T04424

SOFTWARE THUMBPRINTING VIA IL CODE TRANSFORMATIONS

Developed By:

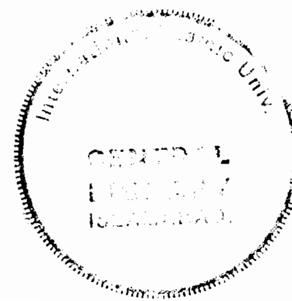
Ahmed Hasan
Muhammad Shakeel Anjum

Supervised By:

Prof. Dr. Khalid Rashid

DEPARTMENT OF COMPUTER SCIENCE
FACULTY OF BASIC & APPLIED SCIENCES
INTERNATIONAL ISLAMIC UNIVERSITY, ISLAMABAD

2008



MS
005.276
AHS

1- Microsoft . Net
2- Web Site development
3- Internet Programming

M. ul

MS M.d

17-07-10

Accession No TH-4424
AHS

DATA ENTERED
16/4/2012

International Islamic University, Islamabad
Faculty of Basic & Applied Sciences
Department of Computer Science

Dated: 09/15/04/ 2008

FINAL APPROVAL

It is certified that we have read the thesis, entitled "Software Thumbprinting VIA IL Code Transformations", submitted by Mr. Ahmed Hasan 44-FAS/MSSE/F04 and Mr. Muhammad Shakeel Anjum 49-FAS/MSSE/F04. It is our judgment that this thesis is of sufficient standard to warrant its acceptance by the International Islamic University Islamabad for the award of MS degree in Software Engineering.

PROJECT EVALUATION COMMITTEE:

External Examiner

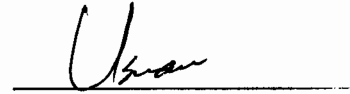
Dr. Muhammad Ansari

Associate Professor,
Department of Computer Science,
Federal Urdu University,
Islamabad, Pakistan.

Internal Examiner

Ms. Muhammad Usman

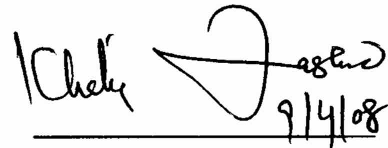
Lecturer,
Department of Computer Science,
International Islamic University,
Islamabad, Pakistan.



Supervisor

Prof. Dr. Khalid Rashid

Former Dean,
Faculty of Basic & Applied Sciences,
International Islamic University,
Islamabad, Pakistan.



In the Name of

ALLAH

The Most Merciful
The Most Beneficent

A Thesis Submitted to the Department of Computer Science, Faculty of Basic & Applied Sciences, International Islamic University, Islamabad, Pakistan, as a Partial Fulfillment of the Requirements for the Award of the Degree of

MS in Software Engineering

To
The Holiest man ever born,
PROPHET MUHAMMAD (PEACE BE UPON HIM)
&
OUR DEAREST PARENTS & FAMILY
Who are an embodiment of diligence and honesty,
Without their prays and support
This dream could have never come true
&
PRECIOUS FRIENDSHIP
That made us laugh, held us when we cried
And always stood by us

DECLARATION

We hereby declare and affirm that this thesis neither as a whole nor as part thereof has been copied out from any source. It is further declared that we have completed this thesis and accompanied software application entirely on the basis of our personal efforts, made under the sincere guidance of our supervisor. If any part of this report is proven to be copied out or found to be a reproduction of some other, we shall stand by the consequences. No portion of the work presented in this report has been submitted in support of an application for other degree or qualification of this or any other University or Institute of learning.

Ahmed Hasan
44-FAS/MSSE/F04

Muhammad Shakeel Anjum
49-FAS/MSSE/F04

ACKNOWLEDGEMENTS

We bestow all praise, acclamation and appreciation to Almighty Allah, The Most Merciful and Compassionate, The Most Gracious and Beneficent, Whose bounteous blessings enabled us to pursue and perceive higher ideals of life, who bestowed us good health, courage and knowledge to carry out and complete our work. Special thanks to His Holy Prophet Muhammad (SAW) who enabled us to recognize our Lord and Creator and brought us the real source of knowledge from Allah, the Qur'an, and who is the role model for us in every aspect of life.

We consider it a proud privilege to express our deepest gratitude and grand tribute to our supervisor **Dr. Khalid Rashid**, who kept our morale high by his suggestions and appreciation. His motivation leads us to this success and without his sincere guidance and cooperative nature we could have never complete this task.

It would not be out of place to express our profound admiration to **Dr. Hafiz Farooq Ahmad** for his dedication, inspiring attitude, untiring help, and kind behavior through out the project efforts.

Finally we must mention that it was mainly due to our family's moral support during our entire academic career that enabled us to complete our work dedicatedly. We once again would like to admit that we owe all our achievements to our most loving parents, who mean most to us, for their prayers are more precious then any treasure on the earth. We are also thankful to our truly, sincere and most loving brothers, sisters, friends and class fellows who mean the most to us, and whose prayers have always been a source of determination for us.

Ahmed Hasan
44-FAS/MSSE/F04

Muhammad Shakeel Anjum
49-FAS/MSSE/F04

PROJECT IN BRIEF

Project Title:	Software Thumbprinting VIA IL Code Transformations
Organization:	International Islamic University, Islamabad, Pakistan.
Objective:	The objective of the project is to fulfill the degree requirement of MS in Software Engineering.
Undertaken By:	Ahmed Hasan Reg. No. 44-FAS/MSSE/F04 Muhammad Shakeel Anjum Reg. No. 49-FAS/MSSE/F04
Supervised By:	Prof. Dr. Khalid Rashid Department of Computer Science, Faculty of Basic & Applied Science, International Islamic University, Islamabad.
Started On:	June 2005
Completed On:	February 2008
Research Area:	Digital Rights Management, Software Protection, .NET, MSIL
Tools:	Visual Studio 2005, Ildasm, Ilasm, Dotfuscator Community Edition, Remotesoft .NET Explorer, Lutz Roeder .NET Reflector

ABSTRACT

Protection of software code against illegitimate use by its malicious users is a pressing issue for software industry today. Researchers have been devising technological countermeasures, both hardware-based and software-based, to resist against such user's attacks to violate software copy rights. Contemporary studies in software-based protections have proposed various techniques like watermarking, obfuscation and tamper-proofing. These schemes aim to protect software copy rights by restricting piracy, illegal reverse engineering and tampering. Many of these mechanisms are either too weak or too expensive to apply. Most of the technological implementation of these methodologies is found for Java applications. Where as, protecting .NET applications is rarely discussed. In this thesis, we present and explore a methodology that we believe can protect software copy rights, exclusively of .NET framework based applications. We have proposed a unique software watermarking technique called *software thumbprinting*, which in effect specializes the idea of *software fingerprinting*. We have demonstrated our proposed scheme to show how distinct thumbprints can be encoded into a .NET program through IL code transformations. Our evaluation shows that thumbprints encoded this way are more robust and more tamper-proofed. The proposed technique has good resistance against different types of malicious user's attacks. Moreover we aim to emphasize the need of developing better mechanism to protect .NET applications.

ACRONYMS & ABBREVIATIONS

ADT: Abstract Data Type

API: Application Programming Interface

BSA: Business Software Alliance

C#: C Sharp

CLR: Common Language Runtime

DB: Database

DLL: Dynamic Link Library

DRM: Digital Rights Management

EXE: Executable Program

ILASM: Intermediate Language Assembler

ILDASM: Intermediate Language Disassembler

JVM: Java virtual Machine

MSIL: Microsoft Intermediate Language or Intermediate Language (*IL*)

PPCT: Planted Planner Cubic Tree

TCPA: Trusted Computing Platform Alliance

VB: Visual Basic

XML: Extensible Markup Language

TABLE OF CONTENTS

Ch. No.	Contents	Page No.

1. INTRODUCTION 1

1.1 Software Protection 1
1.2 Software Piracy 2
1.3 Piracy of .NET Applications 3
1.4 Proactive Solutions 3
1.4.1 Hardware-based Protections 4
1.4.2 Software-based Protections 4
1.4.2.1 Software Watermarking 5
1.4.2.2 Software Fingerprinting 7
1.4.2.3 Tamper-proofing 8
1.4.2.4 Code Obfuscation 9
1.5 Overview of Proposed Methodology 10
1.6 Our Contributions 10
1.7 Thesis Organization 12

2. LITERATURE SURVEY AND PROBLEM DEFINITION 13

2.1 Literature Review 13
2.2 Problem Rationale 18
2.3 Software Piracy Problem Redefined 19
2.4 Problem Statement 22
2.5 Scope of Work 23

3. PROPOSED METHODOLOGY 24

3.1 Piracy Prevention Approaches 24
3.1.1 Piracy Education 25
3.1.2 Piracy Prevention 25
3.1.3 Piracy Detection 25
3.1.4 Piracy Assertion 25
3.2 Anti-Piracy Lines of Defense 26

3.3 Software Thumbprinting 27

3.3.1 Design of Proposed Methodology 28

3.3.1.1 Thumbprint Encoding 28

3.3.1.2 IL Code Transformations 29

3.3.1.3 Thumbprint Decoding 32

3.3.2 Formal Model of Software Thumbprinting 33

4. IMPLEMENTATION 36

4.1 Environment 36

4.1.1 .NET Platform 36

4.1.1.1 Programming Languages 36

4.1.1.2 Microsoft Intermediate Languages (MSIL) 37

4.1.2 Tools 37

4.1.2.1 IL Code Disassembler (*ildasm.exe*) 38

4.1.2.2 IL Code Assembler (*ilasm.exe*) 38

4.1.2.3 Marka 39

4.1.2.4 Dotfuscator 40

4.1.2.5 .NET Reflector 40

4.1.3 Techniques 41

4.2 Case Study: Tower of Hanoi 41

4.3 Issues regarding the Automation of Proposed Scheme........ 48

4.4 Limitation........ 49

5. RESULTS AND EVALUATION 50

5.1 Thumbprinting VS Prior Arts 50

5.2 Thumbprinting Thwarts .NET Reverse Engineering Tools 51

5.3 Robust and Tamper-proofed Thumbprints 51

5.4 Experimental Evaluation Framework 52

5.4.1 Evaluation Model 52

5.4.2 Evaluation Properties 54

5.5 Branch Function........ 55

5.6 Experimental Results 56

5.6.1 Approach 56

5.6.2 Results 56
5.6.2.1 Results before Implementation.......... 56
5.6.2.2 Results with Thumbprinting Scheme 57
5.6.2.3 Results with Branch Function Scheme 58
5.6.3 Results Analysis and Discussion.......... 59
5.6.3.1 LOC.......... 59
5.6.3.2 Size.......... 60
5.6.3.2 Performance.......... 60

6. CONCLUSIONS AND FUTURE WORKS 63

TERMINOLOGY 65

REFERENCES AND BIBLIOGRAPHY 66

APPENDIX A: PUBLICATION .... A1 – A9
APPENDIX B: CODE LISTING .... B1 – B7
APPENDIX C: INTRODUCTION TO MSIL .... C1 – C3
APPENDIX D: USING *Ildasm.exe* AND *Ilasm.exe* .... D1 – D5

LIST OF FIGURES

Fig. No.	Figure	Page No.
Figure 1.1	Easter Egg Watermark Example	5
Figure 1.2	Software Fingerprinting	8
Figure 2.1	Dollar Losses Due to Piracy by Region	19
Figure 2.2	Types of Software Piracy	21
Figure 3.1	Piracy Prevention Approaches	24
Figure 3.2	Anti-Piracy Lines of Defense	26
Figure 3.3	Thumbprint Encoding	29
Figure 3.4	Control Flow Graph of Thumbprint Code Block	30
Figure 3.5	IL Code Transformation	31
Figure 3.6	Thumbprint Decoding	33
Figure 4.1	Disassembling *HelloWorld.exe* through *Ildasm*	38
Figure 4.2	Re-Assembling Modified *HelloWorld* Program using *Ilasm*	39
Figure 4.3	Software Thumbprinting Tool Marka	40
Figure 4.4	Tower of Hanoi Application	42
Figure 4.5	Hashing and Encryption Setting in Marka	43
Figure 4.6	Thumbnail Generation using Marka	44
Figure 4.7	License File Creation using Marka	44
Figure 4.8	License Database Maintained by Marka	45
Figure 4.9	Control Flow Graph of Thumbprint 35_{10} (100011_2)	46
Figure 4.10	Decoding Thumbprint	47
Figure 5.1	Branch Based Watermarking	55
Figure 5.2	LOC	61
Figure 5.3	SIZE	61
Figure 5.4	Performance	62

LIST OF TABLES

No.	Table	Page
Table 5.1	Results Before Implementation	57
Table 5.2	Results with Proposed Scheme	57
Table 5.3	Results with Branch Function	58

Chapter 1

INTRODUCTION

1. Introduction

Software Security has two major divisions; a) data protection and b) software protection. Data protection refers to the security of information of an organization. Software protection, on the other side, refers to the security of software code which is ownership of a software vendor. We choose *software protection* as our major research area. In this chapter we start by presenting a broader view of the domain of software protection. After introducing the basics and backgrounds of software copy rights management, we will describe the problem of software piracy and emphasize the need of protecting Microsoft .NET framework-based applications specifically. Then we explore customary solutions against piracy and present an overview of proposed methodology. Chapter ends by describing our contributions and thesis organization.

1.1 Software Protection

Software programs comprise of valuable intellect, art and trade secrets. Software vendors aspire to control access to and usage of their software products on which they had spent their invaluable time and resources. These digital products are developed to make the life of their users easier. Among these software users, there are few crooks who aim to steal the source or semantics of software products in order to reproduce and redistribute them illegally. Software protection thus refers to the prevention of such unauthorized reproduction and redistribution. Software vendors possess copyrights of their digital work as a legal measure to enforce software copyright protection. International software copyright laws enable copyright owners to accuse some one who infringed digital products in order to use personally or to resale illegally. Besides this legal endorsement, to implant some technical protection measure inside their products, software vendors coined the term *Digital Rights Management (DRM)*. It refers to the techniques and technologies used by software vendors to enforce their copyrights over their digital products [1].

Technical implementation of DRM can in principle be reverse engineered, so it should be employed in strong conjunction with legal acts enforcement, so as to make software

copyright protection more effectual. In this endeavor we have proposed a technological implication of DRM that effectively combines software licensing and digital watermarking (as discussed in Chapter 3). Before presenting the proposed solution, we describe software piracy problem in coming lines.

1.2 Software Piracy

Wikipedia defines software piracy as [2],

> "*The unauthorized use of copyrighted material is a manner that violates one of the copyright owner's exclusive rights, such as the right to reproduce or perform the copyrighted works or to make derivative works that build upon it.*"

Collberg et. al. defines software piracy as an activity that some malicious user carries out by tampering with software code so as to replace or wreck its original ownership or authorship mark [10].

The first statement defines software piracy in general and second statement defines it in technical terms. We may simply define the term as,

> "*Software piracy is the act of making and reselling illegal copies of software product.*"

Many computer users intentionally or unintentionally become part of software piracy [4]. The very reason for this is misinterpretation of the word "copy" by software users. Users consider that copying fully functional software which someone else has bought legally is not piracy. Users don't feel illicit when they think that they are copying it for personal use not for resale. According to SIIA "it's more than a copy, it's a crime" [4]. The money loss caused by software piracy badly affects the profitability of software companies [3]. Hence, software owners and authors take serious legal actions to protect their products from piracy. Punishment of committing piracy varies from financial penalties to imprisonment. To avoid such unpleasant surprises, it is better to know the basic ways which intentionally or unintentionally may lead the user towards software piracy. SIIA and BSA describe the major types of piracy in. The greatest threat among different modes

of piracy is internet piracy. There are number of pirate websites which provide cracked versions of commercial software. Another widely used approach is CD-R piracy in which a person obtains a legal copy of software and makes its illegal copies for commercial resale [3, 4].

1.3 Piracy of .NET Applications

Software authors, mostly, do not consider copyright issues in early stage of development; rather it is passed over to the later stages of development cycle. Most of the times, it overlooked at all or if observed then weak protection mechanisms (like serial-key checks) are employed [35]. This is also true for .NET applications, where as the technology is getting its significant share in market. This lack of protection is derived from several reasons. Very first cause is user resistance to accept protection mechanisms and essentially .NET applications are more users intensive. Secondly .NET applications are more vulnerable to copyright infringements due to the ease of reverse engineering. An ordinary developer who is somehow familiar with .NET environment can simply use framework's built-in tools to expose business secrets, valuable proprietary algorithms and functional code from executable file of an application. There are more commercial tools on hand to disassemble .NET applications to get reverse engineered code and conversely protection tools are available in quite less proportion [19 , 20]. Theoretic approaches towards the solution of software piracy problem have demonstrated that a solution that comprises of merely software based measures like watermarking, code obfuscation or tamper-proofing is not feasible [6]. Besides this an efficient software distribution scheme is required to detect and react against encroachment [7, 8]. Further we take a glimpse of customary solutions which software vendors employ to protect their products against copyright violations.

1.4 Proactive Solution

The legal measure to protect software copyrights like, patent, copyright, and license does not provide complete control against piracy. There arises the need to enforce legal measures through technical mechanisms. For example, the serial-key check that most

commercial software implements to validate the legal product user. This is a software-based anti-piracy technique, although thwarted but still being widely used. So there is an imperative need of developing technical protection measures besides legal means.

Researchers have been devising technological countermeasures, both hardware-based and software-based to resist against piracy [1, 10, 14, 17, 21]. For our proposed solution, software-based protections are of major concern which we will shortly present after taking a quick look of hardware-based protection measures.

1.4.1 Hardware-based Protection

Hardware-based protections provide a variety of features. These are powerful, fast and autonomous. Such solutions barricade software piracy by affixing application authentication mechanism with some fixed hardware device which contains embedded ownership signature [1]. These systems make functional program dependent upon certain hardware components. The program runs successfully if it finds specific device attached with the system and also if the device contains particular authentication mark embedded on it. Naumovich et. al. have explored several hardware-based protection tools including dongles, smart cards, re-writeable media installers, dedicated processing devices and Trusted Computing Platform Alliance (TCPA) [7, 8]. Some of these also support licensing schemes. These tools are certainly more secure but their fabricating cost and realization overheads confine their viability to large application domains [7]. Software-based protections discussed below, are on the other hand extensively practicable due to their less implementation overheads and low installation cost.

1.4.2 Software-based Protection

This type of protection does not involve any special hardware device to be used to restrict use of software product. Instead it refers to certain property of software program that indicates the original owner and legal buyer. We will here explore particularly those measures which are the focal point of our research and contribute in the proposed protection scheme, like watermarking, tamper-proofing and code obfuscation.

1.4.2.1 Software Watermarking

Software watermarking is the process of embedding secret information into the software. This information may identify the ownership of software. When an unauthorized use of this software takes place, its copyright owner can prove her ownership by extracting this secret information from this unauthorized copy. Software watermarks can be classified by their extracting techniques as static or dynamic. A *static software watermark* is one, the extraction of which does not require the program to be run. Instead the code statements are analyzed to find some static string or numeric constant that can be some secret identification or vendor name or license information. A *dynamic software watermark* is embedded in the execution state of a software program. More precisely, in dynamic software watermarking, there are two ways to insert watermarks. Firstly, what has been embedded can be the watermark itself and secondly, some unique input sequence which causes the watermark to be extracted can be the watermark [7, 9, 10, 12].

Dynamic watermarks are embedded in several ways and a simpler one of them is *easter egg watermarks* in which watermark can only be extracted by a highly unusual input. An example of easter egg watermark is Adobe Acrobat Reader 4.0 [11]. In Acrobat Reader 4.0 select Help -> About Plug-ins -> Acrobat Forms, the screen shown in Figure 1.1a will be displayed. Then hold Control+Alt+Shift while clicking on the credits button. This special input sequence will expose the watermark, as shown in Figure 1.1a.

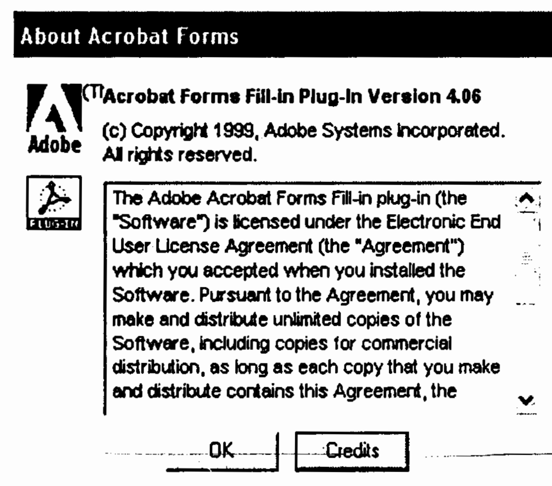


Figure 1.1a Easter Egg Watermark Example [11]

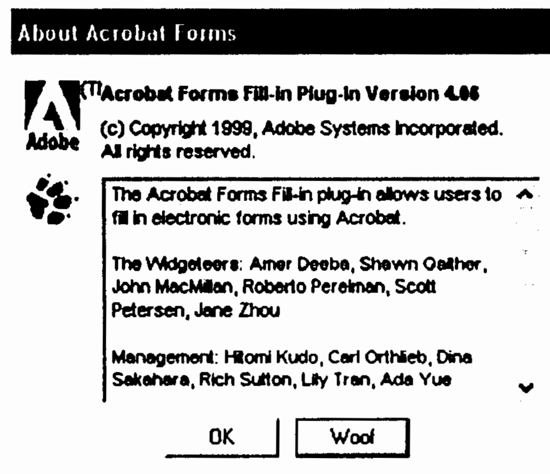


Figure 1.1b Easter Egg Watermark in Adobe Acrobat Reader 4.0 [11]

This type of watermark is considered to be relatively weaker protection. There are several other dynamic protection mechanisms which offer higher level of security. Clark Thomberson has presented a comprehensive survey of watermarking techniques in [9].

Properties of an Efficient Software Watermarking Scheme

There are certain characteristics of an efficient watermarking scheme which are observed while devising the software protection technique. Presented below is the list of most significant properties which an efficient software protection scheme must exhibit. These characteristics also help in evaluating the strength of a watermarking technique.

Tamper-proofing: The first and foremost requirement after a watermark in inserted in a program, is to safeguard the watermark or the code that renders the watermark against code tampering. Tamper-proofing aspect of a protection scheme causes the watermarked program to malfunction if its code is tampered in order to distort the watermark [13, 17].

Robustness: Refers to the characteristics of a watermarking scheme which means that the watermark value embedded in a program must be retained after some adversary applies semantic-preserving transformations on the watermarked program. Robustness is one of the most significant properties that a watermarking scheme must hold in order to prove the ownership of a program even after it is tampered [10, 14, 17].

Fingerprinting: The third important characteristic of an efficient protection scheme is its ability to trace back to the source of infringement if it is happened. For this purpose distinct watermarks must be embedded in each distributable copy of a program [7, 10].

Resilience: The watermarked program must be resilient in terms of size, space and performance. It means that if a watermark is encoded in a program then the size (LOC), space (memory consumption) and performance (computation time) of watermarked program must not be verily differing from the original program. Drastic changes in these measures weaken the efficiency of a protection scheme [9, 10].

Data Rate: It refers to the number of times a watermark is inserted in a program. It is usually on the contrary to the *resilience* trait of a watermark because increasing the rate of watermark insertion also increases the size, space and computation time of the program that decreases the *resilience*. So the goal of an effective watermarking scheme is to increase the data rate while keeping the resilience high [7, 10].

Stealth: Watermarked program must be stealthy so that its observable behavior must be similar to the original or its pear programs [7, 9, 10].

Cost: The cost of a protection scheme must be less in terms of time and resources required to implement it. On the other hand this cost must be high enough for an adversary who intends to break it [10].

