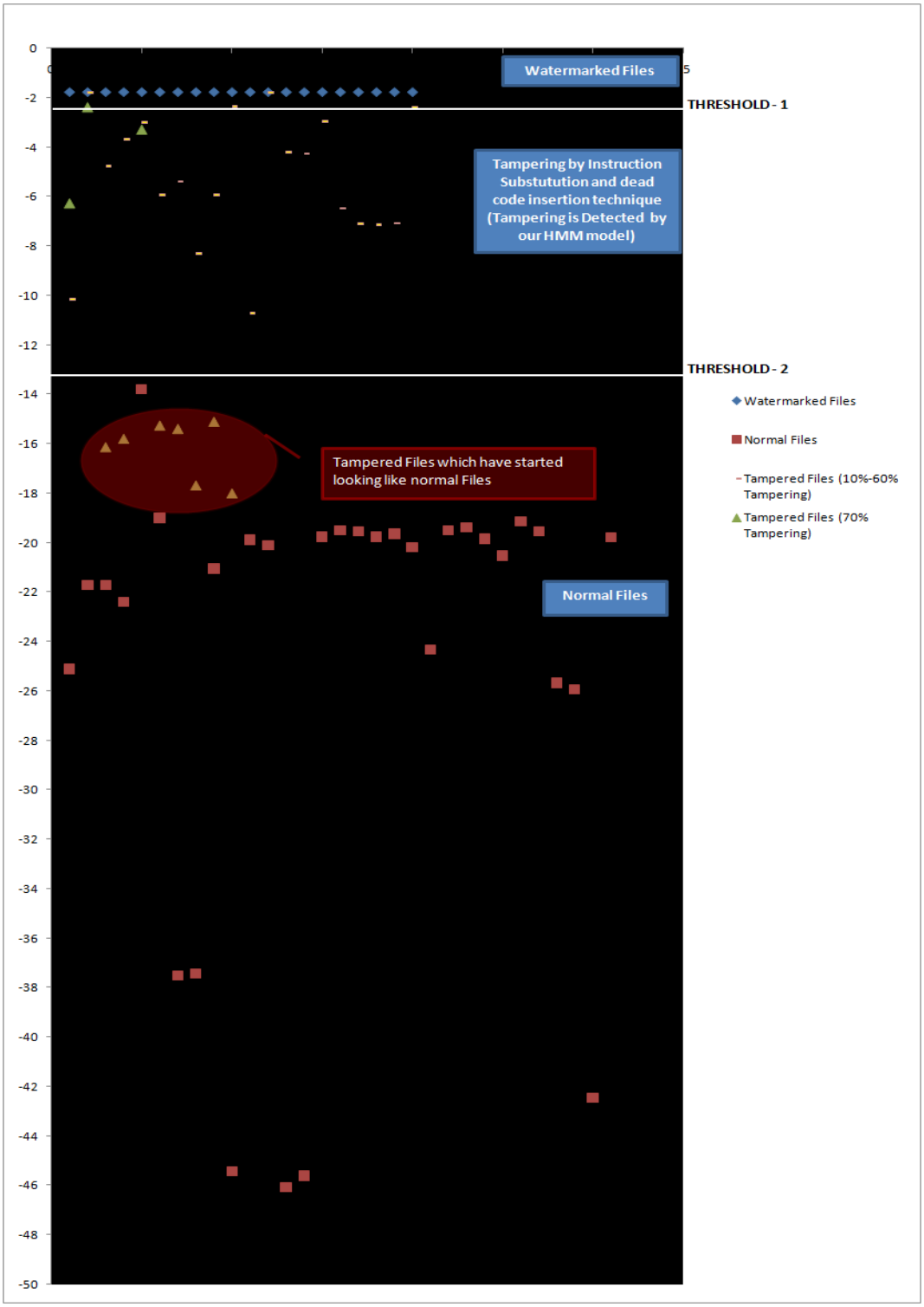

Figure 11: Detecting Tampering (Dead Code Insertion and Code Substitution) using 0% Morphing.