1.4.2.2 Software Fingerprinting

Software fingerprinting is a technique that specializes software watermarking. Fingerprinting refers to embedding unique watermarks in each distributable copy of intended software [7]. Digital fingerprints can be used to identify individual copies of pirated software programs. Technically watermarking and fingerprinting are similar to each other because both aim to embed a secret message in a software program. While in general a watermark identifies only the software authors and a fingerprint can also identify the source infringer which tinkers with the fingerprinted program. To do so, fingerprinting embeds a unique customer identification number into each distributable

copy of an application to trace back the copyright violators [7]. Figure 1.2 depicts the basic idea of fingerprinting. Alice owns a program *P* and she intends to sale it to Bob, Charles and Douglas. She embeds unique fingerprints (F_1, F_2,, ..., F_n) into distinct copies of program *P* to produce its fingerprinted copies (P_1, P_2,, ..., P_n). Each copy holds a unique fingerprint that points towards its original buyer hence tracing back the source of piracy if happens.

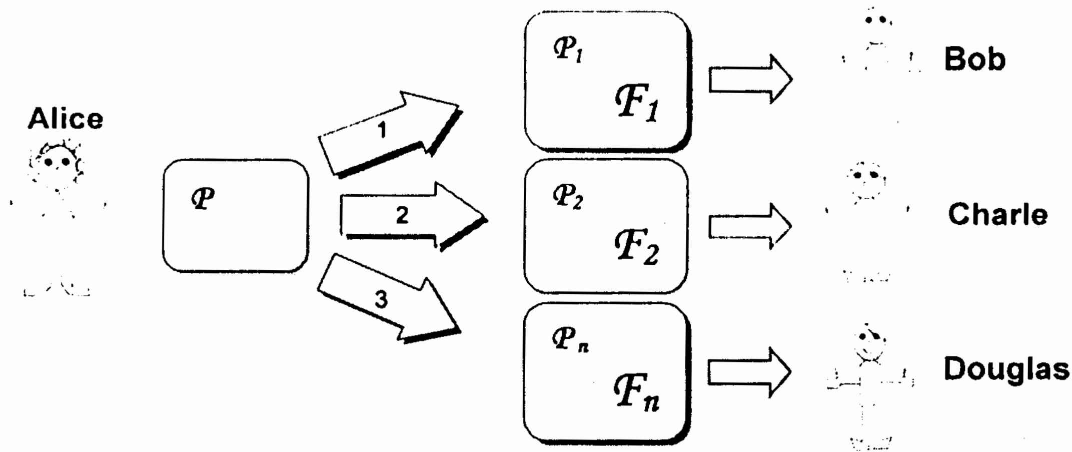


Figure 1.2 Software Fingerprinting [10]

Proposed *software thumbprinting* scheme realizes the idea of software fingerprinting. We introduced the notion of *thumbprinting* that refers to the characteristics of intended solution as detailed in Chapter 3. We propose to embed unique *thumbprints* into distinct copies of software product.

1.4.2.3 Tamper-proofing

Tamper resistance is a technique used to safeguard the embedded watermark or fingerprint. It aspires to cause the application crash or malfunction if tampered by some adversary [7, 10, 35]. Tamper resistance resists unauthorized modification of protected program code. In order to make a protection scheme more effective, tamper-proofing is combined with other techniques like watermarking and fingerprinting, to confound an attacker. A common approach is to calculate the hash value of the whole block of software code and save it in the program. When program is run, this hash value is

compared with the hash value of currently running copy of the software to check the integrity of software. If these two hash values do not match, the software is assumed to be tampered and program terminates or starts malfunctioning [13, 14].

Another approach to implement tamper-proofing can be to create strong dependency among original program code structures and the protected code structures so that if some malicious user tries to tamper with the protected code area, the program starts malfunctioning [23, 31]. In proposed thumbprinting scheme we craft such dependencies among the embedded thumbprint and program source code by transforming executable instructions.

1.4.2.4 Code Obfuscation

Most of the times software vendors require to prevent malicious users from understanding program code through reverse engineering. Code obfuscation refers to the practice of transforming program code into functionally equivalent code which is difficult to be understood by an adversary. This way code obfuscation restricts an attacker from stealing the source or semantics of critical code blocks. Code obfuscation is also used to disguise other software-based protection mechanisms like watermarking, fingerprinting and tamper-proofing. It is done by obfuscating the whole program code or specific blocks of code which are containing some secret information or behavior. It has been a consensus of researchers that given enough time and resources, a determined attacker can reverse engineer even machine level code [7]. So the practical goal of obfuscation is to make reverse engineering too expensive, so that the cost of reverse engineering must be no less than the cost of developing the software from scratch. Moreover obfuscation aims to alter the code so that to make it difficult for disassembler to decompile it to higher level language constructs which are easier to understand.

Likewise, intended protection scheme proposes similar code transformations which thwart .NET decompilers to produce high-level language code from low-level IL code instructions.

1.5 Overview of Proposed Methodology

We aim to prevent software piracy by embedding thumbprints into every copy of software distribution. *Software Thumbprinting* works in tandem with licensed distribution to detect and react to an infringement. The scheme is specific to protecting .NET based applications because it has not been done before, as opposed to Java based applications where several watermarking algorithms are available. The idea is to deter an attacker who buys a legitimate copy with a digital watermark, and redistributes it after removing the watermark. We identify the problem as devising tamper-proofed watermark.

Proposed solution is to obtain unique thumbprints from a thumbnail. The thumbnail itself is a short fixed version of the encrypted license. The unique thumbprints are then encoded into the disassembled instance of the program. The instance of the program is then reassembled and distributed. This license information along with the thumbprint is also recorded in the database in order to prove ownership. The idea is that every instance will have a unique embedded identifier. Chapter 3 discusses the proposed methodology in more detail.

1.6 Our Contributions

Much has been done to thwart copyright infringements but not a single method discussed so far is found to be perfect for software protection. Every method tries to make the copyright violation harder but not impossible. In this thesis, we initiate the use of software licensing with other technical protection measures like watermarking, tamper resistance and obfuscation. The proposed mechanism is called *software thumbprinting.* We propose to encode unique thumbprints into each distributable copy of software and then distribute these thumbprinted copies along with their usage licenses to legal product users. We contribute towards raising the level of software copyright protection for .NET applications. The feasibility of intended approach is realized by its real implementation (in Chapter 4). Following are our major contributions while devising the proposed software protection scheme:

- Scheme specifically protects .NET framework-based applications against copyright infringement, code tampering and reverse engineering. Most of the protection schemes so far devised discuss Java and Assembly language programs' protection. Where as .NET application protection specifically based on .NET framework constructs is rarely discussed. In this dissertation we have explored MSIL (the fundamental construct of .NET framework) to devise an anti-infringement system for .NET applications.
- Introduced the concept of embedding unique thumbprints as ownership signatures to protect software copyrights. The word thumbprint refers to, 1) a trace mark of the buyer, 2) a watermark that exhibit the idea of fingerprint and 3) a thumbnail (short length) version of complete license information which is produced at the time of a copy-sale instance.
- Proposed scheme works in tandem with licensed distribution and supports piracy prevention, detection and response mechanism. On every copy-sale instance, a legal buyer is given an exclusively compiled program copy and license file. The program and the license file are bind to the same thumbprint value and the license information is recorded in a license server which is afterwards queried while decoding the thumbprint to assert program ownership.
- Protection scheme caters two of major characteristics of an efficient software watermarking scheme, 1) Robustness and 2) Tamper-proofing. This way a robust watermark is encoded in a .NET program at method level. The watermarked methods are tamper-proofed through strong dependency crafted among watermark code and original code.
- Provided complete architectural and formal design of proposed methodology and further explained implementation details of devised watermarking algorithm through a case study. As observed most of previous protection schemes are not practicable due to lack of their implementation details. In this thesis, proposed thumbprinting technique is presented with complete architectural, formal and technical details. Moreover experimental results are calculated and evaluated against another recently proposed protection scheme

1.7 Thesis Organization

The thesis is organized as follows;

Chapter 2 presents related literature survey that highlights the significance of software piracy problem. Then we define research problem under investigation in the same chapter that points up research objectives and confines the scope of this work. Then proposed thumbprinting scheme is discussed in Chapter 3. Firstly we explain where proposed solution lies among other software-based protection mechanisms and then we present architectural and formal model of intended solution. Chapter 4 demonstrates proposed protection scheme through a case study that explains its implementation details. Chapter 5 presents results and evaluation of proposed scheme. Finally, future directions and conclusion is included in Chapter 6.

In Appendix A, research publication is annexed. Appendix B contains code listing. In Appendix C, an introduction to MSIL is provided. Lastly, Appendix C contains a technical overview about the usage of IL code assembler and disassembler.

Chapter 2

LITERATURE SURVEY AND PROBLEM DEFINITION

2. Literature Survey and Problem Definition

As soon as use of computers increased, unauthorized copying of software became a critical problem. The advancement in code analysis tools and the popularity of internet created more opportunities to steal software. The money lost due to software piracy is included in the cost of legal software and therefore pirated copies are also paid by the legal users. Today's complex software is of much value to its inventor. This software vendor can be a big company with many products or it can be a small company with a single product. Of most concern is the protection of the software, such that it will always retain the functionality which its creator intended. Software must always protect the intellectual property embedded in the program, and must prevent malicious attempts to make its illegal copies by overwriting its ownership signature.

This chapter presents related literature review which shows that major emphasis of research in software protection is on developing such algorithms and techniques which prevent malicious software users from distorting owner's copyright signatures. Further it highlights the significance of problem by presenting statistics of monitory loss being faced by software vendors every year due to piracy. These facts and figures rationalize the need of solving the problem of piracy from both, vendor and buyer's point of view. Then we outline the problem that helps identifying basic terms in the milieu. After this we state concrete problem definition and describe our research objectives which our intended solution will cater. Chapter ends by defining the scope of work.

2.1 Literature Review

Software protection is a continuous battle between two forces, first one is the defender who wants to secure her intellectual property and the second one is attacker who always looks for loopholes to break into product protection mechanisms. One of the current limitations with legal deterrents is that, the pirates are aware of the practical difficulties of their enforcement. However, an adversary may be less likely to redistribute a piece of software if he knows that it can be traced back to him. Software watermarking is a major protection measure employed by product vendors to fight against software piracy. The

core objective of watermarking is to prove ownership of intellectual property, while discouraging its illegal use. Software watermarking provides a mean to prove ownership of pirated software and in the case of fingerprinting it even identifies the source of the illegal distribution by embedding unique identifier in it [7, 10, 22, 23, 25].

Watermarking refers to the insertion of static or dynamic signatures into a program, which serve to identify the original owner. Static watermarks never change, and are therefore subject to some level of reverse engineering. Dynamic watermarks change with the program execution. Watermarks are either extracted from an image embedded inside a program, or from program code, from program data, or from program execution itself. A comprehensive overview of software watermarking techniques is presented in [22, 23, 24, 25].

It is also important to keep in mind, that in order to increase the effectiveness of protection mechanism, multiple approaches should be combined. One should think carefully about how to mingle different approaches, and strive to mask the weaknesses of one, with the strengths of another. For example, combining watermarking with tamper-resistance increases its effectiveness. There exists a wide range of tamper resistance methodologies that can be used to safeguard a watermark [31, 34, 35]. Another approach being widely used is code obfuscation [23]. It aims to obscure program code to make it hard for an adversary who tries to steal critical functional code. Like in, combining control flow monitoring with obfuscation can lead to a monitored program that requires significant efforts to reverse engineer [26].

In this research we have presented a blend of software watermarking and temper proofing. The previous approaches have mostly disused watermarking java programs. Many researches contributed towards the definition of software protection as a major problem. Gleb et. al. in '03 describes each term associated with software protection, very precisely and clearly [7]. They discussed that software protection is not limited to watermarking but it includes securing software through licensing files, application servers, hardware based-solutions, temper proofing and obfuscations. We need complete software protection framework in order to fully protect software product. Collberg in '02

discussed watermarking, tamper-proofing and obfuscation as tools for software protection [10]. These tools are effective against three types of attacks of a malicious host, including software piracy, reverse engineering and tamper-proofing.

We need to apply a combination of these protection tools in order to completely secure the software. Thomborson et. al. in '05 especially focuses on software watermarking as a protection measure against software piracy. They described four major attacks on software watermarks, including additive, subtractive, distortive and collusion attacks. They have also discussed different algorithms proposed by other people for software watermarking. They proposed a unique watermarking technique that refers to embed watermarks in the addresses of caller and calling function [9]. These addresses were dynamically manipulated according to branch-function based watermarking introduced by Collberg in '05 [27]. This technique adds a dummy branch function which dynamically generates return address. Sander et. al. in '98 discussed a technique that reduces the risk of reverse engineering [28]. Their idea is to compile a distributable instance of a program in such a way that it only contains calls to functional code. A server is placed alongside, that hosts all the functionality of program. When ever user runs this program, a functional call to server is made with required parameters and server processes these parameter values and finally returns back the results. This way functional code is never distributed to the user hence reduces the chance of reverse engineering and tampering at all. The major downside of this approach is that it is dependent on heavy client-server communication which hinders its applicability to wide range of software.

Min Chen et. al. in '01 presented a detailed study of legal & technical measures to protect software products [8]. In legal means they discuss the copy rights, patents and licenses and in technical means they covers software distribution model. Wiener et. al. in '05 explored two of commercial obfuscation tools for .NET based applications [19]. These tools are 1) Lutz Roeder .NET Reflector and 2) Remotesoft .NET Explorer. We have employed such tools for evaluation purpose of proposed methodology. Software piracy problem can also be solved through diversity. As described, diversity is a mechanism of distributing unique installation to users and providing then customized updates. In this scenario, customer has to be connected to the vendor in order to get updated functionality

of software. This scheme exhibits certain drawbacks, like, a) user can use and distribute illegal version of software product as long as he don't require an update and b) it requires huge cost overhead in order to create customized updates for each customer [29].

We have seen how different researchers have tried to solve software piracy problem for Java and Assembly language. Mishra et. al. in '05 proposed a method of static watermarking for whole program [30]. Watermark is inserted in at least 2N+1 methods of a program if N methods are required to be protected. A control flow graph of a method is constructed and values are assigned to its code constructs, including if-else, loop and etc. Watermark is considered as a numeric value and a code block is constructed whose statements exhibits equivalent value to that of the watermark. This watermark code block is merged within the code of a method in order to protect it. The basic problem with this scheme is that it lacks the implementation details of how to create a watermark code block and how to embed it in original code while keeping it obscure. Curran et. al. in '03 has proposed a new technique called method-depth technique. Proposed scheme embeds a value of watermark on the basis of call graph depth of a method. Scheme is mainly based on recursion that increases performance overhead [16]. Another approach is presented by Collberg et. al. in '04 that embeds the watermark in the dynamic branch structure of a program. They have used temper-proofing techniques to prove that their watermark is resilient [17].

Various techniques of temper-proofing and obfuscation have been used because if watermark is not temper-proofed properly it can easily be detected and removed. Thomberson et. al. in '04 presented a technique based on constant encoding. They replaced the numeric constants appearing in a program with function calls. The function computes back this numeric value through a dynamic data structure, called (PPCT) [14]. Though this technique seems robust and tamper-proofed but at the same time the presence and creation of a dynamic data structure makes it un-stealthy and performance overhead. Collberg et. al. in '98 presents several obfuscation techniques for Java based applications [31]. They presented opaque constructs and introduced them in the program where there is a chance of condition evolution. They have also used basic constructs like if-else with some abstract data types (ADT) like tree, graph or threads to introduce

opaque predicates. Another obfuscation scheme presented by Fukushima in '02 was based on destructing encapsulation structure of classes by transferring one class's local methods, variables & instructions to another class as static members [32]. This creates high dependency among classes, thus complicate reverse engineering by causing temper-resistance.

Code obfuscation make reverse engineering a difficult tasks in terms of time and effort [32]. Obfuscation is normally done by transforming original code into equivalent code that will hard to understand using some static analysis methodology [31, 33, 34]. Obfuscation can be employed with proposed methodology to strengthen its level of protection. Obfuscation techniques involve lexical, control and data transformations [35]. Proposed thumbprinting scheme employs control flow transformations, as discussed in Chapter 3.

There are automated obfuscation tools available in the market for both Java and .NET based applications. Two of these are tools LOCO (for Java applications) and Dotfuscator (for .NET applications). Dotfuscator removes debug information and non-essential metadata from a MSIL file as it process it [19]. This tools work on complied MSIL code, not source code. The main thing is that obfuscated MSIL is functionally equivalent to traditional MSIL code and will execute on Common Language Runtime (CLR) with identical results. This tool supports both incremental and control flow obfuscation. Another relatively new tool LOCO employs control flow flattering branch function & opaque predicates as basic transformation on java code [21]. We have also explored some commercial tools available to protect .NET applications, like Crypkey [41]. CrypKey is an easy-to-use, cost-effective, digital rights management solution for protecting intellectual property from unauthorized use. Virtually any data file that can be opened in a Windows environment, including proprietary software, can be protected by CrypKey. CrypKey works by protecting and binding files to a specified computer or CrypKey USBKey, thereby thwarting illegal copying. Even the authorized access to any CrypKey protected software can be issued for any restricted time period. Remotesoft's Salamander [19, 42] is another .NET decompiler that explores executable files (.EXE or .DLL) to Intermediate Language (IL, MSIL, CIL). Salamander has been rigorously tested and is

being successfully used to produces equivalent and recompilable MSIL code from .NET executables. Besides .NET explorer, Salamander also contains a .NET obfuscation tool to protect .NET applications. Another widely used decompiler for .NET is Lutz Roeder's .NET Reflector [19, 43]. Reflector is developed in .NET and is more user friendly tool than Salamander. Reflector not only decompiles .NET executables to MSIL but also generates equivalent high-level code, such as C#, managed C++, Visual Basic.NET, J#, etc. Eziriz has provided a protection tool for .NET called, NET Reactor [43]. Reactor thwarts reverse-engineering of .NET applications by encrypting the executables. We tested Reactor over several .NET assemblies and found that executables protected with Reactor are even not decompilable by Salamander and Reflector. Reactor successfully stops these decompilers to produce MSIL code or high-level .NET code.

Besides above mentioned 3rd party tools for .NET framework, Microsoft has provided and obfuscation tools, Dotfuscator [19, 20]. A free Community Edition of Dotfuscator is shipped with .NET Visual Studio that renders basic code obfuscation algorithms. Dotfuscator's Professional Edition is also available at [20] which expose most of obfuscation algorithms including string encryption and control flow obfuscation. Dotfuscator also contains a utility to watermark .NET applications. Dotfuscator is one of the most widely used tools for the protection of.NET applications.

2.2 Problem Rationale

As per our findings through above literature review, most of technological considerations and implementations of software protection schemes are found for Assembly language's binary code and Java language's byte code [13, 15, 16, 17]. Where as, Microsoft Intermediate Language (MSIL or shortly IL) is rarely discussed. Researchers are always on the move to devise more secure and more robust technical protection measures against software piracy. This study will particularly consider MSIL constructs for their support to devise generic solutions for the protection of .NET framework-based applications against illegitimate adaptation, distribution and reproduction. Moreover it emphasizes that there is momentous need of developing research-oriented as well as commercial tools to protect .NET applications.

Alongside the researchers' call of attention towards solution of software piracy problem, there are statistical facts from software industry which emphasize the need to resolve this grave trouble. According to an annual global study on software piracy conducted by Business Software Alliance; 35% of installed software packages in year 2005 were pirated. That cost a penalty of $34 billions to software vendors in one year, which is forecasted to grow up to $200 billion during five years [5]. These results confirmed that piracy will continue to be a significant problem for coming years and will keep on raising the revenue loss bar. Figure 2.1 below, shows the geographical statistics of money losses due to software piracy.

Vendors and authors suggest that software piracy negatively affects the global economy because it decreases the profit which allows growth and development within the software industry. Laws regarding copyright infringement state piracy as a crime that may cause penalty of up to $2, 50,000, to the prosecute [3].

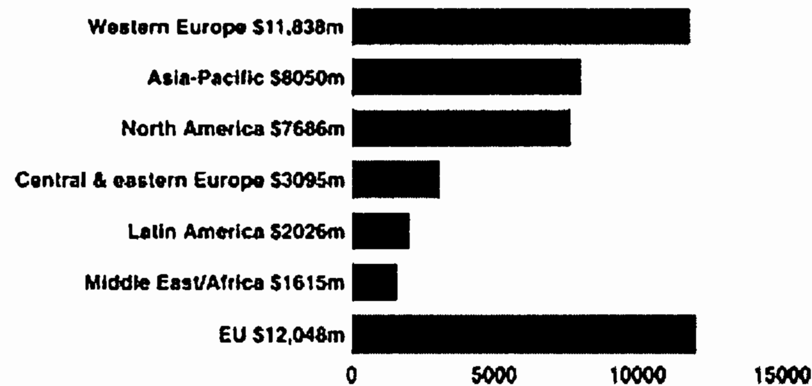


Figure 2.1 Dollar Losses from Piracy by Region [5]

2.3 Software Piracy Problem Redefined

We redefine software piracy problem in its real context to model the framework for anticipated solution. It will identify different objects of problem domain.

- Alice is the owner of software program P. She intends to earn profit from P on its copy-sale basis. So she has acquired legal copy rights of P for its reproduction and redistribution

- Alice has embedded her ownership signature into P that proves P as an intellectual property of Alice
- Bob purchases a licensed copy of P from Alice
- Bob determines to pirate P to earn unlawful gains from its resale. So he applies certain attacks on P to replace Alice's ownership signature with one of his own
- Alice necessitates some prevention measure to be employed into P which will resist against Bob's malicious attacks and will safeguard Alice's ownership mark
- Having enough time and resources, Bob manages to override Alice's signature
- Bob sales pirated copies of P to Charles
- Charles shares this illegitimate copy of P with his colleague Douglas
- Alice hence requires detecting pirated copies of her software
- Moreover, Alice needs to detect the actual pirate Bob so as to prosecute him
- Once detected, Alice has to render her ownership signature from P, in the court of law so that to assert her ownership

Above problem is based upon the illustration of Collberg et. al. [10]. They have discussed basic modes of copyright infringement which Bob can opt in order to pirate P. Figure 2.2 depicts next level of formalization of context of software protection. It illustrates technical distinctions among several types of software copyright violations. Figure 2.2 (a) depicts an illegal copy-sale instance, as Bob makes illicit copies of Alice's program *P* and resells them to Charles and Douglas. Figure 2.2 (b) describes malicious reverse engineering attack in which Bob buys one legal copy of *P* from Alice. This original program is having several subparts like *M, N* and *O*, which perform useful functionality in *P*. Bob decompiles *P* and extracts one of its functional modules (e.g. *O*) and reuses it in his own program *Q*, without Alice's permission. Then he sells his program *Q* to different users while claiming that this *Q* is entirely his own product.

Figure 2.2 (c) points up software tampering attack, in which Bob purchases a digital container *C* from Alice. *C* contains a song file embedded in it. Alice has programmed *C* in such a way that any of its users have to pay $0.05, each time he plays the song. Bob disassembles *C* and performs code tampering attack on it in two ways.

1) He modifies C by changing the amount $0.05 to $0.01, so that he has to pay less for playing the media file
2) He extracts the song file from C for illegal resale or reuse

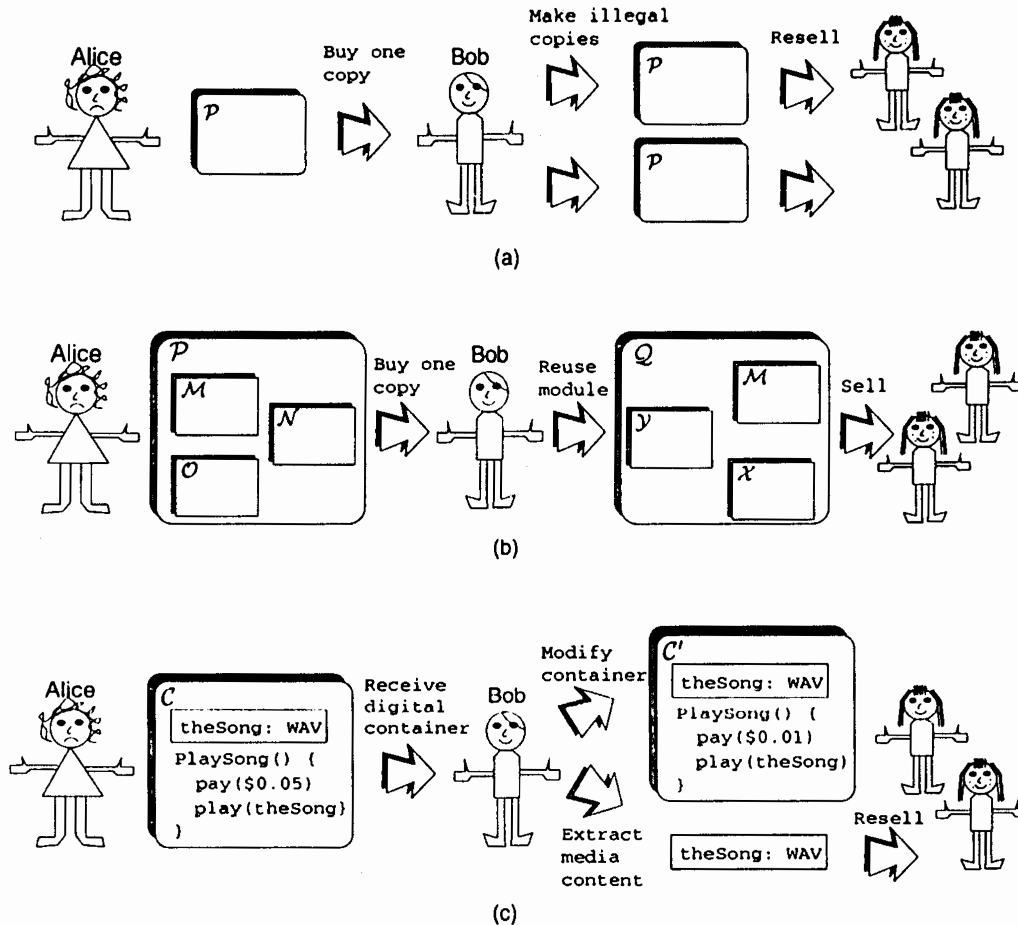


Figure 2.2 Types of Software Piracy [10]

Bob may choose any of the above approaches to pirate Alice's program. He can attack Alice's ownership by distorting or overriding her original copyright signature. Alongside, he has to keep the semantics of P intact, so that to get benefit of full functionality of program P even after tampering with it. Collberg et. al. classifies these types of software piracy and maps them to their relative protection measures like software watermarking, obfuscation and tamper-proofing respectively [10]. Researchers tend to agree on the fact that it is almost unrealistic to prevent Bob from distorting watermark and also from using P, at the same time, if he is given enough time and resources for a determined manual

attack [5]. Likewise, it is unacceptable to allow him to use full functionality of *P* after successfully destroying the original ownership mark.

Before stating the anticipated problem we mention the following points as basic concerns:

a) Bob must loose the functionality of *P* if he distorts Alice's ownership mark, otherwise
b) Bob must use the functionality of *P* only if he could not distort Alice's ownership mark

These concerns help us materializing problem statement in coming section.

2.4 Problem Statement

" How to devise a *robust* and *tamper-proofed* software protection scheme for .NET framework-based applications? "

Above problem statement highlights two major characteristics of a good watermark. Firstly, *robustness*, that refers to retain the watermark inside the program even after any attacker applies transformation attacks on it. Secondly, *tamper-proofing*, that aims to safeguard the watermark and making the program malfunction if tampered by the attacker. These two characteristics are found most imperative from the literature survey sighted in previous section, in order to device an efficient software watermarking technique.

Objectives

In order to solve the question points of above stated problem, research goals are set as follows;

- ***Robustness***

  Allow Bob to use program *P* but keep Alice's ownership mark intact

- ***Tamper Resistance***

  If Bob successfully distorts Alice's ownership mark then make program *P* malfunction

2.5 Scope of Work

- The protection scheme will primarily focus on .NET framework-based applications
- Scheme will impose software fingerprinting
- Proposed scheme will work in tandem with software licensed distribution
- Protection scheme will be demonstrated with sufficient implementation details
- Strength of proposed scheme will be evaluated against other protection schemes

The anticipated solution is primarily presented for .NET based application but it will be evenly practicable for other high level and low level languages like Java, which have similar code constructs to that of .NET.

Chapter 3

PROPOSED METHODOLOGY

3. Proposed Methodology

This chapter presents detail of proposed software protection scheme in accordance with the problem definition stated in preceding chapter. We start by exploring generic piracy prevention measures. Then anti-piracy defense suit is described which comprises of existing technical solutions to protect .NET applications. Finally proposed *Software Thumbprinting* scheme, its architecture and formal model is presented.

3.1 Piracy Prevention Approaches

The customary solution against software piracy lies among certain *ethical, technical and legal* measures. Piracy prevention scheme is divided into four major divisions; i.e. *piracy education, piracy prevention, piracy detection* and *piracy assertion*. These subdivisions at the same time help sorting out their respective anti-piracy measures. Figure 3.1 illustrates the idea that how Alice tries to protect her program P from its intended user community by employing these defenses.

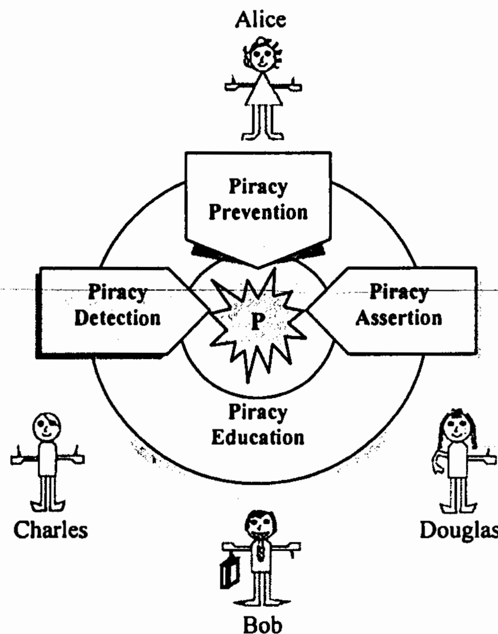


Figure 3.1 Piracy Prevention Approaches

3.1.1 Piracy Education

Software vendors at first place, ethically edify users about benefits of intellectual property protection and its contribution towards worldwide economic growth. Organizations like Business Software Alliance (BSA) are in force in this arena with partnership of leading software companies like Microsoft, Apple Mac., RSA, McAfee, etc. to promote safe and legal digital world through education and enforcement of digital copy rights [5]. These organizations educate people about the importance of using legal software and consequences of using illegitimate software.

3.1.2 Piracy Prevention

It refers to the technical defensive measures which software providers implant into their products so as to prevent piracy at first place. It is sure that copying digital artifacts can not be restricted. These preventive measures cause illegitimate copies to malfunction or stop running at all or cause enough cost overhead to pirate so that it may exceed actual development cost of software [7].

3.1.3 Piracy Detection

If in any way a pirate successfully wrecks preventive measures and start profiting from it through illegal redistribution or personal usage, than copy right owner needs to detect such copy right violations in order to take some legal actions against such malicious user.

3.1.4 Piracy Assertion

Finally when some software vendor fortunately detects any illicit copy of his software, he/she is required to prove his/her ownership in the court room.

Technical and legal measures are in strong conjunction with each other. One of the major reasons why piracy prevention has not yet been achieved successfully is that most of the time vendors (if do) typically employ mere technical protections without decisively taking up reactive legal actions against their copyright's infringements. Therefore *anti-*

piracy defense suit, discussed in succeeding section, embraces all technical piracy deterrence mechanisms like prevention, detection and assertion.

3.2 Anti-Piracy Lines of Defense

In order to realize software protection measures discussed above, we present a technological solution that comprises of existing lines of defense against software piracy. Proposed anti-piracy defense suit is already in practice for .NET framework-based applications [12]. We intend to align it in more explicable and rational order, as Figure 3.2 illustrates it. This alignment suggests how any of protection schemes can be strengthened by combining it with other ones.

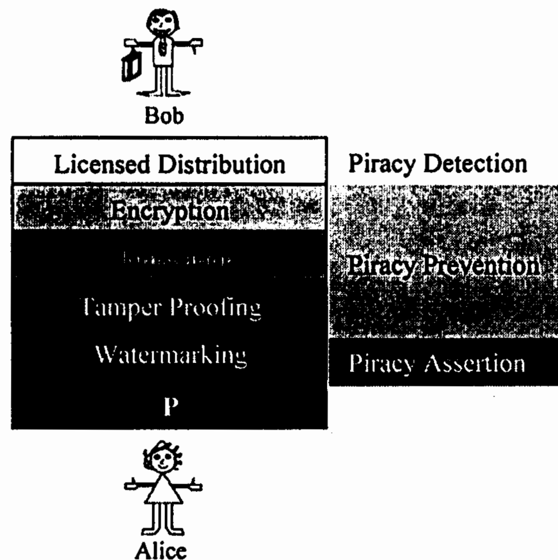


Figure 3.2 Anti-Piracy Lines of Defense

In order to employ this protection suit, Alice starts by developing the inner most layer, the program *P* and keeps on wrapping protection layers around it till its *distribution*. First she embeds her secret ownership signature into P as a *watermark*, which she will afterwards render in the court of law to assert her copy rights. Then she *tamper-proofs* her watermark so that P must become useless for Bob if he determines to distort Alice's ownership mark. After that she *obfuscates* that tamper-proofed program to make P's source code obscure enough against malicious reverse engineering. She subsequently *encrypts* executable of P to make its code indecipherable. Finally she distributes *licensed*

copies of her software to respective *licensed* users. Alice may choose to employ any or all of these lines of defense according to the required level of protection.

On the other hand, Bob needs to defeat all of the drawn protection layers, starting from the outer most, the *licensing* mechanism. He keeps on infringing by *decrypting* the executable and *de-obfuscating* program P's source code. After that he *tampers* with the code in order to get a fully functional instance after successfully distorting the *watermark.*

Proposed thumbprinting technique, conforms this defense suit. It employs software watermarking, tamper-proofing and licensed distribution and consequently realizes piracy assertion, prevention and detection mechanism. Coming section illustrates the idea in more detail.

3.3 Software Thumbprinting

Software Thumbprinting refers to:

"*The process of encoding unique thumbprints into each distributable instance of a program.*"

A thumbprint serves as covert signature of software vendor which he may use to prove his ownership over the program. This thumbprint is a numeric value formulated from licensee's information produced at each legitimate copy-sale occurrence of source program. This way, every time vendor sales a product to a buyer, he creates a new copy of original program and gives it to the legal buyer after encoding unique thumbprint into it. This unique thumbprint identifies one and only copy-sale instance, hence traces back to the original buyer of that particular copy. The notion of *software thumbprinting* implies this idea as in this technique buyer's information is embedded into the program as his *thumbprint.* This thumbprint identifies that who is the legal buyer of a particular copy of software. Furthermore, each legal buyer will be having a custom-made copy of a product, along with its custom-built license file. This license comprises of licensee's information in enciphered form. Every product user is required to bring un-tampered

version of thumbprinted program and respective license file, in order to prove that he is a legitimate buyer of it. Otherwise Alice can accuse him for using her product illegally without buying its legal usage license. Following section presents detail design of the proposed thumbprinting scheme.

3.3.1 Design of Proposed Methodology

Software Thumbprinting has following parts in design;

- Thumbprint Encoding
- IL Code Transformations
- Thumbprint Decoding

3.3.1.1 Thumbprint Encoding

Once a .NET program is developed and its source code is compiled into a software product, then Alice needs to generate unique thumbprints which are to be embedded into each of its distributable instances. These *thumbprints* are distinct numbers produced from thumbnails of license information. A *thumbnail* is composed of complete information produced from a particular copy-sale instance, which may include but not restricted to, vendor's information, buyer's information and product's information. The process of thumbprint encoding goes through following steps (illustrated in Figure 3.3).

1. Generate a *thumbnail* of vendor's, buyer's and product's information.
2. Create a *license* file and save encrypted thumbnail into it
3. Generate a unique *thumbprint* by taking hash of encrypted contents of license file
4. Encode unique thumbprint into the original program. Encoding process goes through following steps:
   4a. Disassemble .NET program to its IL level code
   4b. Embed thumbprint into IL code through proposed code transformations (discussed in subsequent section)
   4c. Re-assemble modified IL code to a thumbprinted version of original program

This thumbprinted instance of original program is given to Bob along with its legitimate usage license. Finally, Alice preserves all the information produced in this copy-sale instance into the database. This database can be hosted as a license server. Querying the database with valid product information will yield its licensee's complete information.

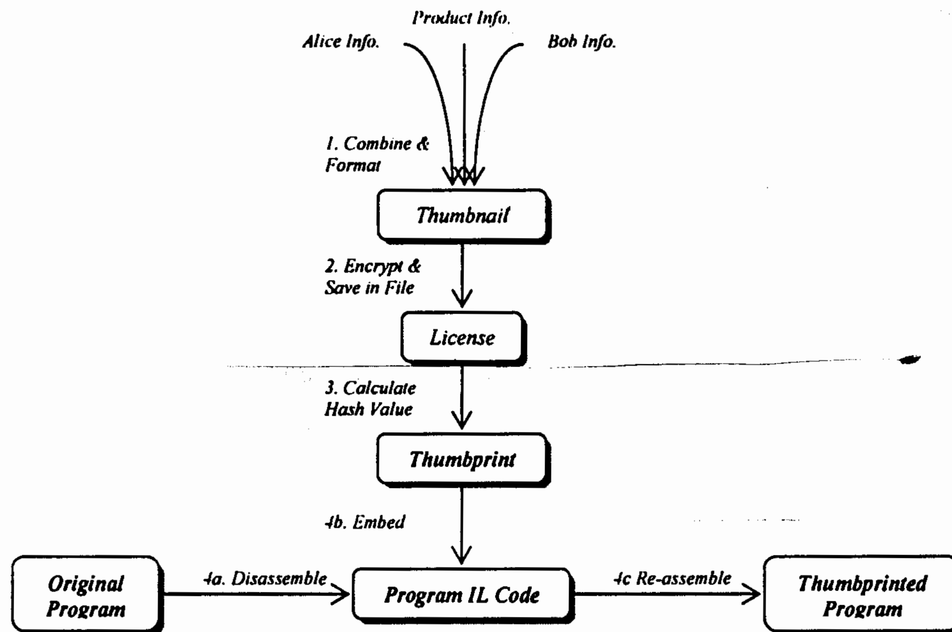


Figure 3.3 Thumbprint Encoding

3.3.1.2 IL Code Transformations

Step 4b in Figure 3.3 depicts embedding of a unique thumbprint into the IL code of original program. As described above, a thumbprint is a numeric value so embedding it into a program requires some property of code statements that will hold this number. For this purpose we choose *jump* instructions in IL code, which transfer program execution flow from one statement to another statement in the same method. The idea is to modify program flow by introducing forward and backward jump calls, such that, jump to a subsequent instruction means a binary 0 is encoded and jump to a preceding instruction means a binary 1 is encoded. Hence forming complete binary of a thumbprint number, as shown in Figure 3.4. Such transformations preserve the semantics of target .NET application. These transformations serve at the same time three purpose;

i. Encode binary of a thumbprint in forward and backward jump calls while maintaining program observable behavior
ii. Fabricate tamper-resistant dependencies among original program code and thumbprint related code
iii. Prevent reverse engineering by thwarting .NET disassemblers to generate high level language code of thumbprinted application

First step in performing code transformations is to construct a code block comprising of forward and backward jump calls constituting binary of a thumbprint value. If we consider 9_{10} (i.e. 1001_2) as a thumbprint value then its control flow graph of equivalent code block is shown in Figure 3.4.

Thumbprint = 9_{10} (i.e. 1001_2)

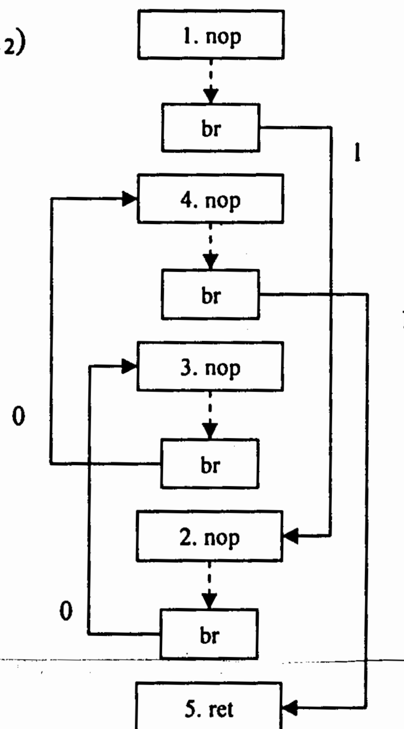


Figure 3.4 Control Flow Graph of Thumbprint Code Block

Above control flow graph consists of two different kinds of IL instructions, a) *nop*: no operation instruction, b) *br*: break or jump call instruction. *nop* does nothing but is used as a statement place holder and compiler simply transfers control to its subsequent instruction. As shown in figure above, dashed lines moving from *nop* to *br* instructions

are depicting normal execution flow. A *br* instruction accepts a label of target statement and instructs compiler to transfer the control to that statement within a method body. Solid lines in above figure, moving from *br* to *nop* instructions this depict crafted execution flow.

If we observe the numbers assigned to *nop* instructions, they represent the order of their execution. As shown above, first of all, the compiler will execute first *nop* instruction and its succeeding *br* transfers the control to second *nop* instruction. Its subsequent *br* transfers execution flow to a preceding *nop* instruction, and so on until compiler encounters method *ret* (*return*) statement. This way forward and backward jump representing 1's and 0's of a thumbprint are embedded. One group comprising of a *nop* and a *br* statement, encodes one bit of a binary number. So for a binary number consisting of four bits (i.e. 1001_2), we need four such groups. The order of execution of these groups is so far manually configured. An automatic *Encoder* can be set in place to do this task but that is not in the scope of this work.

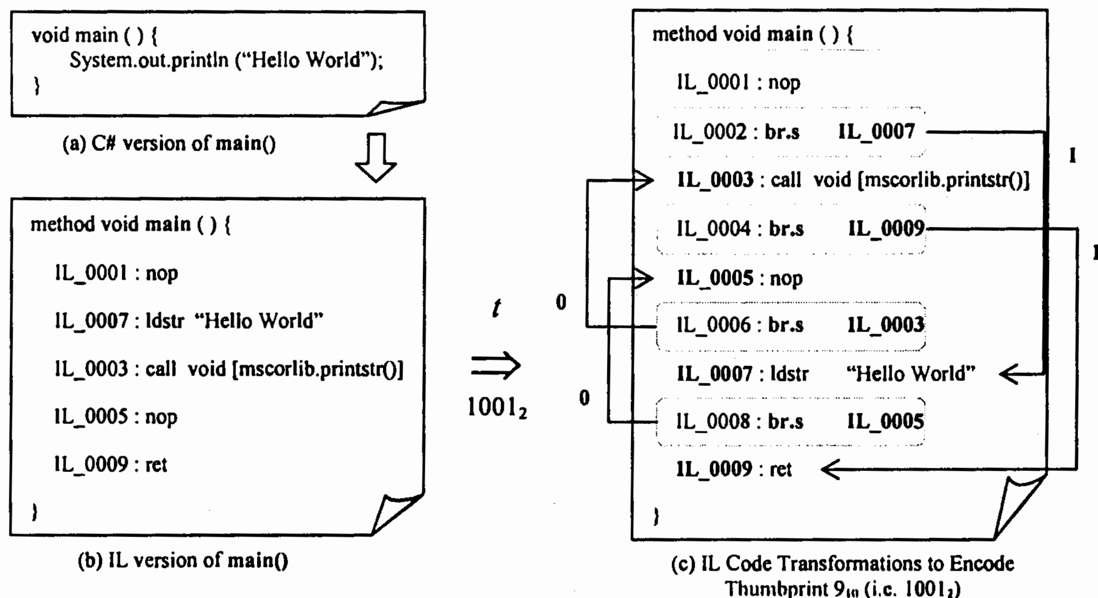


Figure 3.5 IL Code Transformations

Finally the *nop* instructions of above control flow graph will be replaced with available original program statements while observing the order of original program's instructions

(Figure 3.5). This way a complete binary of a thumbprint number is embedded into a method code while keeping its semantics intact.

Above figure demonstrate proposed IL code transformations through a trivial "HelloWorld" example, in which one line high-level language code of method *main* (Figure 3.4a) is disassembled to its IL-level code (Figure 3.4b). Then transformations *t* are applied on it to encode binary of thumbprint 9_{10} (i.e. 1001_2) (Figure 3.4c). Original IL code statements of Figure 3.4b are transformed to Figure 3.5c. The observable behavior of both blocks of code is similar because both contain same lines of functional code and compiler executes them in the same order.

3.3.1.3 Thumbprint Decoding

Once a thumbprinted instance of an application is produced and sold to its legal buyer along with its license file. Then vendor might need to extract the thumbprint from that instance in order to prove her product's ownership or to verify that her product is being run by its legitimate license holder. For this purpose the product user is required to provide a thumbprinted program and its valid usage license. Then process of thumbprint decoding goes through following steps, as shown in Figure 3.5.

1. Decode *thumbprint* from the thumbprinted program. Decoding process goes as follows:
   1a. Disassemble .NET program to its IL level code
   1b. Extract thumbprint from IL code by analyzing control flow graph of different methods' to see if they contain forward and backward jumps such that forming thumbprint's binary number
2. Generate a *thumbprint* by taking hash of encrypted contents of license file

Two thumbprint values are obtained from; 1) decoding thumbprinted program and 2) taking hash of license file contents. Both of these numbers will match if the program and its license file are not tampered. Vendor will query license server by inserting program's information like name of the original product and copy number of this thumbprinted instance. Server will provide the complete information that was preserved at the time of

this particular copy-sale instance. That information includes unique thumbprint value embedded in this program copy, its legitimate buyer's information and etc.

If first two thumbprints match with this value obtained from database and the buyer information is same as the current user information then this particular program is proved to be the ownership of original vendor and the user running the program is its legal user. On the other hand there can be following cases in which vendor may accuse the buyer:

- If user fails to provide valid usage license or the license provided has been tampered
- If the program being run is tampered and the thumbprint extracted from it does not match with the database thumbprint value
- If buyer's information acquired from license server is not same as the program user's information

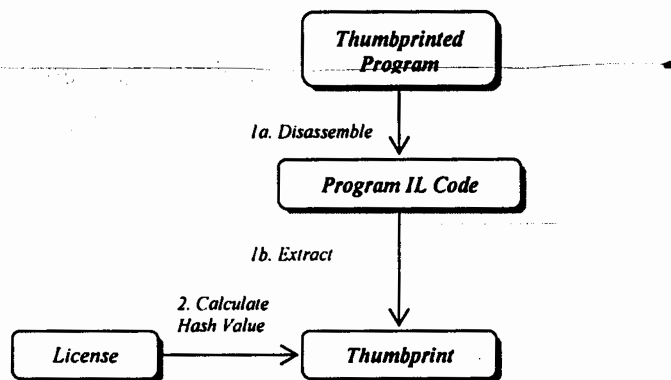


Figure 3.5 Thumbprint Decoding

3.3.2 Formal Model of Software Thumbprinting

Formal model of software thumbprinting is presented below. The notion used to formalize the context is inspired from Collberg et. al. [24].