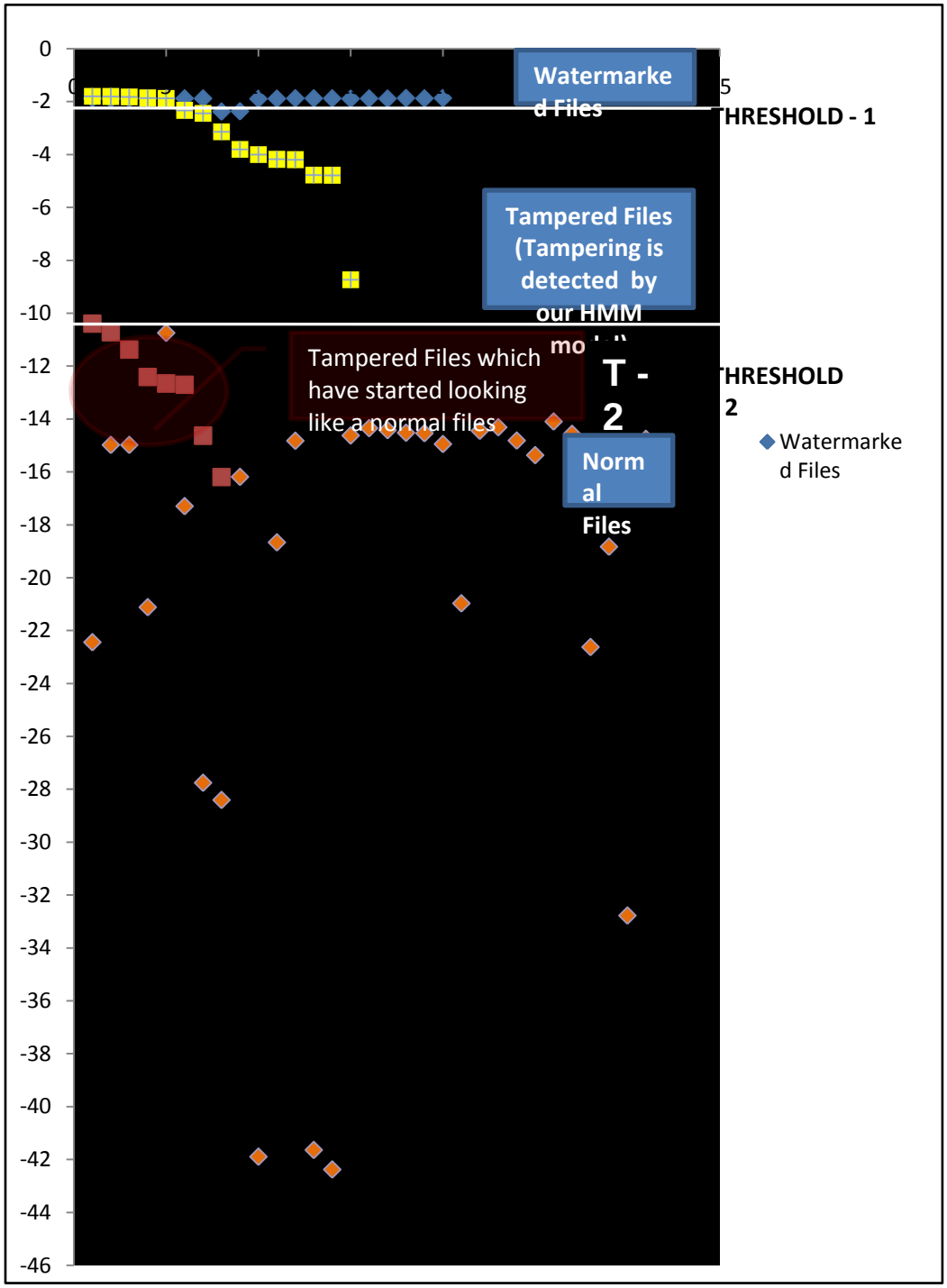

Figure 12: Detecting Tampering (Dead Code Insertion and Code Substitution) using 10% Morphing.

6. Conclusion and Future Work

Developing a robust watermarking scheme is a very challenging problem. No single watermarking scheme can prevent all types of attacks [23]. In this project we developed a watermarking scheme inspired by the success of previous research in identifying metamorphic virus using statistical methods. Our experimental results show that our scheme is robust and can withstand certain kinds of attacks very well.

We believe that this work is just the beginning and a lot more research needs to be done in this area. In the future, we plan to investigate using more sophisticated techniques for metamorphic generator and evaluate its effect on the robustness of the scheme. Also, we plan to address more attacks in the future. And finally, we want to extend the ideas in this work by developing meta-metamorphic generator. A meta-metamorphic generator would be able to generate new metamorphic generators any time that some software needs to be watermarked. Such a scheme can potentially be more robust than our current scheme.

References

[1] BSA/IDC, "Seventh Annual Piracy Report", 2009, Available at http://portal.bsa.org/piracyimpact2010/studies/piracyimpactstudy2010.pdf.

[2] G. Cronin, "A Taxonomy of Methods for Software Piracy Prevention", Department of Computer Science, University of Auckland, New Zealand, Tech Rep., 2002.

[3] "Digital Watermarking", Available at http://en.wikipedia.org/wiki/Digital_watermarking.

[4] Wing Wong, "Analysis and Detection of Metamorphic Computer Viruses", Master's thesis, San Jose State University, 2006, Available at http://www.cs.sjsu.edu/faculty/stamp/students/Report.pdf.

[5] Julien P. Stern and Gaël Hachez and François Koeune and Jean-Jacques Quisquater and Ucl Crypto Group and Batiment Maxwell and Place Du Levant, "Robust Object Watermarking: Application to Code", Information Hiding, 2000.

[6] R. Chandramouli, Nasir Memon, Majid Rabbani, "Digital Watermarking", Encyclopedia of Imaging Science and Technology, 2002.

[7] Robert L. Davidson and Nathan Myhrvold, "Method and system for generating and auditing a signature for a computer program", US Patent 5,559,884, Assignee: Microsoft Corporation, 1996.

[8] Scott A. Moskowitz and Marc Cooperman, "Method for stega-cipher protection of computer code", US Patent 5,745,569, Assignee: The Dice Company, 1996.

[9] David Nagy-Farkas, "The easter egg archive", 1998. Available at http://www.eeggs.com/lr.html.

[10] Christian Collberg and Clark Thomborson, "Software Watermarking: Models and Dynamic Embeddings", 1999.

[11] SECURE DIGITAL MUSIC INITIATIVE. http://www.sdmi.org.

[12] SECURE DIGITAL MUSIC INITIATIVE. SDMI public challenge, Sept. 2000. http://www.hacksdmi.org.

[13] STEIGLITZ, K. "A Digital Signal Processing Primer: with Applications to Digital Audio and Computer", Music. AddisonWesley, 1996.

[14] Scott A. Craver, Min Wu, Bede Liu, Adam Stubblefield, Ben Swartzlander, Dan S. Wallach, Edward W. Felten, "Reading Between the Lines: Lessons from the SDMI Challenge", 10th USENIX Security Symposium, 2001.

[15] Mark Stamp, “Risks of Monoculture”, Inside Risks Columns, 2004.

[16] Smita Thaker, “Software Watermarking via Assembly Code Transformations”, Master’s Thesis, San Jose State University, 2004.

[17] Kip R. Irvine, “Assembly Language for x86 Processors”, Florida International University School of Computing and Information Sciences, 2007.

[18] Intel, “IA-32 Architectures Software Developer’s Manuals”, Available at http://www.intel.com/products/processor/manuals/index.htm.

[19] Da. Lin, “Hunting for Undetectable Metamorphic Viruses”, Master’s Thesis, San Jose State University, December 2009.

[20] Mark Stamp, “A Revealing Introduction to Hidden Markov Models”, 2004.

[21] Lawrence R. Rabiner, “A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition”, 1989.

[22] “Hidden Markov model”, Available at http://en.wikipedia.org/wiki/Hidden_Markov_model The Watermark Copy Attack

[23] W. Bender, D. Gruhl, N. Morimoto, and A. Lu, “Techniques for data hiding”, IBM Systems Journal, 1996.