DEFINITION 1 (SOFTWARE THUMBPRINT): Let W be a set of mathematical construct representing unique thumbprints, as $W = \{w_1, w_2, w_3, ..., w_n\}$ where $\forall w \in W : w_i \neq w_j$. Let $p(w)$ be the programmatic structure equivalent to a thumbprint such that $\forall w \in W : p(w)$.

DEFINITION 2 (COVER OBJECT): Let *P* be the set of copies of a source program P, as $P=\{p_1,p_2,p_3,...,p_n\}$ and each of these copies is considered as a cover object in which thumbprint *w* is to be embedded since that p_w is an encoding of a thumbprint $w \in W$ into $p \in P$.

DEFINITION 3 (THUMBPRINT ENCODING): Let t be a semantic preserving transformation to be applied on a method *m* of *p* to encode *p*(*w*) into it. If $M=\{m_1,m_2,m_3,...,m_n\}$ be the set of methods in P and M_p the total number of methods in P then *n* is the thumbprint encoding data rate.

E is the thumbprint encoder that constructs thumbprinted p_w by applying t iteratively on *n* methods in *M*, by selecting m_i at random.

$$E_{i=1}^{n}(t(m_i, p(w))) \rightarrow p_w \qquad (1)$$

So that the resultant p_w and original P are functionally equivalent as $p_w \equiv \mathrm{P}$.

Let *DB* be the storage medium for all information used in thumbprinting different elements in *P*. For each thumbprinting instance *DB* is updated as $\Delta DB = DB + (I_p, I_l, w, n)$. Where I_p is the product information and I_l is the license information. For any product P, a thumbprint *w* exclusively refers to one and only one copy-sale record in *DB*.

DEFINITION 4 (THUMBPRINT DECODING): Let *D* be the thumbprint decoder. While asserting the ownership, vendor needs to decode the thumbprint from claimed program p_w. For this firstly she will query *DB* against product information to get actual cop-sale information that was preserved at the time of its sale of this particular copy. This information includes original licensee's information, thumbprint encoded in this copy and encoding data rate.

$$DB(I_p) \rightarrow I_l, w_1, n \qquad (2)$$

Then D will analyze the program p_w for decoding thumbprint w from it, by detecting n instances of $p(w)$ from it.

$$D:(p_w, n) \rightarrow w_2 \qquad (3)$$

If decoder detects an instance of $p(w)$ in p_w then vendor successfully proves her ownership and can prosecutes the false claimant Bob.

Proposed methodology is described in terms of architectural and formal design. Next chapter will discuss its implementation with reference to a detailed case study.

Chapter 4

IMPLEMENTATION

4. Implementation

Proposed methodology discussed in previous chapter is implemented on a case study. Firstly, the development environment required to implement *software thumbprinting* is introduced. Then we run through successive stages involved in thumbprinting a .NET application as described in Chapter 3.

4.1 Environment

The environment to implement proposed methodology comprises of specific technology platform, several tools and some techniques, which are discussed below.

4.1.1 .NET Platform

Microsoft introduced a new development framework, intended as the unifying model for Microsoft Windows development. The framework is being extensively used for all Microsoft-based development, from Window-based, user applications, to web-based database applications. The framework, called .NET framework, is modeled after Sun Microsystem's Java system. Similar to Java, it also consists of byte code executed in a virtual machine, called CLR (Common Language Runtime). However, by introducing this virtual machine, the framework allows several high level languages to be compiled into a single abstract intermediate language, called Microsoft Intermediate Language (or shortly MSIL). This MSIL is then converted into platform-specific machine code.

4.1.1.1 Programming Language

The proposed scheme does not restrict its applicability to any single high-level .NET language like VB.NET, C#, J#, etc. Instead, an application written using any of .NET framework compliant language can be protected through proposed thumbprinting technique because thumbprinting plays at MSIL level code not higher level language code. High level language choice does not make any difference to thumbprint encoding and decoding mechanism. The only restriction being observed is that the final product

must be a valid .NET executable or library (i.e. *.exe* or *.dll*) which is to be considered as the distributable source.

4.1.1.2 Microsoft Intermediate Language (MSIL)

Microsoft Intermediate Language (MSIL) or simply *Intermediate Language* (IL) is abstract intermediate representation of .NET applications, intended for the *Common Language Runtime* (CLR). CLR is a runtime environment for .NET applications same as JVM is for Java applications. Visual Studio compiler first of all compiles high-level .NET language program into IL code and CLR compiles this IL into machine specific code. We will take the flavor of underlying ideas that forms the basis of our selection of MSIL as implementation language. The compelling features of architecture of MSIL are:

- Considering MSIL as implementation language make proposed scheme independent of source program language
- Flat structure of IL code helps encoding / decoding thumbprint without bothering about logical and physical file structure of an application
- IL instruction set provides lot of flexibility for code transformations to embed a numeric thumbprint in various ways
- One of the utmost advantages of using IL is that certain code constructs at IL level have no equivalent in high level language, so reverse engineering such constructs results no useful information

4.1.2 Tools

The toolset required to implement software thumbprinting over .NET applications comprises of IL code assemblers and disassembler. Moreover for evaluation purpose we have used an obfuscation tool and a third party reverse engineering tool. The assembler and disassembler are shipped with .NET framework hence reducing implementation cost overhead. The obfuscation tool used for evaluation purpose can also be employed to strengthen the proposed protection scheme by obfuscating thumbprinted instance of a product before distributing it.

4.1.2.1 IL Code Disassembler (*ildasm.exe*)

Microsoft .NET framework is shipped with two built-in tools called *ilasm* (IL Assembler) and *ildasm* (IL Disassembler). *ildasm* disassembles an executable file like .exe or .dll and produce its IL code. It can be found in directory *[DIR]:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin* on a system where .NET framework is installed. More details regarding the use of *ilasm* can be found in Appendix-B. Figure 4.1 is displaying a shoot of disassembler's window which is displaying the disassembly of *HelloWorld.exe*.

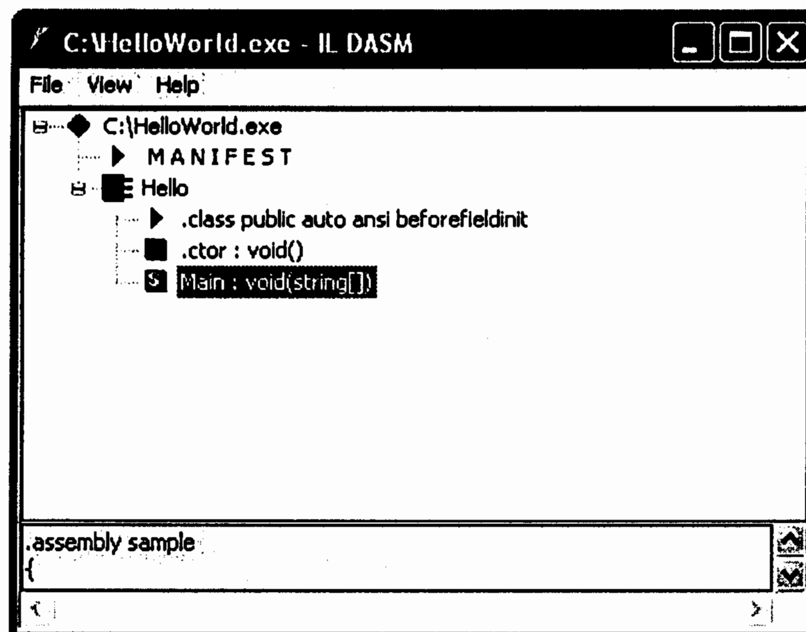


Figure 4.1 Disassembling *HelloWorld.exe* through *Ildasm*

4.1.2.2 IL Code Assembler (*ilasm.exe*)

Ilasm assembles valid IL code to build an executable file like .exe or .dll. It can be found in directory *[DIR]:\WINDOWS\Microsoft.NET\Framework\v2.0.50727*. In order to execute this tool, run system command prompt or use command prompt companioned with Visual Studio. In our implementation we used command prompt accompanied with our Visual Studio 2005 framework. In order to re-assemble IL code to an executable file we move to the above directory and run *ilasm* by specifying the name of file containing complete IL code which exhibits valid syntax and semantics. Second parameter is the

output switch as '*/exe*' that instructs the assembler to build an *.exe* file from this IL code. Figure 4.2 shows a successful run of *ilasm* to build *HelloWorld.exe* from *HelloWorld.il*.

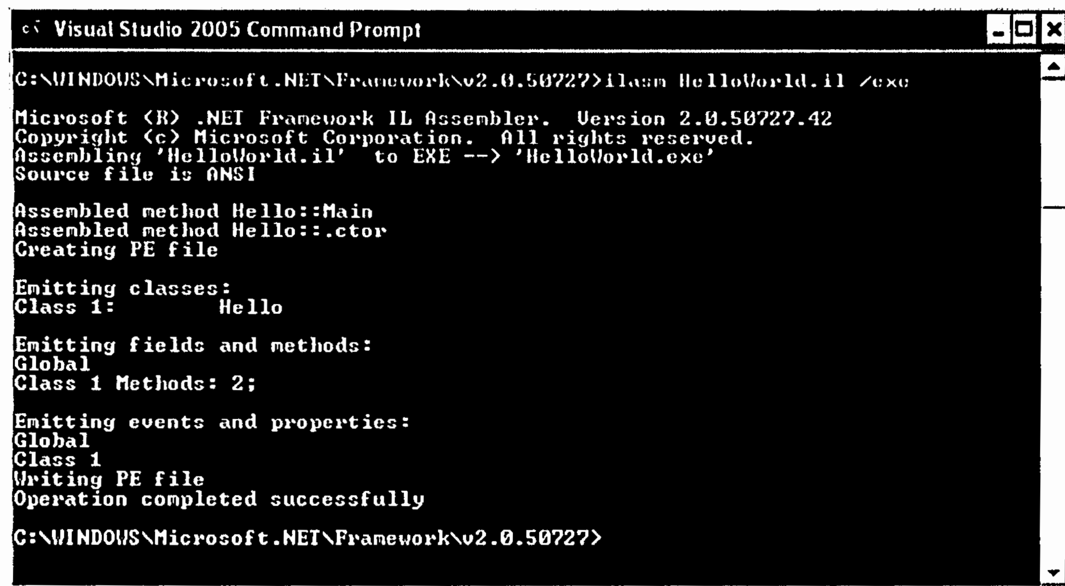


Figure 4.2 Re-Assembling Modified *HelloWorld* Program using *Ilasm*

4.1.2.3 Marka

We have developed Marka to automate the process of software thumbprinting. The tool is developed in VB.NET. Current version of Marka partially implements the processes of thumbprint encoding and decoding. Vendor is required to run Marka for each copy-sale instance of a product. Figure 4.3 shows a screen shot of Marka. It performs following functionalities for thumbprint encoding:

- Takes license information as input and create a license file
- Generates thumbprints from thumbnails of license information
- Preserves product license information in database

It automates thumbprint decoding process in following ways:

- Takes license file as input and computes back the thumbprint
- Helps searching database for licensee's information against particular product information

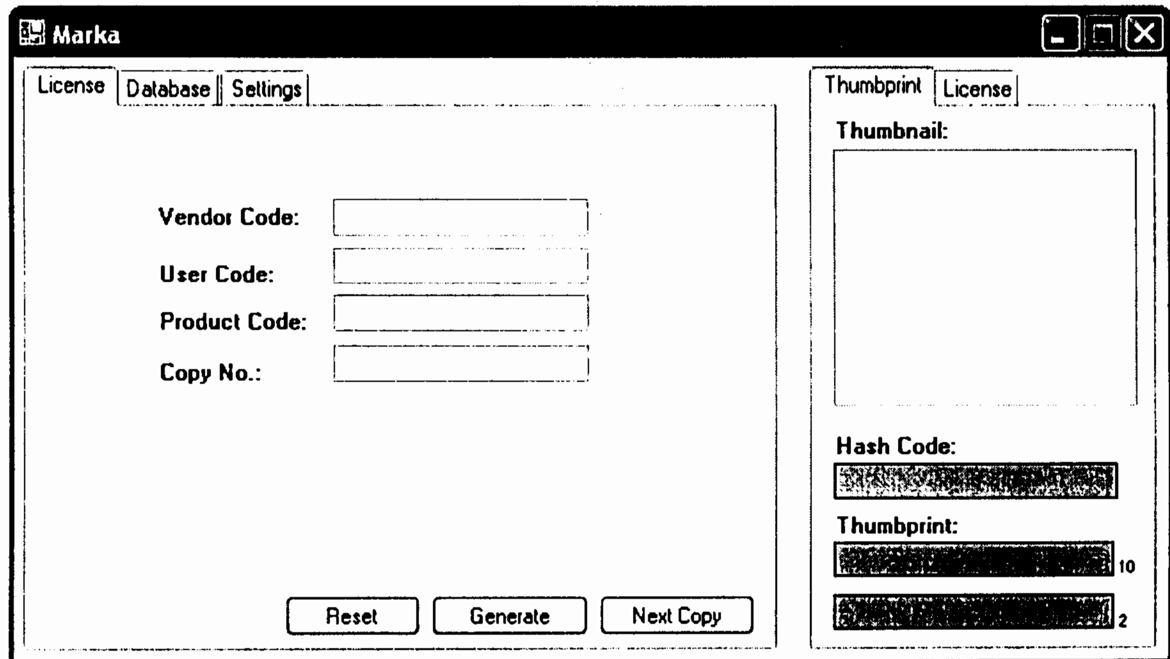


Figure 4.3 Software Thumbprinting Tool Marka

So far Marka does not perform thumbprint encoding and decoding operation on program IL code. This is set out of the scope of this project. Tool employs most of standard hashing and encryption algorithms and user can select some particular algorithm depending upon requirements.

4.1.2.4 Dotfuscator

Dotfuscator (or .NET Obfuscator) protects intellectual property by making it extremely difficult to be reverse engineered by decompiler tools. Microsoft uses Dotfuscator Professional to obfuscate their .NET code [19, 20]. Microsoft has also introduced a simplified version of .NET obfuscator for developers which is shipped with Visual Studio. Dotfuscator can be used to obfuscate thumbprinted instances of an application to minimize the chance of its reverse engineering.

4.1.2.5 .NET Reflector

.NET Reflector is a third party decompiler for .NET applications. It allows browsing and searching the metadata, IL instructions, resources and XML documentation stored in a

.NET assembly. Reflector is one of the oldest tools that are running with .NET Framework's since its beta release [19]. Hence this is considered to be a reliable reverse engineering tool used to explore .NET applications. Reflector takes a valid .NET .exe or .dll file and disassembles it to produce its IL code, VB.NET code, C# code and J# code. We used this tool to evaluate the strength of proposed methodology. We have experimented several thumbprinted executables with Reflector and found that it failed to reverse engineer these applications to high-level language code.

4.1.3 Techniques

.NET framework exposes a complete set of standard cryptographic algorithms. Proposed scheme employs hashing and encryption techniques from .NET *Cryptography* API for generating thumbprint value and creating license file respectively. MD5 and SHA are most widely used hashing algorithms which are supported by this API. Proposed methodology can be implemented using any of these hashing techniques. Purpose is to run it on a variable length piece of data (i.e. license information) in order to produce a fixed length representation of that data. This fixed length representation is called a *thumbnail.* It is impossible to reproduce original information from its thumbnail. Furthermore a numeric hash value is obtained from this thumbnail which serves as a *thumbprint* that is to be encoded into the program as an ownership mark. Secondly, this API provides wealth of encryption techniques like AES, RC2, DES, 3-DES etc. The fixed length thumbnail formed from license information is encrypted before saving it into a license file. We have developed a tool in .NET that automates the process of thumbprint generation and license file creation. This tool can be setup to choose any of .NET frameworks' supported hashing and encryption technique. Further we demonstrate all these ideas through a case study.

4.2 Case Study: Tower of Hanoi

The case study selected to implement proposed thumbprinting scheme, is a well known problem called '*Tower of Hanoi*'. This classic problem has a standard algorithmic solution which has been implemented in almost every programming language. One main

reason for choosing *Tower of Hanoi* as our benchmark problem is that it has notable overhead on computing resources due to recursive nature of its algorithm. Implementing proposed scheme on such application yields significant results in terms of program performance, size and lines of code. High-level language code and corresponding IL level code of *Hanoi* application is presented in section B.1 of appendix B. The application implements *Tower of Hanoi* algorithm in VB.NET. It comprises of single window application form (as shown in Figure 4.4) and a user class named *Hanoi*.

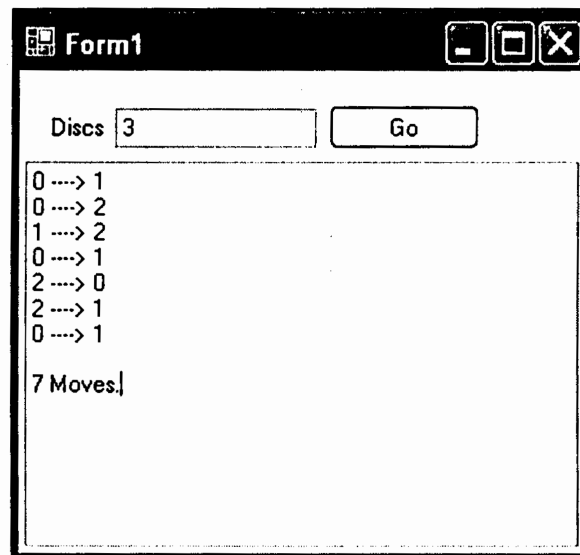


Figure 4.4 Tower of Hanoi Application

User inputs total number of discs required to move from first post to third and clicks *Go* button. Application assumes that all the discs are initially placed on first post and are required to be moved to the second post while using third post as temporary. Class *Hanoi* that implements the algorithm is called to move the stack of discs. It provides a list of moves required to successfully transfer all discs from one post to another.

In order to exercise software thumbprinting we build the scenario such that Alice is the owner of this .NET application, called *Hanoi*. She is going to sale it to its first buyer Bob. To protect this application using proposed methodology, she has to employ the process of software thumbprinting described in section 3.3. Following steps execute a complete copy-sale instance of *Hanoi*. These steps can be referred to extend implementation of proposed thumbprinting scheme on small-to-large scale .NET applications.

Step 1: Specify Hashing and Encryption Technique

Firstly, Alice runs Marka and specifies which of hashing and encryption techniques will be used to process license information. For this she clicks to Settings tab in main window of Marka. Hashing techniques available are SHA1, SHA256, SHA384 and SHA512. Encryption techniques supported are DES, RC2, Rijindeal and TripleDES.

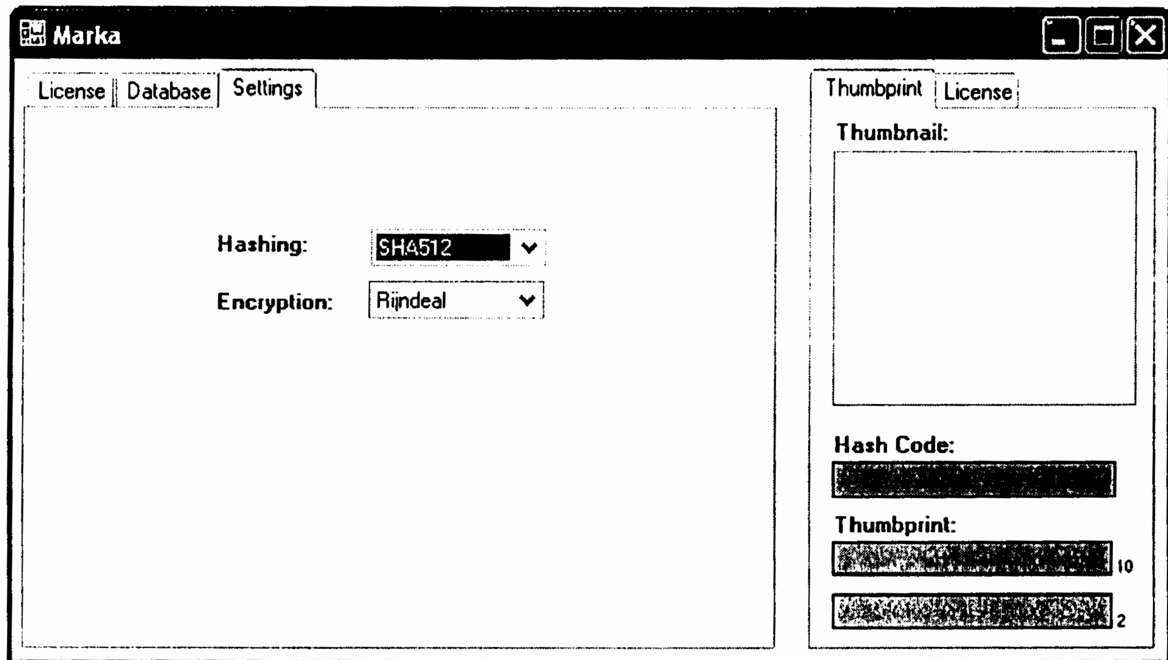


Figure 4.5 Hashing and Encryption Setting in Marka

Step 2: Generate Thumbprint and Create License File

Then Alice provides license information including Vendor Code, Buyer Code, Product Code and Product Copy serial number and clicks *Generate* button. Marka combines this information and computes a fixed size thumbnail by applying selected hash algorithm on it. The length of thumbnail string depends upon hashing technique like SHA1 yields a thumbnail of 128 bits (16 bytes), SHA256 yields a thumbnail of 256 bits (32 bytes) and SHA512 yields a thumbnail of 512 bits (64 bytes). The thumbnail is shown in text area visible on the *Thumbprint* tab as shown in Figure 4.6. Then Marka encrypts the thumbnail using specified encryption techniques and saves it into a license file (Figure 4.7).

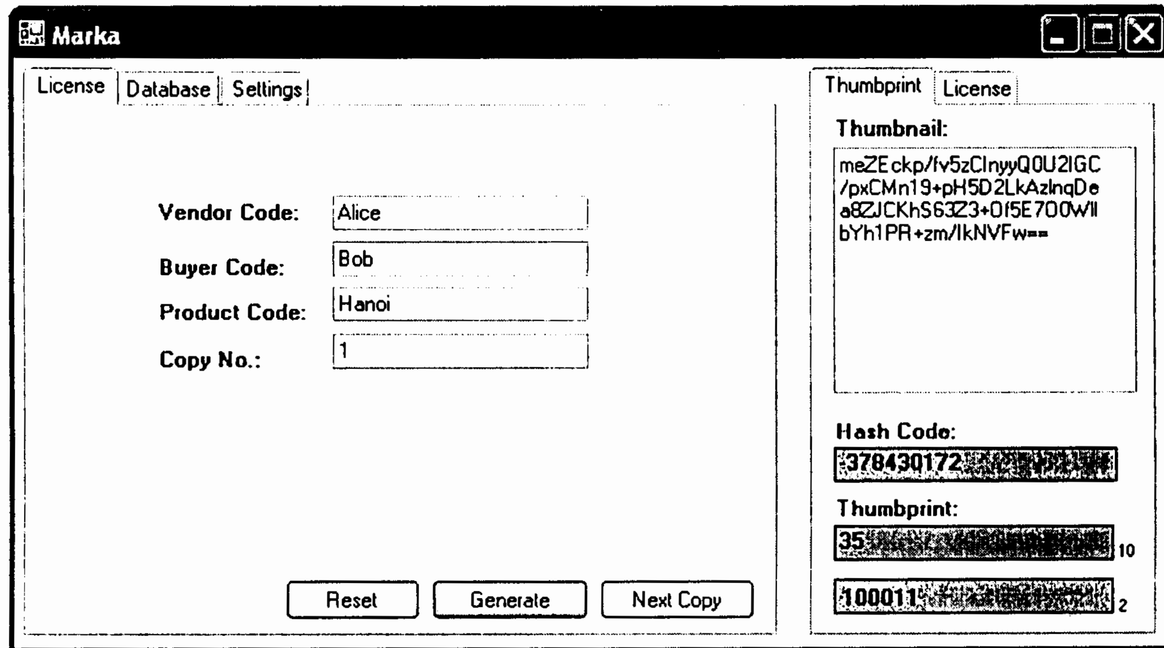


Figure 4.6 Thumbnail Generation using Marka

Figure below displays the encrypted thumbnail that is saved in a license file. The name of license file comprises of product code concatenated with its copy number.

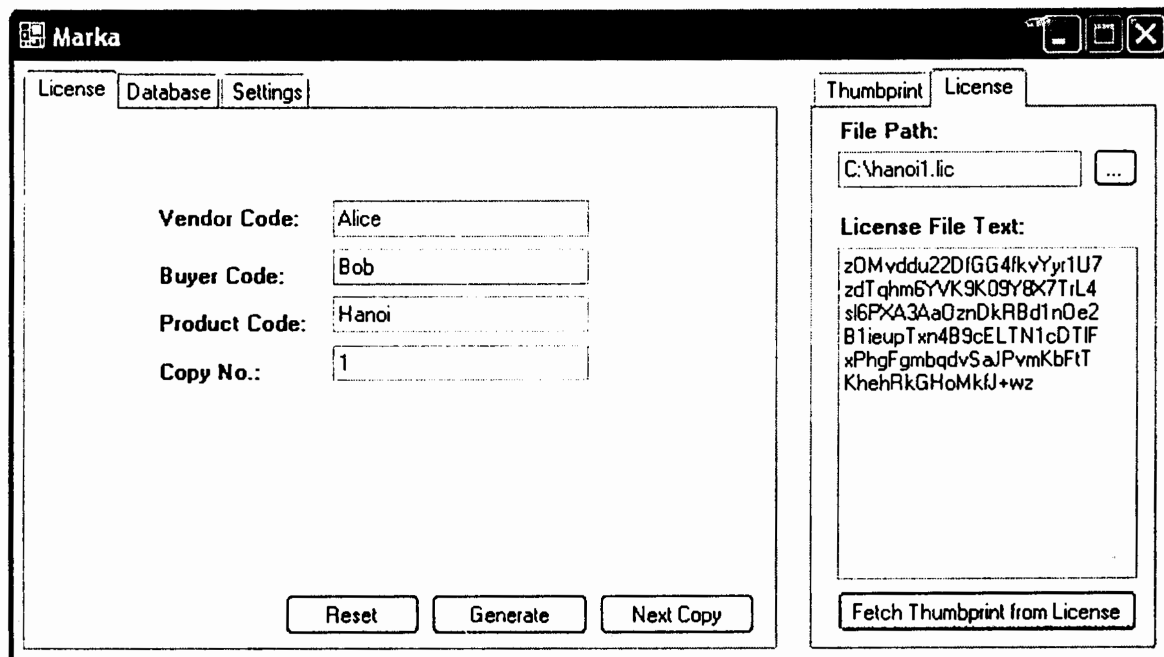


Figure 4.7 License File Creation using Marka

Marka computes hash value of this encrypted license file. The hash number consists of around 10 - 12 digits and may include both positive and negative numbers. Such long number is not as easy to embed into methods of IL code so a simple logic is defined. That is to sum up the digits of a hash code. This way thumbprint becomes a shorter number like 35_{10} (or 100011_2) as shown in Figure 4.5.

Before encoding this thumbprint manually into *Hanoi* IL code, Marka saves all this information in the database (Figure 4.8).

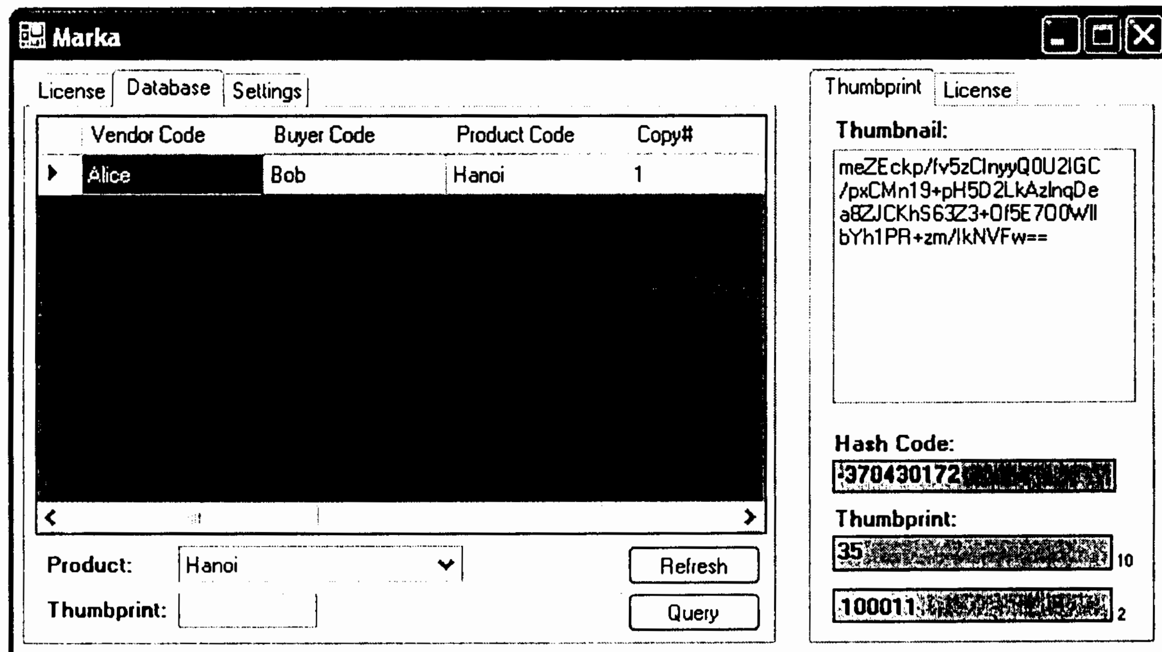


Figure 4.8 License Database Maintained by Marka

Step 3: Encode Thumbprint

Finally Alice encodes this thumbprint into IL code of Hanoi. In order to do this, first of all she disassembles *Hanoi.exe* using *idasm.exe*. Complete IL code of Hanoi program is saved in single file named *Hanoi.il*. Then she constructs a control flow graph equivalent to thumbprint value as show in Figure 4.9. Alice then selects *n* methods in which she will embed the thumbprint by mapping original IL code statements to the nop instructions of thumbprint control flow graph. For instance she defines n=1 (as thumbprint encoding

data rate) and embeds thumbprint into one method, i.e. *DoMoves()*. The thumbprinted code of this method is given in section B.1.3 in appendix B.

Thumbprint = 35_{10} (i.e. 100011_2)

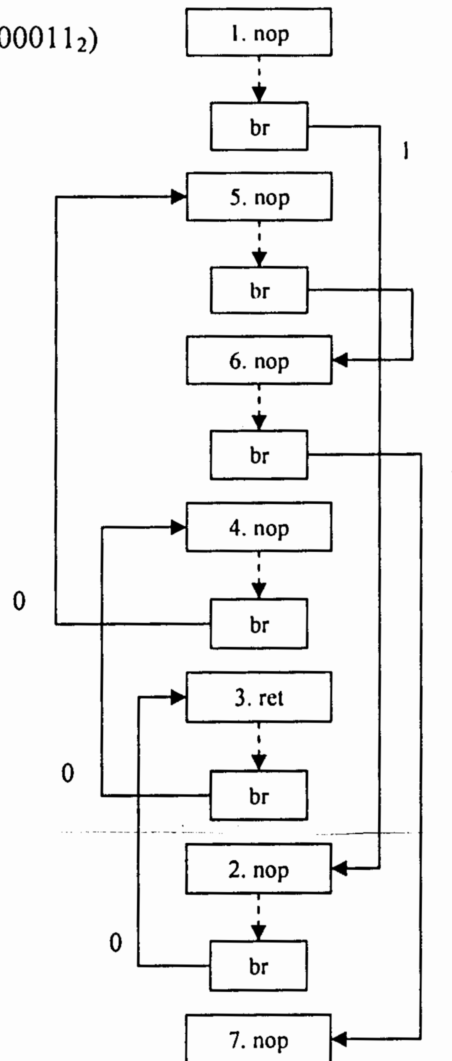


Figure 4.9 Control Flow Graph of Thumbprint 35_{10} (100011_2)

Applying code transformations on IL code of *DoMoves()* method according to above control flow graph will embed thumbprint in it. Then this modified IL code block is combined with rest of IL code of *Hanoi.il* to create a modified IL code file. Then Alice re-assembles modified *Hanoi.il* into *Hanoi1.exe* using *ilasm.exe*. Finally she distributes the resultant thumbprinted *Hanoi1.exe* to Bob along with its license file *Hanoi1.lic*.

Step 4: Decode Thumbprint

Alice might need to extract thumbprint from a copy of *Hanoi.exe* which Bob would be running. First she requires its usage license file from Bob and if he provides then it is given to Marka as an input. She clicks on *Fetch* button and tool computes back the thumbprint from it as shown in Figure 4.11. Alice then query database by providing Product Code as *Hanoi* and Thumbprint as 35_{10}. Marka searches database and displays the complete information against this copy-sale instance. If the buyer is same as the one identified by database, i.e. Bob, then he is authenticated as a legal user of that particular copy of *Hanoi.exe*. Also if the thumbprint value recorded in database is equal to the one computed from license file then it proves that the license file is valid and is un-tampered.

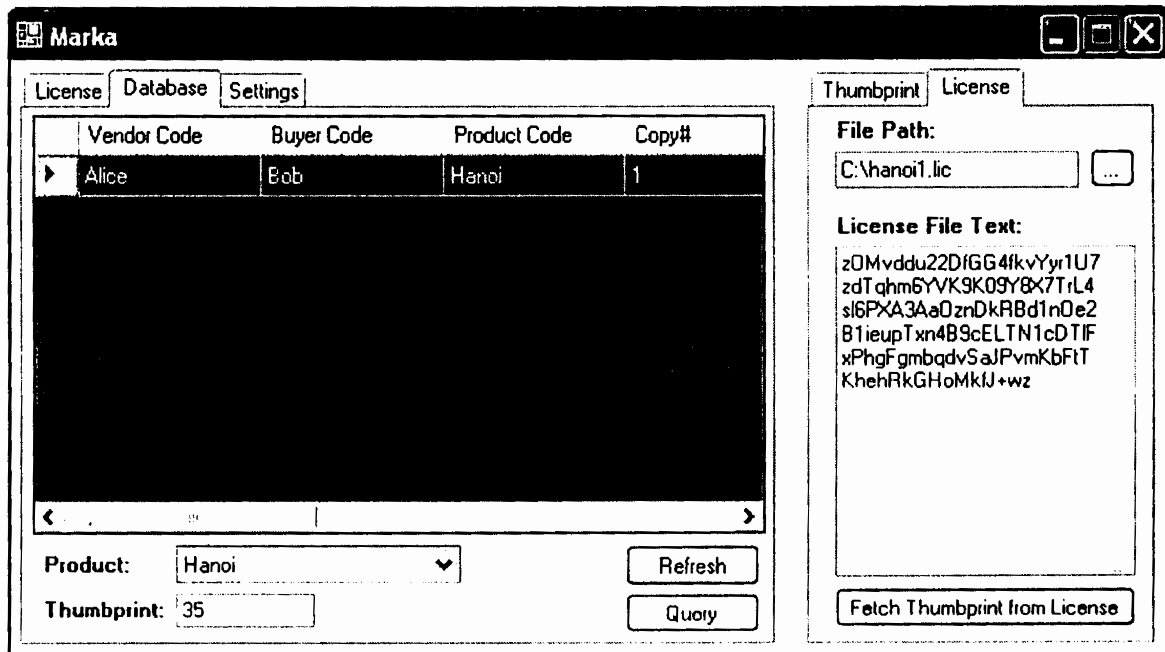


Figure 4.10 Decoding Thumbprint

Lastly Alice manually analyzes the IL code of Hanoi program which Bob is running. If n number of its methods contain thumbprint code block and control flow graph constructed from that block matches with the one created while encoding 35, then the program is proved to be un-tampered and all these authentications confirm that Alice is the buyer of *Hanoi.exe* and Bob is a legal user of this product.

4.3 Issues Regarding Automating the Proposed Scheme

Further work in this direction can lead to develop a thumbprinting codec to automate the process of thumbprint encoding and decoding. The codec may extend Marka's functionality. Marka automates basic processes of generating thumbprints and creating license files. Furthermore, following are the basic concerns in order to automate the encoding process;

- The trickiest part of thumbprint encoder will be the construction of a control flow graph which will essentially be a programmatic construct, equivalent to the thumbprint's binary value. Currently we are doing this manually (as shown in Figure 3.4) and the flow graph creation is so far almost based upon heuristics.
- Next task of the encoder will be to map the original program's instructions onto the *nop* and *br* instructions of control flow graph. This will perform the IL code transformations required to encode a thumbprint's binary value. An issue arising at this point can be that from which statement of original program the encoder should start mapping the CFG's *nop* instructions. In manual process we start by mapping the last *ret* statement of the CFG and the original method. Then we map 2nd last *nop* of CFG with 2nd last statement of original program and the third last and so one. At the same time CFG's *br* statements are placed at required positions in original program. The encoder can adopt the same or even better way of mapping thumbprint CFG onto original program instructions. The important point is to remind the same position at the time of decoding, i.e. where from the decoder should start locating a thumbprint? Another consideration can be to preserve some identification of thumbprinted methods in the database which the decoder will analyze in order to extract the thumbprint from thumbprinted program.
- Once the encoder have encoded a thumbprint into *n* methods of program by mapping CFG's *nop*'s onto original program statement and introducing *br*'s then the decoder will take the thumbprinted program and will start analyzing its thumbprinted methods at the same position where thumbprint CFG was mapped. This way decoder will decode 1-n instances of thumbprints in the program.

4.4 Limitations

Proposed scheme has to undergo certain limitations in our scheme too, these limitations can be removed in future works. Following are the limitations in presented work.

- Thumbprint encoding and decoding scheme is manual so far which needs to be automated, the issues regarding its automation are listed in above which can be referred to develop a software thumbprinting codec.
- Technique involves addition of digits of the calculated hash value of thumbnail to make a shorter numeric thumbprint. It restricts the thumbprints values to very small number, so the method of thumbprint generation should be improved.
- Proposed scheme is embedding watermark through jump calls (i.e. IL *br* instruction) which may affect the performance of large applications. So there should be some better way to insert the watermark so as to make it more robust.

Chapter 5

RESULTS AND EVALUATION

5. Results and Evaluation

For a software watermarking technique to be effective against software piracy and copyright infringement, it should be resilient against determined attempts at discovery and removal of watermark. Very little work has been done on evaluating the strength of software watermarking systems. Such limited evaluation makes it difficult to adequately compare the various techniques. In this chapter we first evaluate proposed thumbprinting scheme for its advantages over previously proposed techniques. Then we discuss how better it employs robustness and tamper-proofing attributes of an efficient protection scheme. Further we present its experimental evaluation based upon comparison of results of proposed thumbprinting scheme and branch function-based watermarking technique [17].

5.1 Thumbprinting VS Prior Arts

Most of the previous software-based protection mechanisms aim to protect Java and Assembly language applications [14, 15, 16, 17, 21, 23]. These techniques are either specifically discussing Java or Assembly language constructs or their implementation (if found) is in Java or Assembly language. Quite a few of these solutions are applicable to .NET application while other schemes are either strictly bound to their specific language constructs. Proposed thumbprinting scheme has basic similarity with the idea of branch function-based watermarks presented by Collberg et. al. [17]. Branch function-based watermarks are the most investigated technique among other recent protection schemes [17, 27]. This is the reason why we have chosen this scheme for comparative study. Thumbprinting encodes 0's and 1's of a watermark in backward and forward jump calls like they do. Their scheme has one major drawback that the number of calls to branch function is quite high. This not only makes the scheme un-stealthy but also raises the chance of single-point failure. An adversary can easily locate and tamper with the single branch function to distort the watermark. At the same time proposed thumbprinting scheme radically differs from their technique in several ways, like 1) it encodes thumbprints at method level where as a branch function is for the whole application, 2) it is more obscure because there is no more branch functions introduced, 3) its data rate is

higher at low size and space overhead, 4) there is no chance of single-point failure due to insertions of watermark at multiple places and above all, 5) it considers specifically IL language constructs.

M. Chen in [40] by the first time highlighted the need of combining license distribution with other technical protection schemes. They merely explored the basic software and hardware-based protection mechanisms but did not discuss any algorithm. Software thumbprinting realizes their idea of binding license enforcement with other technical solutions like watermarking and tamper-proofing.

5.2 Thumbprinting Thwarts .NET Reverse Engineering Tools

Another major advantage of introducing jump calls in the way proposed is that it causes .NET reverse engineering tools like *Reflector* and *Salamander* to malfunction, if used to generate higher level language code from disassembly of a program. None of the previous protection schemes offer this level of code protection for .NET applications. We have run several programs after introducing breaks at certain points in their IL code and successfully restrained these tools to generate code in VB.NET, C#, etc. The reason is that .NET CLR is unable to find higher level language constructs equivalent to such a abnormal sequence of IL code instructions. Taking advantage of this limitation of .NET runtime, we have gained a reasonable level of code protection because after this the attacker has limited or no access to the higher level language code. He has to keenly observe the flattened IL code instead which requires a lot of time and resource overhead.

5.3 Robust and Tamper-proofed Thumbprints

The idea behind proposed thumbprinting scheme is to embed a unique number as a watermark in forward and backward jumps sequence of a .NET program's IL code instructions. The reason to choose IL code for encoding thumbprints is its flattened structure that makes static program analysis difficult. If we have a look into the IL code of an ordinary .NET program, we will find several similar jump calls which construct its actual control flow. Introducing such innate transformations make it hard for an adversary to figure out the secret behavior of the program. More over the thumbprints

encoded in through proposed IL code transformations are tamper-proofed because any change in code at IL level or at higher level will disrupt the ultimate IL instructions' labels (like IL_003, etc. in figure 3.5). This will cause jump calls to hit invalid targets and results in an malfunctioning program. This way quite obscure and rigorous dependency is created between thumbprint and original program instructions. More over the data rate *n* (number of times thumbprint is embedded) can be any number for which it becomes quite intricate for an attacker to sort out all the hidden thumbprints from IL code. Even after revealing maximum number of thumbprint occurrences, it is hectic to remove all of them while keeping program's semantics intact. Hence the thumbprint will remain in the program or other wise program will no more function as intended. As per our exploration, there isn't any debugger for dynamic program analysis of IL code. Proposed scheme ideally preserves semantics of the program without forsaking its performance.

5.4 Experimental Evaluation Framework

To evaluate the strength of a watermarking technique with respect to a particular property requires a variety of different experiments. In these we use one or more .Net application which varies in size and complexity.

Prior to running our experiments we will check both application and make sure the no obfuscation scheme is apply on these programs. This step is performed to create the baseline for our size and performance experiment which will eliminate any effects.

5.4.1 Evaluation Model

The strength of a software watermarking algorithm or techniques is evaluated based on a well defined threat model. The threat model describes tools and techniques generally employed by an attacker. Such attacks are categorized as manual, automated and blended. In a manual attack the software is analyzed and modified by hand using reverse engineering techniques. An automated attack is characterized by the use of tools which automatically apply an attack such as the DeCSS script used to disable DVD encryption. The most common form of attack, the blended attack, uses both manual and automated techniques to disable the protection. For example, dissemblers, debuggers, and

recompiles are often used by the attackers to interactively explore applications. Information gleaned from this analysis can be used to develop automated tools for disabling the protection on all copies of the program.

Since an attacker has full control over the execution of the software, it is generally believed that given "enough" time, effort and/or resources, a sufficiently determined attacker can completely break any software protection technique. With this in mind, most techniques are deigned to make the cost of the attack as high as possible. This can be accomplished by making the attack costly to carry out or by requiring an attack which degrades the performance of the software to an unacceptable level. For example, a software watermarking algorithm could be considered effective if the attack required to destroy the watermark also slows down the un-watermarked program such that it no longer has any economic value.

The threat model specific to software watermarking algorithms include four attacks:

- Additive
- Distortive
- Subtractive
- Collusive

To illustrate the attacks consider Alice and Bob. Alice produces a program P which contains her watermark W. she sells a copy to Bob not aware that he wants to illegally redistribute the program. In order for a Bob to successfully redistribute P he needs to destroy the watermark W.

Additive Attack: In an additive attack Bob embeds his own watermark W' into Alice's program P. by doing this Bob has made it difficult for Alice to prove ownership since she will have to show that her watermark was embedded prior to Bob's.

Distortive Attack: In a Distortive attack Bob applies a series of semantics-preserving transformations to P in an attempt to destroy W. for Bob's attack to be successful the watermark must be unrecognizable while preserving program functionality.

Subtractive Attack: In a subtractive attack Bob analyzes the (disassembled/recompiled) program P to identify the location of the watermark and to remove all or part of it. As with a Distortive attack the original functionality must be preserved for the attack to be successful.

Collusion Attack: In a collusive attack Bob obtains two different fingerprinted programs P1 and P2. Because the programs only differ in their fingerprints, Bob compares the programs to identify the location of the fingerprints.

5.4.2 Evaluation Properties

Through the study of software watermarking algorithms we have compiled the following properties which we believe aid in evaluating the strength of an algorithm [23, 36, 37].

Data-Rate: The number of times a watermark is embedded in a program

Overhead: The decrease in performance and/or the increase in program size and space.

Stealth: The degree of similarity between the watermark and the original code. Stealth can be characterized as either statistical or visual indefectibility.

Robustness: The ability to withstand the four attacks described in the threat model.

As with any software protection technique, the design of a software watermarking algorithm generally requires a trade-off between the various evaluation properties. For example, the embedding of a larger watermark could decrease the level of stealth and increase the size and memory overhead. In the same way to, to increase the robustness it is necessary to have larger data rate by increasing number of watermark insertions. But higher data rate at the same time causes low stealth and larger program size. Hence, a thorough evaluation of the known software watermarking techniques based on the above properties makes it possible for a developer to choose the appropriate algorithm for the required protection level.

5.5 Branch Function

A branch function is a function that is called in the normal manner, but which manipulates its return address such that, when it returns, control may be transferred to an address different from the original call site [38].

Consider a program containing a particular set of unconditional jumps of interest, at locations a1...........an, with targets b1........bn respectively, i.e., the instruction at location ai is

ai : jmp bi where 1 < i < n

With branch functions, we replace each of these jumps by a call to the branch function f, resulting in code of the form:

ai : call f where 1 < i < n

The function f uses the return address to figure out the location ai (1<i< n) it was called from, and then uses this information to change its return address to the value bi. When it subsequently executes a ret instruction, therefore, control is transferred to the original target bi [17]. This situation is illustrated in figure 5.1.

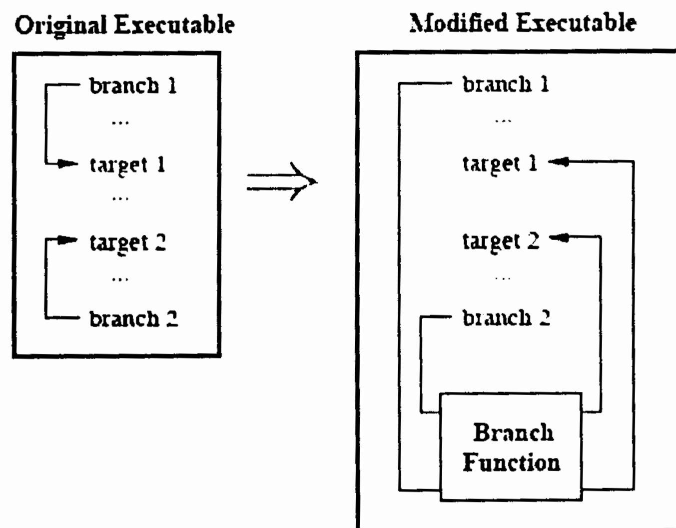


Figure 5.1 Branch Based Watermarking

We have used comparative study with other watermarking algorithm and technique chosen for this purpose is the latest state of art technique proposed by Giner in 2006 [39]. This technique discusses the watermarking using branch method which resembles closely to our proposed technique as we are also inserting forward and back jump in the code. we have checked the above said characteristics for software evaluation and then compared the results, difference between two techniques are presented below.

5.6 Experimental Results

We have chosen Tower of Hanoi a famous example as a case study to evaluate the results of both techniques.

5.6.1 Approach

Following steps are taken while evaluating the two techniques using Tower of Hanoi.

- Implementation of Tower of Hanoi using Branch function watermarking technique
- Calculation of result for Branch Function Scheme
- Implementation of Tower of Hanoi using Our proposed technique
- Calculation of result for Proposed Scheme
- Comparing the results of both schemes

5.6.2 Results

In coming lines we have presented the results which are divided into three categories.

- Results before Implementation
- Results with Thumbprinting Scheme
- Results with Branch Function Scheme

5.6.2.1 Results before Implementation

Tower of Hanoi is implemented in c# language and here we presented the results for parameters discussed above before the implementation of any of the two schemes

Parameter Name	Measurement
1. Size of Program	32768 bytes
2. LOC (MSIL)	2361
3. Performance	5 Disks takes 60 Mille Seconds
4. System Classes	5
5. User Classes	3
6. Total Method	57
7. User Method	11
8. No of Method to be Watermarked i-e Data Rate	$n = M_P / 2 = 5$

Table 1: Results Before Implementation

5.6.2.2 Results with Thumbprinting Scheme

Implementation of case study is given in Appendix-I with complete coding details too. Here we have shown results

Parameter Name	Measurement
1. Size of Program	24576 bytes
2. LOC (MSIL)	2361+20
3. Performance	5 Disks takes 63 Mille Seconds
4. System Classes	5

5. User Classes	3
6. Total Method	57
7. User Method	11
8. No of Method to be Watermarked i-e Data Rate	$n = M_P / 2 = 5$
9. Tamperproof	Obfuscated *Pw* through Dotfuscator Community Edition. *Pw* retained exact thumbprints intact. Hence proposed scheme is resistant against transformation attacks

Table 2: Results with Proposed Scheme

5.6.2.3 Results with Branch Function Scheme

Implementation of case study is given in Appendix-I with complete coding details too. Here we have shown results.

Parameter Name	**Measurement**
1. Size of Program	32768 bytes
2. LOC (MSIL)	2361+16
3. Performance	5 Disks takes 63 Mille Seconds
4. System Classes	5
5. User Classes	3

6. Total Method	57+1
7. User Method	11+1
8. No of Method to be Watermarked i-e Data Rate	1
9. Tamperproof	As branch function is easily visible and detectable even it is inserted as low level (IL). So branch function is not temper-proofed.

Table 3: Results with Branch Function

5.6.3 Results Analysis and Discussion

We have discussed the result analysis for LOC and Size and Performance in coming paragraphs.

5.6.3.1 LOC

Line of code is a software metric used to measure the size of a software program by counting the number of lines in the text of the program's source code. LOC is typically used to predict the amount of effort that will be required to develop a program, as well as to estimate programming productivity or effort once the software is produced.

In our scenario LOC will help in calculating the effect on LOC by adding watermark using both the approached. As from the figure 5.1 shown below it is clear that our approach take much line but that is because of dependency with data rate as both data rate and LOC are directly proportional so when ever we will

increase data rate in branch function then its LOC will increase drastically as every branch function roughly add 10 line in a smaller program.

5.6.3.2 SIZE

Size is another software metrics used to calculate the total size of software in terms of kilobytes. Size metrics is typically used to predict the amount of space occupied by the program on hard-disk.

We have calculated the Size of program before implemented any watermarking scheme then size was calculated after the implementation of two schemes. We have shown the difference through graph in figure 5.2. We can easily see through graph that proposed approach have a better impact on software size as compare to branch based watermarking. There are two reasons for this one is that we are using the existing structure of the program without adding much code and another we are implementing our approach at assembly level which has lesser impact on size as compare to implementing the scheme at high level language.

5.6.3.3 Performance

Performance is the major issue for every watermarking technique same is the case with ours. Performance is generally measure as time taken by the application while performing certain task. Our example takes disks out from the applet one by one and when we increase the disk then the time taken to draw them out will also increase. We have fixed the disk to 5 and check the time impact on both approaches. There was minor difference from the original as we can see in the figure 5.3. But one thing is clear that this minor difference will change into huge for large scale application where time matters.

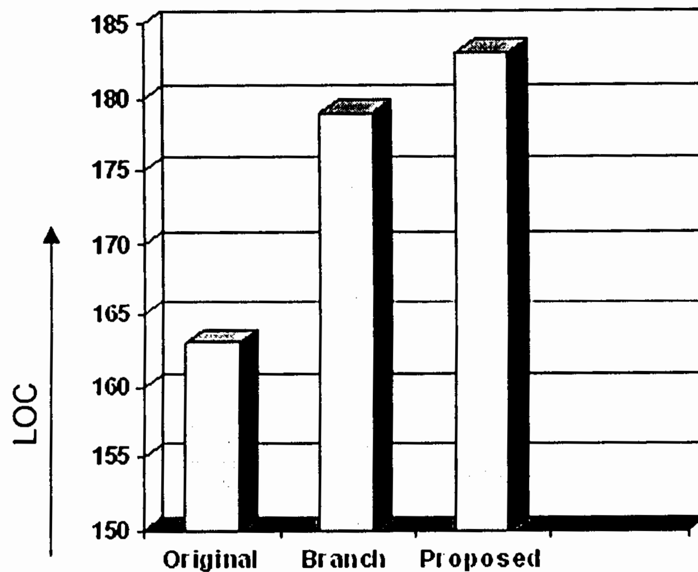


Figure 5.2 LOC

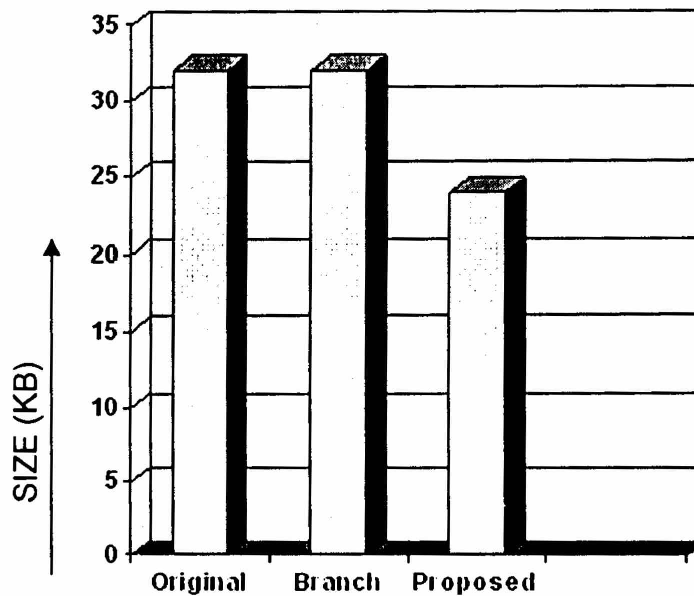


Figure 5.3 SIZE

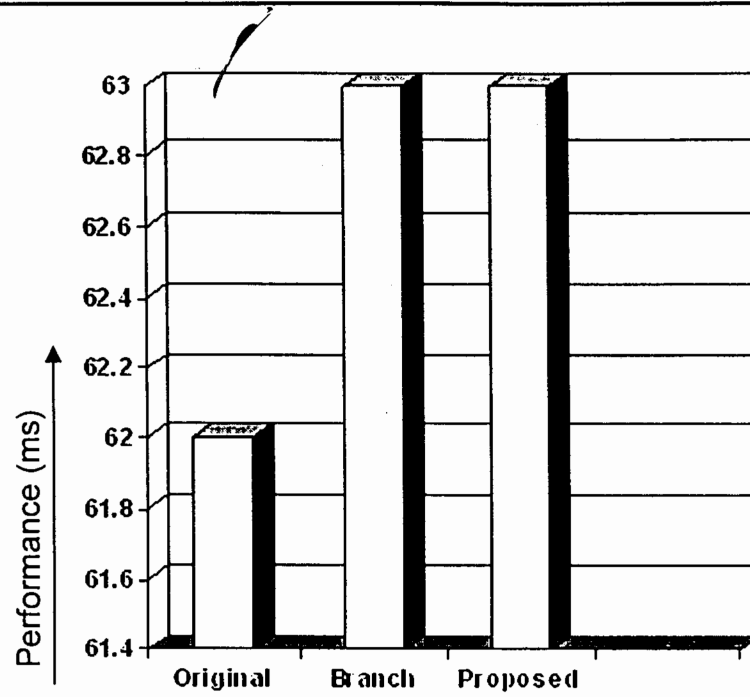


Figure 5.4 PERFORMANCE

Chapter 6

CONCLUSION AND FUTURE WORKS

6. Conclusions and Future Works

There are three major threats recognized against the intellectual property in software: malicious reverse engineering, software tampering, and software piracy [2, 7, 10, 12, 27, 36]. Researchers are striving to devise stronger protection mechanisms to defeat these malicious attacks. The foremost of software-based protection schemes so far developed are software watermarking, code obfuscation and tamper-resistance [7, 9, 10, 11, 13, 17, 32, 33]. In this research we have focused on addressing the threat of software piracy particularly for Microsoft .NET framework based applications. Software piracy has been compounded by several factors. Among those, rich distribution formats, such as provided by Java and .Net, make attacker's life much easier by exposing trade secrets which are coded inside an executable [19, 20, 21, 30, 34]. The availability of high speed internet and peer-to-peer systems also adds to the ease of distributing pirated software [3, 4, 5]. Statistical loses proves that piracy has become a prevalent problem of software industry and it requires the utmost attention to explore software protection schemes to better prevent and detect copyright violations [1, 3, 5, 12].

This research explores .NET framework constructs to identify its potential for developing software watermarking algorithms which particularly protects applications developed using .NET. We have devised a novel software protection scheme, called *software thumbprinting*. Proposed technique employs control flow transformations to encode unique thumbprints into every distributable copy of software. A *thumbprint* serves like a secret signature which proves the original ownership and legal buyership of a particular copy of .NET program. Software thumbprinting crafts non-reversible and semantic-preserving code transformations to tamper-proof the embedded thumbprints. In order to strengthen the theft detection and assertion mechanism, proposed technique is tied with licensed distribution.

Moreover presented work aims to emphasize the need of developing *Sand Mark* and *Loco* like tools, based on .NET framework constructs so that the automatic program analysis may rise on this platform. One of the greatest challenges in this area is the lack of evaluation of various software protection mechanisms. For that purpose, a thorough study

and evaluation of contemporary software protection techniques with a practical viewpoint is presented. Research aims to practice and promote research and development of better software protection mechanisms.

Future works in this direction could lead to develop a codec to automate the process of thumbprint encoding and decoding. For this purpose, our thumbprint and license generation tool Marka (discussed in Chapter 4) can be extended to perform codec's functionality. Technical details of issues regarding automation of proposed scheme are presented in Section 4.3 and 4.4. Like other software based protections, thumbprinting too aspires to make attacker's life harder but not to make copyright violation impossible. So there exists strong need of improving such protection mechanism and developing better ones. For example, thumbprinting can be improved by using some alternate of CIL *br* (break/jump) instructions to encode thumbprint so that to make the algorithm more robust, stealthy and performance effective. Another approach can be to adapt proposed model to devise protection solutions for other languages and platforms which offer similar programming constructs, like Java.

One of the most pressing issues, we found through exploring the copyright thefts and their respective defenses, is the detection of any infringement. There have been substantial studies and commercial efforts found so far to implement, prevent and assert copyright violations but all these efforts are mean less without an effective detection mechanism. No matter how complex algorithm is used to embed a secret signature, and even it does not matter that how stealthily it is encoded and how good program performs after encoding. All this is futile if we can not detect that some one has pirated our software. We see uncountable number of software companies who have compromised over the fact that software copyright violation is inexorable because we have no such strong detection mechanism through which we may get to know the piracy. Internet is the only medium that can serve some how to detect pirated software. Like, Microsoft has recently employed Windows Genuine Advantage (WGA) that enforces Window's online validation to detect pirated instances of Windows [45]. But this scheme works only if some illegal user will visit WGA online, other wise the theft can not be detected. Hence there remains a long way to go towards inventing better software protection mechanisms.

TERMINOLOGY

Software Protection: Protecting ownership rights which a vendor possess over his/her software product

Digital Rights Management: An umbrella term that refers to the technologies used by copyright owners to control access to and usage of their digital data

Software Piracy: The act of making and reselling illegal copies of software

Software Reverse Engineering: The act of extracting and reusing some parts of software program, with out permission of its copyright owner

Code Tampering: The act of modifying software program code illegally so as to make it behave unintended

Software Watermarking: Embedding some covert information into the software program that serves as a secret signature of copyright owner

Software Fingerprinting: Embedding unique watermarks into every distributable copy of a software product

Code Obfuscation: Transforming software code so as to make it obscure for an adversary while keeping its semantics intact

Tamper-proofing: Safeguarding software watermarks so that an attacker's attempt to distort watermark must cause the program malfunction

Software Thumbprinting: The process of encoding unique thumbprints as watermarks into each distributable instance of a program

REFRENCES & BIBLIOGRAPHY

[1] M. Shakeel Anjum, A. Hasan and K. Rashid. "*Thumbprinting .NET Applications*", IATED, Valencia Spain, 2007

[2] Wikipedia, http://en.wikipedia.org/wiki/software_piracy/

[3] BSA, http://www.bsa.org/, "*Software Management Guide*"

[4] SIIA, http://www.siia.net/piracy/whatis.asp. "*Anti-Piracy*"

[5] BSA, http://www.bsa.org/. "*Annual BSA and IDC Global Software Piracy Study*", 2006

[6] M. Antonio and P. Ernesto, "*An Efficient Software Protection Scheme*", Proceedings of the 16th international conference on Information security: Trusted information: the new decade challenge, pp. 385-401, 2001

[7] G. Naumovich and N. Memon. "*Preventing Piracy, Reverse Engineering, and Tampering*". *In IEEE Computer*, Vol 36, No. 7, pp. 64--71, July 2003

[8] M. Chen. "*Software Product Protection*". Article in T-110.501 Seminar on Network Security, ISBN 951-22-5807-2, 2001

[9] W. Zhu, C. Thomborson and F.Y. Wang. "*A Survey of Watermarking*". In LNCS vol 3495, Springer-Verlag,. pp. 454 - 458, April 2005

[10] C. Collberg and C. Thomborson. "*Watermarking, tamper proofing and obfuscation – tools for software protection*", In IEEE transaction on software engineering, vol 28, no 8, August 2002

[11] Ginger Myles. "*Using Software Watermarking to Discourage Piracy*", IEEE Security and Privacy, v.1 n.4, p.40-49, July 2003

[12] P. T. Devanbu and S. Stubblebine. "*Software Engineering for Security: A Road Map*", In Proceeding of ICSE Special Vol on the "Future of Software Engineering", 2000

[13] B. Fu, G. Richard III and Y. Chen. "*Some New Approaches for Preventing Software Tampering*", In ACM SE'06, Melbourne, Florida, USA. 2006

[14] C. Thomborson, J. Nagra, R. Somaraju and C. He. "*Tamper-proofing Software Watermarks*", In AISW, Conference in Research and Practice in Information Technology, Vol. 32, Dunedin, New Zealand, 2004

[15] B. Anckaert, B. D. Sutter, D. Chennet and K. D. Bosschere. *"Steganography for Executables and Code Transformation Signatures"*. In LNCS 3506, Springer-Verlag Berlen, pp. 431-445. 2005

[16] D. Curran, N. J. Hurley and M. O Cinneide. *"Securing Java through Software Watermarking"*. In Kilkenny 2nd International Conference on the Principles and Practice of Programming in Java, ACM, 2003

[17] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn and M. Stopp. *"Dynamic Path-Based Software Watermarking"*. In Proceeding of the conference on Programming Language Design and Implementation, pp.107-118, 2004

[18] eBook, Inside Microsoft .NET IL Assembler

[19] Dr. R. Wiener. *"Obfuscation and .Net"*. In Journal of Object Technology, Vol.4, No.4, pp.73-92, May-June-2005

[20] Pre-Emptive Solutions, http://www.preemptive.com/. *"Dotfuscator, Technical While Paper, version 2"*, 2004

[21] M. Madou, L.V. Put and K. D. Bosschere. LOCO, *"An Interactive Code (De)Obfuscation Tool"*. In Proceeding of ACM PEPM 1-59593-196-1/06 Charleston, South Carolina, USA. 2006

[22] J. Cox and J.M.G. Linnartz, *"Public Watermarks and Resistance to Tampering"*, Proceedings of the Fourth International Conference on Image Processing, Santa Barbara CA, October 1997

[23] C. Collberg and C. Thomborson, *"Software watermarking: Models and Dynamic Embeddings"*, In Principles of Programming Languages, San Antonio, TX, pp. 311-324, January 1999

[24] Venkatesan, V. Vazirani, and S. Sinha, *"A Graph Theoretic Approach to Software Watermarking"*, In 4th International Information Hiding Workshop, Pittsburgh, PA, April 2001

[25] H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, *"Detecting the Theft of Programs using Birthmarks"*, Information Science Technical Report NAIST-IS-TR2003014 ISSN 0919-9527, Graduate School of Information Science, Nara Institute of Science and Technology, Nov. 2003

[26] K. Gopalakrishnan East Carolina University Nasir Memon Polytechnic University Poorvi L. Vora Hewlett-Packard. *"Protocols for Watermark Verification"*. IEEE 2001

[27] M. Madou, B. Anckaert. B.D. Sutter and K.D. Bosschere. *"Hybrid Static-Dynamic Attacks against Software Protection Mechanism"*. In Proceeding of ACM DRM 1-59593-230-5, Alexandria. Virginia, USA,November 7, 2005

[28] T. Sander and C. F. Tshcudin. *"On Software Protection by Function Hiding"*. In 2nd International Workshop on Information Hiding, December 1998

[29] B. Anchaert, B. D. Sutter and K. D. Bosschere. *"Software Piracy Prevention through Diversity"*. In Proceedings of the 4th ACM workshop on Digital Rights Management, Washington

[30] A. Mishra, R. Kumar and P. P. Chakarabarti. *"A Method-based Whole-Program Watermarking Scheme for Java Class Files"*. 2005

[31] C. Collberg, C. Thomborson and D. Low. *"Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs"*. In Proceedings of ACM SIGPLAN-SIGACT pp.184—196, 1998

[32] K. Fukushima, T. Tabata and K. Sakurai. *"Evaluation of Obfuscation Scheme focusing on Calling Relationships of Fields and Methods in Methods"*. In Proceeding (440) Communication, Network and Information Security, 2003

[33] S. T. Chow, Y. Gu, H. J. Johnson and V. A. Zakharov, *"An Approach to the Obfuscation of Control-flow of Sequential Computer Programs"*, In G. 351. Davida and Y. Frankel, editors, ISC 2001, Lecture Notes in Computer Science 2200, pages 144-155. Springer-Verlag. 2001

[34] C. Collberg, C. Thomborson, and D. Low, *"A Taxonomy of Obfuscating Transformations"*, Technical Report 148, University of Auckland, 1997

[35] A Thesis for the Degree of Master of Science, *"Tamper Resistance for Software Protection"*. Ping Wang School of Engineering Information and Communications University 2005

[36] G. Hachez. *"A Comparative Study of Software Protection Tools Suited for E-Commerce with Contribution of Software Watermarking and Smart Cards"*. Phd thesis, University Catholique de Louvain 2003

[37] G. Qu aand M. Potonjak. "*Hiding Signatures in Graph Coloring Solutions in Information Hiding*". Pages 348-367, 1999

[38] Cullen Linn and Saurmay Debray Department of Computer Science. "*Obfuscation of Executable Code to Improve Resistance to Static Disassembly*". University of Arizons ACM 2003

[39] Ginger Myre Miles. "*Software Theft Detection through Program Identification*". PHD Thesis 2006

[40] M. Chen. "*Software Product Protection*". *Article in T-110.501 Seminar on Network Security, ISBN 951-22-5807-2*, 2001

[41] CrypKey , http://www.crypkey.com/instant.asp

[42] Salamander .NET Explorer, http://www.remotesoft.com/salamander/

[43] .NET Reflector, http://www.aisto.com/roeder/dotnet/

[44] .NET Reactor, http://www.eziriz.com/dotnet_reactor/

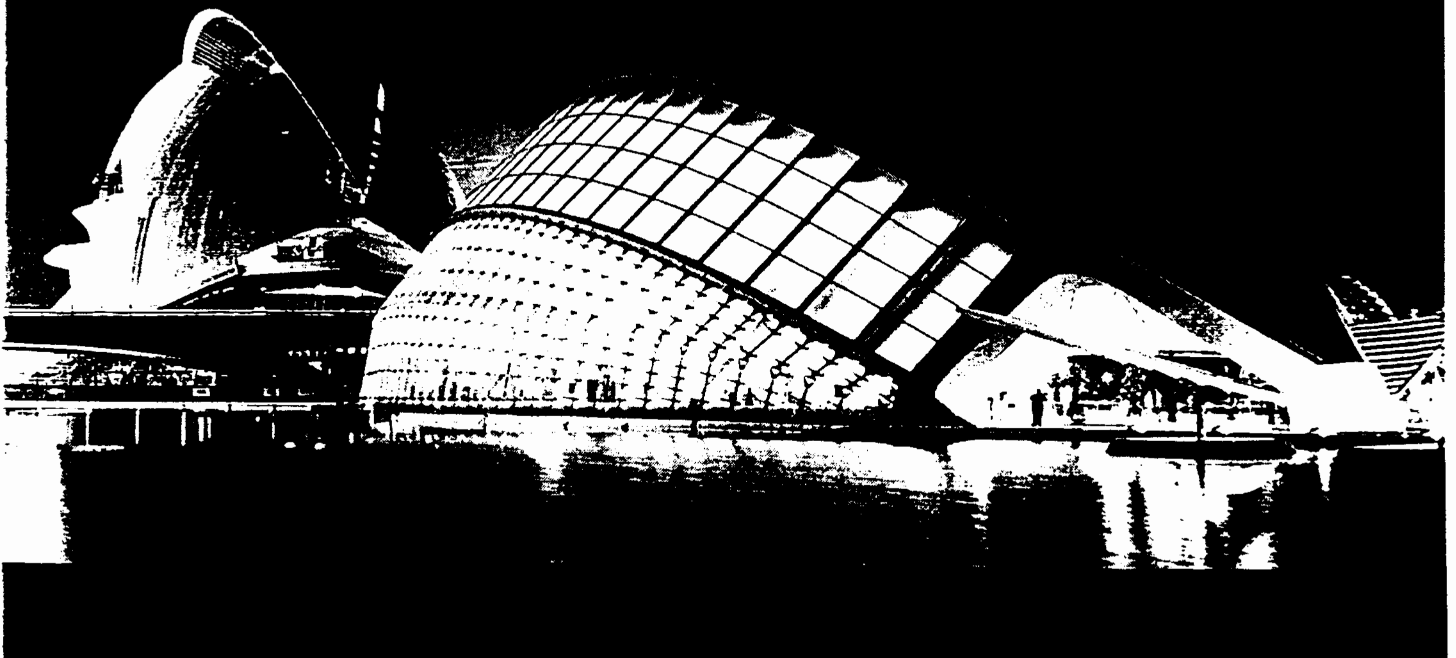
[45] Microsoft Windows Genuine Adventage (WGA), http://www.microsoft.com/presspass /press/2005/jul05/07-25WGA1PR.mspx

Appendix A

PUBLICATION

INTED 2007

International Technology, Education and Development Conference



INTED 2007

International Technology, Education and Development Conference

Proceedings

Published by
International Association of Technology, Education and Development (IATED)
C/ Dr. Vicente Zaragozá 70-9
46020 Valencia SPAIN
Web: www.iated.org

INTED2007 Proceedings
Edited by
L. Gómez Chova, D. Martí Belenguer, I. Candel Torres
International Association of Technology, Education and Development
IATED, Valencia

INTED2007 Proceedings (CD) ISBN: 978-84-611-4517-1
INTED2007 Abstracts Book ISBN: 978-84-611-4516-4

Book & CD covers designed by
J. L. Bernat Tomás

THUMBPRINTING .NET APPLICATIONS

M. SHAKEEL ANJUM
m_shakeel_anjum@yahoo.com

Faculty of Applied Science
International Islamic University
Islamabad, Pakistan

AHMED HASSAN
mrahmedhasan@gmail.com

Faculty of Applied Science
International Islamic University
Islamabad, Pakistan

PROF. DR. KHALID RASHID
dr_khalid@iiui.com

Dean Faculty of Applied Science
International Islamic University
Islamabad, Pakistan

ABSTRACT:

Extensive research has been conducted on software protection schemes like watermarking, obfuscation and tamper-proofing. These techniques aim to protect software copy rights by restricting piracy, illegal reverse engineering and tampering. Most of the technological implementation of watermarking algorithms is found for Java based applications. Where as, protecting .NET based applications is rarely discussed. This paper, by the first time, converses the subject of *software copy right protection* exclusively for *.NET framework based applications*. We have proposed a unique software watermarking technique called *thumbprinting* which in effect realizes software *fingerprinting*. The idea is to embed unique *thumbprints* as software watermarks, in forward and backward jump sequences of .NET program's MSIL code. The proposed scheme is demonstrated through an experiment to show how thumbprints are effectively tamper-proofed. The purpose of our research is to initiate the need of comprehensive watermarking solutions for .NET framework based applications.

Key Words: DRM, Watermarking, Fingerprinting, Thumbprinting, Steganography, Obfuscation, Tamper-proofing, MSIL, .NET.

1. INTRODUCTION

In recent years software piracy has emerged as a major problem for software vendors. Extensive research against [1, 2, 3, 4, 5] software piracy has proposed various protection schemes like watermarking, tamper-proofing, obfuscation, encryption and licensing. These techniques aim to protect software copy rights by restricting piracy, tampering and illegal reverse engineering. A common assumption among these schemes is that the software has been delivered to a malicious client, who aims to steal the source or semantics of intended application by applying certain semantic preserving attacks onto it. Not only he attempts to damage the original ownership signature but also tries to overwrite it with one of his own so as to make a false claim that the software belongs to him originally [6, 7, 8].

Technological countermeasures against such malicious attacks fall in two major categories, *hardware-based* and *software-based* protections. Hardware-based solutions include dongles, smart cards, re-writeable-media installers, dedicated processing devices and Trusted Computing Platform Alliance (TCPA). These technologies barricade software piracy by affixing application authentication mechanism with some fixated hardware device which contains embedded ownership signature. Alongside the fact, these are the most secure means, their fabricating cost and realization overheads confine their viability to large application domains [9].

Software-based protections, on the other hand, are extensively practicable. For this elemental reason they are focal point of our research. First mean in this regard is *watermarking* that is, embedding some covert information as a text message (static watermarking) or into observable behavior of the program (dynamic watermarking) [3, 9, 10, 11]. If this covert signature is set unique for each distributable copy of intended software then it becomes *fingerprinting* [12]. For our particular solution, we will call it *thumbprinting* that refers to the characteristic of our proposed scheme of embedding unique *thumbprints* into each copy of the software (as detailed in Section 4). A related subject is *Steganography* that means to embed a hidden message into an apparently innocuous executable through assembly level code transformations [1]. This is of our distinct interest because in our thumbprinting technique we go by steganographing the MSIL code of intended program. Second protection measure is *tamper-resistance* that means to safeguard the program code against tampering by crafting vigorous dependency among code structures [2, 3, and 9].

Tamper-proofing aspires either to cause application crash if tampered by some attacker or to make tampering expensive enough so that it become impractical for an attacker to remove all of the crafted dependencies [6]. Next software protection mean is code *obfuscation* that refers to the application of semantic preserving transformations on program constructs to make code obscure against reverse-engineering [4, 5, 13]. Fourth protection methodology is *encrypting* the executable which requires decryption of functional code before execution [7]. Last software-based solution is *licensed product distribution* that means to authenticate and authorize valid user to install and run a licensed copy [9].

In this paper we present our anticipated piracy prevention scheme for .NET framework based applications. Among the countermeasures discussed above, our solution is based on software watermarking and more specifically software thumbprinting. We will also describe, how we tamper-proof our encoded thumbprints by employing static code transformations.

In Section 3 we state software piracy problem. After redefining the problem and discussing the current state of art solutions, we propose our methodology in Section 4. Section 5 illustrates implementation details of thumbprinting through an example. Related literature survey is presented in Section 6. Section 7 discusses the problems in previous approaches. At last we conclude the paper in Section 8 and exploit future works.

2. MOTIVATION

Philosophically admitted that nothing is new since the Big Bang, what interacts is how things are combined. People make money by devising novel combination of existing solutions. And accordingly such inimitable solutions become targets of crooks to get benefited from them unlawfully. Software piracy is one of the similar acts that some malicious user does by tampering with software code so as to replace or wreck its original ownership or authorship mark [3, 9]. Pirates carry out copy right violations most of the times for monitory gains by reselling a software product illegitimately. Software piracy gets more severe figures when motive of a pirate is to disrupt the functioning of some corporation, government agencies or online service provider.

Software piracy has become a grave trouble for software vendors from last several years [14]. According to an annual global study on software piracy conducted by Business Software Alliance; 35% of installed software packages in year 2005 were pirated [14]. That amount of a penalty of $34 billions to the software vendors in one year, which is forecasted to grow up to $200 billion during same five years. The results confirmed that piracy will continue to be a significant problem. So, in this endeavor we intend to contribute towards the research against software copy rights violations.

Major emphasis of research in software protection is on developing such algorithms and techniques which prevent malicious hosts from distorting owner's copy right signatures. As per our findings, most of technological consideration and implementation of these algorithms is found for Assembly language's binary code and Java language's byte code [1, 2, 15, 16]. Where as, Microsoft Intermediate Language (MSIL or shortly IL) is rarely discussed. This paper particularly considers IL constructs for their support to devise generic solutions for the protection of .NET framework based applications against illegitimate adaptation, distribution and reproduction. Moreover we motive to present a fine blend of recent research proposals, current industry practices and our proposed solution with sufficient implementation details so as to make it enough practicable.

3. SOFTWARE PIRACY

Wikipedia defines software piracy as "*the unauthorized use of copyrighted material is a manner that violates one of the copyright owner's exclusive rights, such as the right to reproduce or perform the copyrighted works, or to make derivative works that build upon it*" [17]. Laws regarding copy right infringement state piracy as a crime that may cause penalty of up to $250,000 to the prosecute [18]. This highlights the need of solving the piracy problem from both vendor and customer's point of view.

3.1. Problem Redefined

We start by redefining the *software piracy problem* in its real context and then we will truss our focus on it through out rest of the paper while presenting our intended solution.

- Alice is the owner of P, for that she has embedded her ownership signature into P
- Alice intends to earn profit from P on its per-copy-sale basis as she has acquired legal copy rights of P for its reproduction, redistribution and

- Bob purchased a licensed copy of P from Alice
- Bob determines to pirate P to earn unlawful gains from its resale by distorting Alice's ownership signature
- Alice necessitated some prevention measure to be employed into P that must resist against Bob's malicious attack
- Having enough time and resources, Bob managed to wreck Alice's signature and overwritten it with one of his own
- Charles purchased a pirated copy of P from Bob.
- Charles shared this illegitimate copy of P with his colleague Dolly
- Alice hence required to detecting pirated copy of her software that Dolly was running.
- Moreover, Alice needs to detect the actual pirate Bob, so that to prosecute against him in the court room
- In the court of law Alice asserts that P originally belongs to her, for that she has to render ownership signature which she has embedded into P

3.2. State of the Art

The customary solution against software piracy lies among certain *ethical*, *technical* or *legal* means. Concluding from the above problem description we propose to divide our entire piracy prevention scheme into four major divisions; i.e. *piracy education, piracy prevention, piracy detection* and *piracy assertion*, as shown in Figure 1. These subdivisions at the same time help sorting out their respective anti-piracy measures.

Piracy Education: Software vendor ethically means to edify users about benefits of intellectual property protection and its contribution towards worldwide economic growth. Organizations like BSA are in force in this arena with partnership of leading software companies like Microsoft, Apple Mac., RSA, MacAfee, etc. to promote safe and legal digital world through education and enforcement of digital copy rights [14].

Piracy Prevention: It refers to the technical defensive measures which software providers implant into their products so as to prevent piracy at first place. It is of sure that copying digital artifacts can not be restricted, so these preventive measures cause illegitimate copies to malfunction or stop running at all or cause enough cost overhead to pirate so that it may exceed actual development cost of software [9].

Piracy Detection: If in any ways some pirate successfully wrecks preventive measures and start profiting from it through illegal redistribution or personal usage, then copy right owner needs to detect such copy right violations in order to take some legal actions.

Piracy Assertion: Finally when some software vendor fortunately detects any illicit copy of his software, he is required to prove his ownership in the court room.

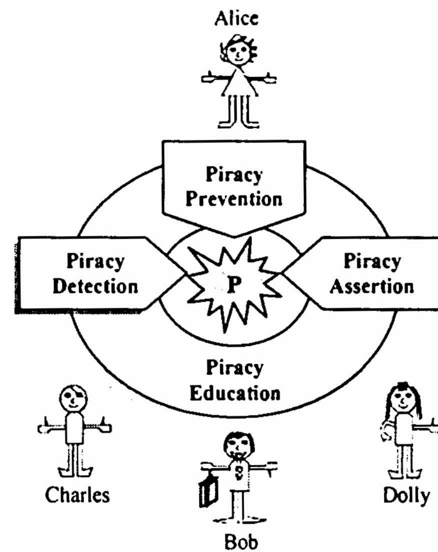


Figure 1: Means of Piracy Prevention

Resolutely speaking, technical and legal means are in strong conjunction with each other, dejecting one, fades other's effectiveness. One of the major reasons why piracy prevention has not yet been achieved successfully is that most of the time vendors (if do) typically employ mere technical protection means without decisively taking up reactive legal strokes against their copy rights infringements. Therefore our *anti-piracy defense suit* presented in next section, embraces all piracy prevention means like prevention, detection and assertion.

3.3. Anti-Piracy Lines of Defense

In order to realize software protection measures discussed above, we present a technological solution that comprises of currently available lines of defense against software piracy. Proposed anti-piracy defense suit is inherently in practice for .NET framework based applications [4] but here we intend to align it in more explicable and rational order, as Figure-2 illustrates it.

Alice starts by developing the inner most layer, the *software* P and keeps on wrapping protection layers around it till its *distribution.* First she embeds her secret ownership signature into P as a *watermark,* which she will afterwards render in the court of law to assert her copy rights. Then she *tamper proofs* her watermark so that P must become useless for Bob, or uneconomical to tamper with P, if he determines to distort Alice's watermark. After that she *obfuscate*s that tamper-proofed executable to make P's source code obscure enough against nasty reverse engineering. She subsequently *encrypt* P to make its code indecipherable. Finally she distributes *licensed* copies of her software to respective *licensed* users. Alice may choose to employ any or all of these lines of defense according to the required level of protection.

On the other hand, Bob needs to defeat all of the drawn protection layers, starting from the outer most one, the *licensing* mechanism. He keeps on infringing by *decrypting* the executable and *deobfuscating* program code. Afterward he *tampers* with the source code in order to get a fully functional instance after successfully distorting the *watermark.*

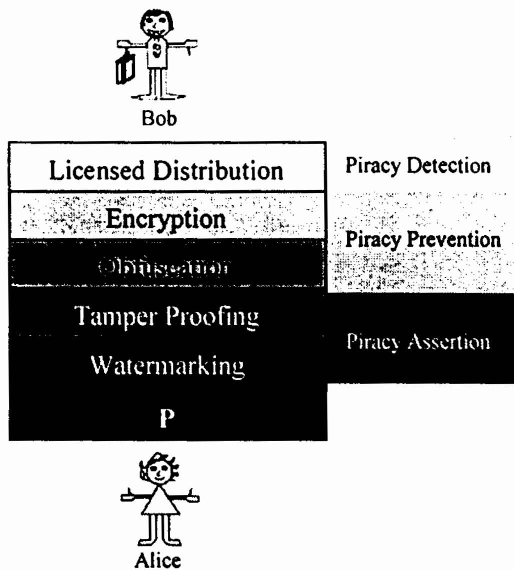


Figure 2: Anti-Piracy Lines of Defense

4. PROPOSED METHODOLOGY

We have gone through different possible attacks and their effects on intellectual property rights of software. Bob may use them to harm Alice's ownership mark. There are four possible ways through which bob can tamper the program P.

1. Bob can neither crack ownership mark nor can he use the program P.

2. Bob may distort the watermark but he can not use the program P.

3. Bob may use program P but can not distort the watermark or ownership.

4. Bob distort the watermark or ownership and also uses the application.

From Alice's point of concern. 1st is the best case and 4 is the worst one where as 2nd and 3rd are equally acceptable. So as Bob should be left with any one of the two choices. either to have a functional product or to live with Alice's ownership. Researchers tend to agree on the fact that it is almost impractical to prevent Bob in a single turn, from doing both, distorting watermark and using P (case 1). Also it is evenly unacceptable to allow him to do both (case 4). Therefore our solution considers case 2 and 3 in intention. Next we will illustrate both thumbprint encoding and decoding one by one.

4.1. Thumbprint Encoding

Once the target product P is developed and its source code is compiled into P.exe, then Alice needs to generate unique thumbprints to be embedded into each of its distributable instances. These thumbprints are distinct numbers produced from thumbnails of license information. Each thumbprint exclusively represents one and only one copy of P with its original licensee's information.

This lic.txt contains a thumbnail which is a fixed length stream of bytes generated from encrypted thumbnail [A]. The selection of encryption scheme is on the choice of vendor and additionally if supported by the technology. In our case, .NET framework provides us with a built-in protection API that includes wide range of popular encryption and hashing schemes like, MD, RSA, DES, SHA etc. The basic purpose of encrypting thumbnail is to obscure the license information from attacker so that s/he would not be able to revive any of the information of original licensee and licensed product. After having encrypted thumbnail [B] we used .NET integral string hash code generation to have a unique number [C]. Hash codes found this way are usually 9-12 digits long and are both positive and negative numbers. So we devised a simple logic to produce a smaller number (around

100) as thumbprint so that it would be easy to be encoded into the program code.

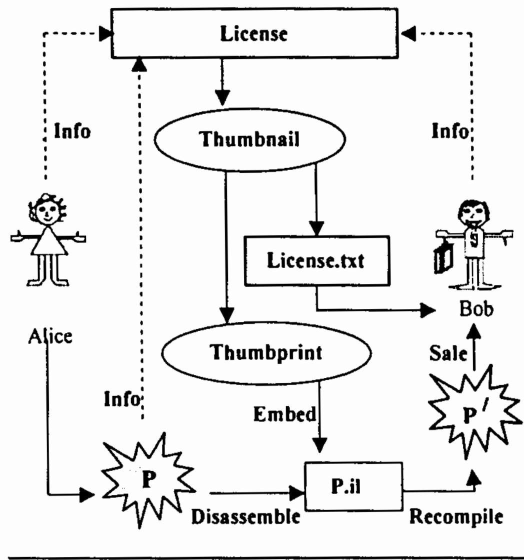


Figure 3: Generating & Embedding Thumbprint into P

The idea was to sum up the digits of a hash code [D]. There is a uniqueness constant (uc) introduced in case of replication of thumbprint. The addition of this constant in thumbprint has no more purpose other than just to make the thumbprint distinctive. This thumbprint is to be inserted as a binary number [E]. Next thing is to calculate the data rate N. That is the number of times (number of methods in which) thumbprint will be embedded into the program [F]. To make our scheme more robust and tamper-resistant we choose to 2N+1 of total number of methods of a program in which thumbprint will be encoded.

Finally $\mathcal{E}$ is our thumbprint encoder. It encodes specified thumbprint into p by N number of times [G]. This information is also then recorded into the DB so that to assert the ownership and to identify the original product licensee [H].

- licInfo = Encrypt (Vendor + Product + User + Copy Sr #) [A]
- thumbnail = computeHash (licInfo) [B]
- hash = hashCode(thumbnail) [C]
- thumbprint = Σ (hash) [+uc] [D]
- thumbprint = $(\text{thumbprint})_2$ [E]
- N = $\{n \wedge 2n+1 = < M(P)\}$ [F]
- $P' = \mathcal{E}$ (P, thumbprint, N) [G]
- DB = DB + (licInfo + thumbprint + N) [H]

4.2. Thumbprint Decoding

To extract a thumbprint Alice starts with Bob's copy of P, i.e. P'. Fist of all Alice will query DB for specific Product to get the value N, i.e. the minimum occurrences of thumbprint required to prove her ownership [I]. Our decoder đ will then extract the thumbprint that was embedded by 2N+1 times in P' [J]. Alice will subsequently requery DB with this thumbprint for original licensee's information [K]. On the basis of this information Alice may determine if Bob is the original licensee or is a pirate. Furthermore, if Bob claims to be the valid licensee of P' then he must be having a lic.txt. Thumbprint extracted from lic.txt by repeating [A, B, C & D] onto it, must be same as the thumbprint extracted from P'. This whole process of decoding thumbprint is shown in figure 4.

- N = DB (Product) [I]
- thumbprint = đ (P', N) [J]
- licInfo = DB (thumbprint) [K]

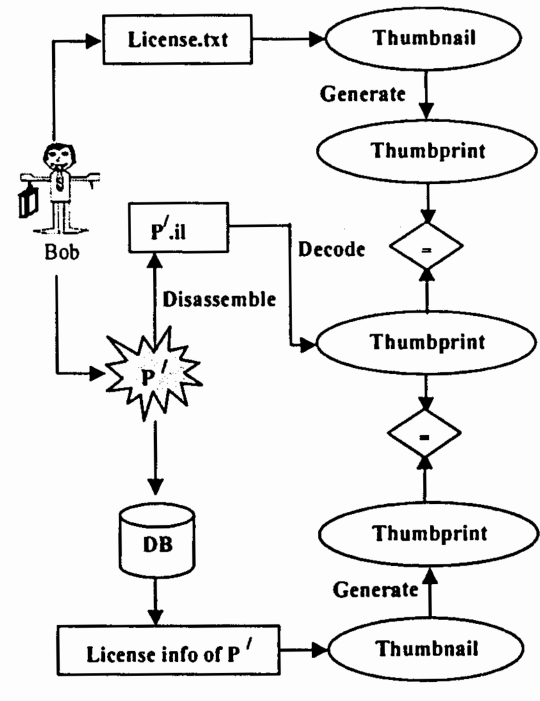


Figure 4: Extraction of Thumbprint from P

5. Thumbprinting .NET IL Code

In this section we present a prototypical implementation of our novel watermarking technique for .NET, called *thumbprinting*. The idea behind is to embed a unique number as a watermark in forward and backward jumps

sequence of program's IL code instructions. The reason to use IL code is that, its flattened structure makes static program analysis difficult to be understood. Also as per our exploration, there isn't any debugger that runs IL code for dynamic program analysis. Additionally it provides much more flexibility to embed thumbprint as employed by our proposed scheme. Collberg has proposed a similar technique to embed bits in forward and backward execution flow of program in [16, 8]. Our approach differs by the fact that we are not using any branch function instead we are encoding bits at direct jump positions. Removing branch function increases the obscurity of thumbprint.

We have developed an application in VB.NET that automates the process of thumbprint generation. This application asks vendor to input user specific license information and produces two outputs. First one is a license file (lic.txt) containing the license information and secondly its thumbprint (a positive whole number).

5.1. Experiment

Here we take our experiment through the steps defined in Section 4.2 to encode and decode thumbprint. For this purpose we have selected the following data set.

- sr = 1 *// Copy Serial Number*
- licInfo = Encrypt (Alice, P, Bob, 1)
  = "OalBlp1CbE"
- thumbnail = computeHash ("OalBlp1CbE")
  = "!@r#$ASyDF09u87p" *// lic.txt*
- hash = hashCode ("!@r#$ASyDF09u87p")
  = 2011013001
- thumbprint = Σ (2+0+1+1+0+1+3+0+0+1)
  = 9 [+ uc = 0] = 9 = 1001
- m = M (P) = 30 *// No. of Methods in P*
- N = 10 *// any No for which 2N + 1 < m*
- DB = DB + (Alice, P, Bob, 1, 9, 10)

5.2. Implementation

The code below is the disassembly of program Hello.exe. For the sake of better understanding we have taken a single method **abc()**. The IL code is simplified to present our code transformations to encode thumbprint's bit stream. Such disassembly can be produced by *ildasm.exe* that is shipped with VS.NET.

```
method void abc ( ) {
    IL_0001 : nop
    IL_0007 : ldstr     "Hello World"
    IL_0003 : call  void [mscorlib.printstr()]
    IL_0005 : nop
    IL_0009 : ret
}
```

Figure 5: P = disassembly of method abc ( )

In its natural flow .NET compiler executes instructions in downs ward direction. So we fabricated a binary sequence (thumbprint) by introducing new forward and backward jumps (MSIL break instruction, i.e. br). Our scheme preserves semantics of the program even after ε runs P through transformations τ to produce the thumbprinted P′. Thumbprinting IL code this way is tamper-proofed because any change in code at IL level or at high level with disrupt the ultimate IL statements' labels and that will cause jumps to hit invalid targets. And the number of times we embed thumbprint is quite high (like in 21 (2N+1) out of 30 methods) so for an attacker it is quite intricate to sort out the hidden thumbprint. Presuming if some attacker may decipher any of encoded thumbprint, yet it is quit difficult to figure out all of its 21 instances or at minimum N instances. Furthermore even after revealing max number of its occurrences, it is hectic to remove them all while keeping semantics intact.

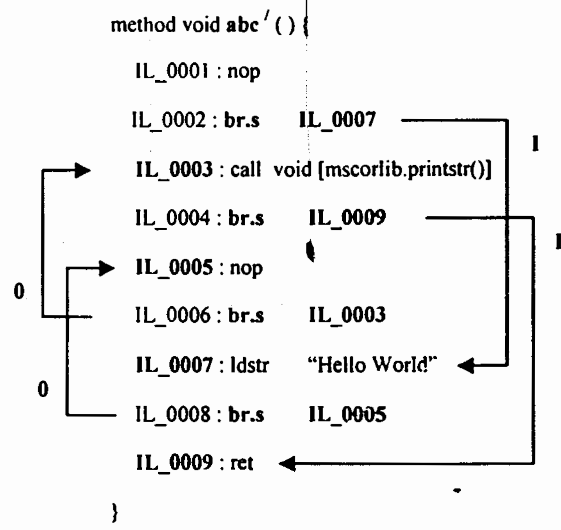


Figure 6: P′ = τ (P)

6. RELATED WORKS

Software protection is a continuously war between two forces, one is defender who wants to secure his product or intellectual property rights and other is attacker who will always look for loopholes to break the product. In this paper we have presented a combination of software watermarking and temper proofing. The previous approaches have mostly disused watermarking java programs. Many papers contributed towards the definition of software protection as a problem. Gleb and Nasir in [9] describe each term i-e associated with software protection very precisely and clearly. They discussed that software protection is not limited to watermarking but it includes securing the software through licensing files, application servers, hardware based-solutions, temper proofing and obfuscations. You need complete software protection framework in order to fully protect your software. Collberg in [3] discusses the watermarking, tamper-proofing and obfuscation as tools for software protection. These tools are effective against three types of attacks by any malicious hosts. Collberg also stress that we can protect our software if we take good care of all attacks i-e software piracy, reverse engineering and tamper-proofing. We need to apply all tolls together in order to completely secure our software. Clark Thomborson [11] especially focuses on the software watermarking as a mean of protection again software piracy. He describes the four attacks on software watermark. Clark has also discussed the different algorithms proposed by other people for software watermarking. We have used these algorithms in our work. Another technique proposed by the [8] statistically changes the addresses of caller and calling function. These addresses were dynamically manipulated according to branch-function based watermarking introduced by Collberg [16]. This technique adds a branch function which when gets controls; dynamically generate return address so the control is never returned to caller functions from this dummy branch function. Thomas sander [7] discussed the method to hide the function means all functionality is hidden from the user which reduce the risk of reverse engineering. Encrypted function is executed with out decryption that eliminates the program analysis. In [19] Min Chen presents a through study of legal & technical means to protect the software. In legal means it discusses the copy rights, patents and licenses. In technical means it only covers software distribution model. Richard wiener in JOT [20] discusses the obfuscation process adopted by two tools. We have used these tools in order to do obfuscation in our process. Software piracy problem can also be solved through diversity as [21] describes a mechanism of diversity through uniqueness of installation, customized updates. In this scenario customer will be connected frequently to the software vendor. Formal cost of proposed sachem is also presented.

Now we will see how different researchers have solved the software protection problem for java. Java is mostly used for development so software developed using this language needs much attention. Java classes are mostly protected and watermarked. Mishra [22] proposed a method of static watermarking for whole program. Watermark is inserted by extracting control flow diagram of method and by assigning it a certain value based on pre-defined scheme. Watermark of the same value is constructed and inserted into that method. If N method required to be protected then at least 2N+1 method need to be watermark. Another good paper [15] by Curran has proposed a new technique called method-depth technique. Proposed scheme embeds a value of watermark on the basis of call graph depth of a method. Scheme is mainly based on recursion. One of similar approach as ours is presented by Collberg [16] embeds the watermark in the dynamic branch structure of program. They have used temper proofing techniques to prove that their watermark is resilient.

We have also gone through different techniques of temper proofing and obfuscation because if watermark is not temper proof properly it can easily be detected and removed. It also helped us to evaluate different tools available for obfuscation. Clark in [6] presented a technique based on constant encoding. He replaced the numeric value with function call; this function computes back this value through watermarked dynamic data structure of actual program. Proposed scheme constructs a graph (PPCT) by mapping WM numeric value code through 'codec', i-e encoder and decode function. Collberg in [13] presents a several obfuscation techniques for java. They present opaque constructs and introduce them in the program where there is chance of condition evolution. They have also used basic constructs like if-else with some ADT's like tree, graph or even threads to introduce opaque predicates. Another obfuscation scheme [23] presented by fukushima is based on destructing encapsulation structure of classes by transferring one class's local method's member, variables & instructions to another class as static members. This creates high dependency among classes, thus complicate

reverse engineering by causing temper resistance. Steganography is another method to embed your watermark as a secret message Bertrand Anchaert [1] discussed this approach and argues that Steganography is different from watermarking because for watermarking we need to have extraction mechanism which is not required in case of Steganography. Author has described four key areas where we can hide our secret message. These are instructions selections, register allocations, instruction scheduling and code layout. Author also developed a tool to exract and embeds a secret message.

Two tools for obfuscations are available in the market. These are Dotfuscator and LOCO. Dotfuscator [4] removes debug information and non-essential metadata from a MSIL file as it process it. This tools work on complied MSIL code, not source code. The main thing is that obfuscated MSIL is functionally equivalent to traditional MSIL code and will execute on Common Language Runtime (CLR) with identical results. This tool supports both incremental and control flow obfuscation. Another relatively new tool LOCO [24] employs control flow flattering branch function & opaque predicates as basic transformation on java code.

7. DISCUSSIONS

We have gone through many approaches for software protection. The common problem with all approaches is that, they all discuss the software protection for Java. Only Dotfuscator tool has discussed obfuscation for .net based application. Where as market is currently captured by .net applications. Now it is requirement of the time that we should have some solution for .net based application too. Technique used by [8] adds one or more branch functions so it can be visible and demark-able among original code. The functionality of this branch function is also not clearly defined. Thomas sander [7] has introduced software as a usage concept which can have some architectural problems. He also used an encrypted function but what this encrypted function contains either some function calls or some patent information. Most of the product protection means discussed by the Min Chen [19] lack tamper-proofing verification code, so they can be defeated any ways. Another important aspect is that most of the solution are from third party and needs to be embedded into to the application which creates dependency. The idea proposed by [21] is unique one but it lacks the practicality. It is also costly to have an automated updating system for every product. There is no justification of details that when software require to be updated.

8. CONCLUSION AND FUTURE WORKS

Software copy right protection is the utmost need of time against software piracy. The purpose of our research is to identify the need for developing new software watermarking algorithms, especially to protect .NET framework based applications. We aimed to expose the need to develop Sand Mark and Loco like tools, based on .NET framework constructs. Our research aimed to practice and promote research and development of better software protection mechanisms for .NET applications.

We have tried to present a through study and evaluation of contemporary software protection techniques with a practical viewpoint. Our proposed methodology is more resilient and stealthy then the previous ones because we have used hybrid watermarking. It is also easy for vendor to decode because the thumbprint is closely coupled with license information and we can also compare the detected thumbprint with our DB so verify the results.

Future work in this direction can be to improve and automate the procedure presented above and to come up with more resilient and stealth method with less cost impact.

9. REFERENCES

[1] B. Anckaert, B. D. Sutter, D. Chennet and K. D. Bosschere. Steganography for Executables and Code Transformation Signatures, *In LNCS 3506, Springer-Verlag Berlen, pp. 431-445.* 2005

[2] B. Fu, G. Richard III and Y. Chen. Some New Approaches for Preventing Software Tampering. *In ACM SE'06, Melbourne, Florida, USA.* 2006.

[3] C. Collberg and C. Thomborson. Watermarking, Tamper-proofing, and Obfuscation-Tools for Software Protection. *In IEEE transaction on software engineering,* vol 28, no 8, August 2002

[4] Pre-Emptive Solutions www.preemptive.com. Dotfuscator, *Technical While Paper, version 2,* 2004.

[5] S. K. Udupa, S. K. Debray and M. Madou. De-obfuscation Reverse Engineering Obfuscated Code. *In 12th Working Conference on Reverse Engineering, IEEE Computer Society,* November 2005.

[6] C. Thomborson, J. Nagra, R. Somaraju and C. He. Tamper-proofing Software Watermarks. *In AISW,*

Conference in Research and Practice in Information Technology, Vol. 32, Dunedin, New Zealand, 2004

[7] T. Sander and C. F. Tshcudin. On Software Protection by Function Hiding. *In 2nd International Workshop on Information Hiding*, December 1998

[8] M. Madou, B. Anckaert, B.D. Sutter and K.D. Bosschere. Hybrid Static-Dynamic Attacks against Software Protection Mechanism. *In Proceeding of ACM DRM 1-59593-230-5, Alexandria, Virginia, USA,* November 7, 2005.

[9] G. Naumovich and N. Memon. Preventing Piracy, Reverse Engineering, and Tampering. *In IEEE Computer*, Vol 36, No. 7, pp. 64--71, July 2003

[10] P. T. Devanbu and S. Stubblebine. Software Engineering for Security; a Road-Map. *In Proceeding of ICSE Special Vol on the "Future of Software Engineering"*. 2000.

[11] W. Zhu, C. Thomborson and F.Y. Wang. A Survey of Watermarking. *In LNCS vol 3495, Springer-Verlag,. pp. 454 - 458*, April 2005

[12] D. Bendersky, A. Futoransky, L. Notarfrancesco, C. Sarraute and A. Waissbein. Advance Software Protection Now. In Corelabs Technical Report available at *http://www.coresecurity.com/corelabs/projects/software protection.php*. 2003.

[13] C. Collberg, C. Thomborson and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. *In Proceedings of ACM SIGPLAN-SIGACT pp.184—196*, 1998.

[14] BSA, www.bsa.org. Annual BSA and IDC Global Software Piracy Study, 2006.

[15] D. Curran, N. J. Hurley and M. O Cinneide. Securing Java Through Software Watermarking. *In Kilkenny 2nd International Conference on the Principles and Practice of Programming in Java, ACM*, 2003

[16] C. Collberg, E. Carter, S. Debray, A. Huntwork, C. Linn and M. Stopp. Dynamic Path-Based Software Watermarking. *In Proceeding of the conference on Programming Language Design and Implementation, pp.107-118*, 2004.

[17] en.wikipedia.org/wiki/software_piracy.

[18] UGS Corp. Guide To Software Piracy Prevention. Jan 2005.

[19] M. Chen. Software Product Protection. *Article in T-110.501 Seminar on Network Security, ISBN 951-22-5807-2*, 2001

[20] Dr. R. Wiener. Obfuscation and .Net. *In Journal of Object Technology, Vol.4, No.4, pp.73-92*, May-June-2005.

[21] B. Anchaert, B. D. Sutter and K. D. Bosschere. Software Piracy Prevention through Diversity. *In Proceedings of the 4th ACM workshop on Digital Rights Management, Washington DC, USA*, November 2004

[22] A. Mishra, R. Kumar and P. P. Chakarabarti. A Method-based Whole-Program Watermarking Scheme for Java Class Files. 2005

[23] K. Fukushima, T. Tabata and K. Sakurai. Evaluation of Obfuscation Scheme focusing on Calling Relationships of Fields and Methods in Methods. *In Proceeding (440) Communication, Network, and Information Security*, 2003.

[24] M. Madou, L.V. Put and K. D. Bosschere. LOCO, An Interactive Code (De)Obfuscation Tool. *In Proceeding of ACM PEPM 1-59593-196-1/06 Charleston, South Carolina, USA*. 2006.

Appendix B

CODE LISTING

B. Code Listing

B.1 Tower of Hanoi

The goal of this classic problem is to get the stack of discs from one post to another. Basically, there are three posts, one of which contains a stack of discs that get smaller as you go up, as shown in figure below:

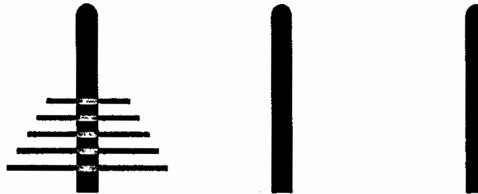


Figure B.1 Tower of Hanoi

Followings are the rules governing how discs can be moved:

1. only one disc can be moved at a time, and
2. never place a larger disc on top of a smaller one

B.1.1 High-level Language Code of Hanoi

The code presented below is VB.NET version of class *Hanoi*. This class is doing the major functionality regarding moving the discs from one tower to another hence solving the problem. Rests of the classes in the application are system classes which do not play any functionality but are created and used by .NET runtime environment.

```
1     ''' VB.NET version of class 'Hanoi'
2     Public Class Hanoi
3
4         Private discCounts(2) As Integer
5         Private moves(,) As Integer
6         Private movesIndex As Integer
7
8         ''' <summary>
9         ''' Constructor sets up the required local variables
10        ''' </summary>
```

```
11        Public Sub New(ByVal numDiscs As Integer)
12
13            discCounts(0) = numDiscs
14            discCounts(1) = 0
15            discCounts(2) = 0
16
17            ReDim moves(2 ^ numDiscs - 2, 1)
18            movesIndex = 0
19        End Sub
20
21        ''' <summary>
22        ''' GetMoves is the public method that kicks off the
23        ''' recursive process.
24        ''' </summary>
25        Public Function GetMoves() As Integer(,)
26            DoMoves(0, 1, 2, discCounts(0))
27
28            Return moves
29        End Function
30
31        ''' <summary>
32        ''' Recursive Method to get all disc moves
33        ''' to solve the Towers of Hanoi problem.
34        ''' </summary>
35        Private Sub DoMoves(ByVal sourceSpindle As Integer, _
36                            ByVal destSpindle As Integer, _
37                            ByVal tempSpindle As Integer, _
38                            ByVal discsToMove As Integer)
39
40            ' A recursive method needs a case where
41            ' to simply return back (the base case),
42            ' and one or more cases
43            ' where to recursively call the method
44            ' again (the recursive cases). In this
45            ' situation, if there are discs to
46            ' be moved, then it does some work,
47            ' otherwise it simply do nothing.
48            If discsToMove > 0 Then
49                ' This next line is a recursive call
50                ' that moves n-1 discs from the source
51                ' spindle to the temp spindle.
52                DoMoves(sourceSpindle, tempSpindle, _
53                        destSpindle, discsToMove - 1)
54
55                ' These next two lines move the last
56                ' disc from the source spindle to the n spindle.
57                discCounts(sourceSpindle) -= 1
58                discCounts(destSpindle) += 1
59
60                ' The next three lines record this
61                ' move in the moves array for use later.
62                moves(movesIndex, 0) = sourceSpindle
63                moves(movesIndex, 1) = destSpindle
64                movesIndex += 1
65
66                ' This recursive call moves the n-1
67                ' discs back from the temp spindle to the
```

```
68              ' destination spindle.
69              DoMoves(tempSpindle, destSpindle, _
70                      sourceSpindle, discsToMove - 1)
71          End If
72      End Sub
73  End Class
```

B.1.2 IL Code of Hanoi

The IL code presented below is obtained from disassembly of *Hano*i class.

```
1     ''' IL version of class 'Hanoi'
2       .class public auto ansi Hanoi
3            extends [mscorlib]System.Object
4       {
5         .field private int32[] discCounts
6         .field private int32[0...,0...] moves
7         .field private int32 movesIndex
8
9         .method public specialname rtspecialname instance void
      .ctor(int32 numDiscs) cil managed
10        {
11          .maxstack 3
12
13          IL_0000:  nop
14          IL_0001:  ldarg.0
15          IL_0002:  call       instance void
      [mscorlib]System.Object::.ctor()
16          IL_0007:  nop
17          IL_0008:  ldarg.0
18          IL_0009:  ldc.i4.3
19          IL_000a:  newarr     [mscorlib]System.Int32
20          IL_000f:  stfld      int32[] Hanoi.Hanoi::discCounts
21          IL_0014:  ldarg.0
22          IL_0015:  ldfld      int32[] Hanoi.Hanoi::discCounts
23          IL_001a:  ldc.i4.0
24          IL_001b:  ldarg.1
25          IL_001c:  stelem.i4
26          IL_001d:  ldarg.0
27          IL_001e:  ldfld      int32[] Hanoi.Hanoi::discCounts
28          IL_0023:  ldc.i4.1
29          IL_0024:  ldc.i4.0
30          IL_0025:  stelem.i4
31          IL_0026:  ldarg.0
32          IL_0027:  ldfld      int32[] Hanoi.Hanoi::discCounts
33          IL_002c:  ldc.i4.2
34          IL_002d:  ldc.i4.0
35          IL_002e:  stelem.i4
36          IL_002f:  ldarg.0
37          IL_0030:  ldc.r8     2.0
38          IL_0039:  ldarg.1
39          IL_003a:  conv.r8
40          IL_003b:  call       float64
```

```
     [mscorlib]System.Math::Pow(float64, float64)
41         IL_0040:  ldc.r8     2.0
42         IL_0049:  sub
43         IL_004a:  call       float64
     [mscorlib]System.Math::Round(float64)
44         IL_004f:  conv.ovf.i4
45         IL_0050:  ldc.i4.1
46         IL_0051:  add.ovf
47         IL_0052:  ldc.i4.2
48         IL_0053:  newobj     instance void
     int32[0...,0...]::.ctor(int32, int32)
49         IL_0058:  stfld      int32[0...,0...] Hanoi.Hanoi::moves
50         IL_005d:  ldarg.0
51         IL_005e:  ldc.i4.0
52         IL_005f:  stfld      int32 Hanoi.Hanoi::movesIndex
53         IL_0064:  nop
54         IL_0065:  ret
55       }
56
57       .method public instance int32[0...,0...] GetMoves() cil
     managed
58       {
59         .maxstack 6
60         .locals init (int32[0...,0...])
61
62         IL_0000:  nop
63         IL_0001:  ldarg.0
64         IL_0002:  ldc.i4.0
65         IL_0003:  ldc.i4.1
66         IL_0004:  ldc.i4.2
67         IL_0005:  ldarg.0
68         IL_0006:  ldfld      int32[] Hanoi.Hanoi::discCounts
69         IL_000b:  ldc.i4.0
70         IL_000c:  ldelem.i4
71         IL_000d:  callvirt   instance void
     Hanoi.Hanoi::DoMoves(int32, int32, int32, int32)
72         IL_0012:  nop
73         IL_0013:  ldarg.0
74         IL_0014:  ldfld      int32[0...,0...] Hanoi.Hanoi::moves
75         IL_0019:  stloc.0
76         IL_001a:  br.s       IL_001c
77         IL_001c:  ldloc.0
78         IL_001d:  ret
79       }
80
81       .method private instance void DoMoves(int32 sourceSpindle,
     int32 destSpindle, int32 tempSpindle, int32 discsToMove) cil
     managed
82       {
83         .maxstack 6
84         .locals init (int32,bool)
85
86         IL_0000:  nop
87         IL_0001:  ldarg.s    4
88         IL_0003:  ldc.i4.0
89         IL_0004:  cgt
90         IL_0006:  stloc.1
```

```
91          IL_0007:  ldloc.1
92          IL_0008:  brfalse.s  IL_0082
93          IL_000a:  ldarg.0
94          IL_000b:  ldarg.1
95          IL_000c:  ldarg.3
96          IL_000d:  ldarg.2
97          IL_000e:  ldarg.s    4
98          IL_0010:  ldc.i4.1
99          IL_0011:  sub.ovf
100         IL_0012:  callvirt   instance void
      Hanoi.Hanoi::DoMoves(int32, int32, int32, int32)
101         IL_0017:  nop
102         IL_0018:  ldarg.0
103         IL_0019:  ldfld      int32[] Hanoi.Hanoi::discCounts
104         IL_001e:  ldarg.1
105         IL_001f:  stloc.0
106         IL_0020:  ldloc.0
107         IL_0021:  ldarg.0
108         IL_0022:  ldfld      int32[] Hanoi.Hanoi::discCounts
109         IL_0027:  ldloc.0
110         IL_0028:  ldelem.i4
111         IL_0029:  ldc.i4.1
112         IL_002a:  sub.ovf
113         IL_002b:  stelem.i4
114         IL_002c:  ldarg.0
115         IL_002d:  ldfld      int32[] Hanoi.Hanoi::discCounts
116         IL_0032:  ldarg.2
117         IL_0033:  stloc.0
118         IL_0034:  ldloc.0
119         IL_0035:  ldarg.0
120         IL_0036:  ldfld      int32[] Hanoi.Hanoi::discCounts
121         IL_003b:  ldloc.0
122         IL_003c:  ldelem.i4
123         IL_003d:  ldc.i4.1
124         IL_003e:  add.ovf
125         IL_003f:  stelem.i4
126         IL_0040:  ldarg.0
127         IL_0041:  ldfld      int32[0...,0...] Hanoi.Hanoi::moves
128         IL_0046:  ldarg.0
129         IL_0047:  ldfld      int32 Hanoi.Hanoi::movesIndex
130         IL_004c:  ldc.i4.0
131         IL_004d:  ldarg.1
132         IL_004e:  callvirt   instance void
      int32[0...,0...]::Set(int32, int32, int32)
133         IL_0053:  ldarg.0
134         IL_0054:  ldfld      int32[0...,0...] Hanoi.Hanoi::moves
135         IL_0059:  ldarg.0
136         IL_005a:  ldfld      int32 Hanoi.Hanoi::movesIndex
137         IL_005f:  ldc.i4.1
138         IL_0060:  ldarg.2
139         IL_0061:  callvirt   instance void
      int32[0...,0...]::Set(int32, int32, int32)
140         IL_0066:  ldarg.0
141         IL_0067:  ldarg.0
142         IL_0068:  ldfld      int32 Hanoi.Hanoi::movesIndex
143         IL_006d:  ldc.i4.1
144         IL_006e:  add.ovf
```

```
145        IL_006f:  stfld      int32 Hanoi.Hanoi::movesIndex
146        IL_0074:  ldarg.0
147        IL_0075:  ldarg.3
148        IL_0076:  ldarg.2
149        IL_0077:  ldarg.1
150        IL_0078:  ldarg.s    4
151        IL_007a:  ldc.i4.1
152        IL_007b:  sub.ovf
153        IL_007c:  callvirt   instance void
      Hanoi.Hanoi::DoMoves(int32, int32, int32, int32)
154        IL_0081:  nop
155        IL_0082:  nop
156        IL_0083:  nop
157        IL_0084:  ret
158      }
159    }
```

B.1.3 Thumbprinted IL Code of Hanoi

Original IL code of Hanoi contains three methods from which *DoMoves()* is thumbprinted using proposed thumbprinting scheme. Code provided below is thumbprinted version of IL code of *Hanoi* in which *DoMoves()* method is containing thumbprint value 35_{10} (i.e. 100011_2). The original code is simplified so that to highlight the thumbprinted code block. We can observe that on an average 6-7 original lines of code are converted to 13 lines of thumbprinted code that adds only 7 or 6 extra lines of code to encode a thumbprint once in a method. Another observation is that the thumbprint is tried to be encoded in last lines of a method which ends at a return (*ret*) or a no-operation (*nop*) statement. This thumbprint encoding place can be varied but here this is done so for experimental purpose.

```
''' Thumbprinted IL version of class 'Hanoi'
  .class public auto ansi Hanoi
       extends [mscorlib]System.Object
  {
    .method private instance void DoMoves(int32
sourceSpindle, int32 destSpindle, int32 tempSpindle,
int32 discsToMove) cil managed
    {
      .maxstack 6
      .locals init (int32,bool)

      IL_0000:  nop
      IL_0001:  ldarg.s    4
      :   :
      :   :
      IL_0077:  ldarg.1
      IL_0078:  ldarg.s    4
```

```
1         IL_007a:  ldc.i4.1
2         IL_011a:  br.s     IL_007b
3         IL_0082:  nop
4         IL_011b:  br.s     IL_0083
5         IL_0083:  nop
6         IL_011c:  br.s     IL_0084
7         IL_0081:  nop
8         IL_011d:  br.s     IL_0082
9         IL_007c:  callvirt   instance void
      Hanoi.Hanoi::DoMoves(int32, int32, int32, int32)
10        IL_011e:  br.s     IL_0081
11        IL_007b:  sub.ovf
12        IL_012a:  br.s     IL_007c
13        IL_0084:  ret
        }
      }
```

B.1.4 Branch Function Implemented Code of Hanoi

```
1     Public Class Hanoi
2           Public Function GetMoves() As Integer(,)
3                  Dummy1()
4                    Return moves
5           End Function
6
7           Public Function Dummy1() As Integer(,)
8                       DoMoves(0, 1, 2, discCounts(0))
9                       Return moves
10          End Function
11
12          Private Sub DoMoves(ByVal sourceSpindle As Integer, _
13                ByVal destSpindle As Integer, _ByVal tempSpindle As
      Integer,    _ByVal discsToMove As Integer)
14          End Sub
15    End Class
```

Appendix C

INTRODUCTION TO MSIL

MSIL code comprises of two main components: metadata and managed code. Metadata is a system of descriptors of all structural items of the application—classes, their members and attributes, global items, and so on—and their relationships. The managed code represents the functionality of application's methods encoded in this abstract binary form known as IL, which defines CLR compliant instructions.

The diagram below shows the sequence of a .NET program execution, from source code to IL code and from IL code to native machine code.

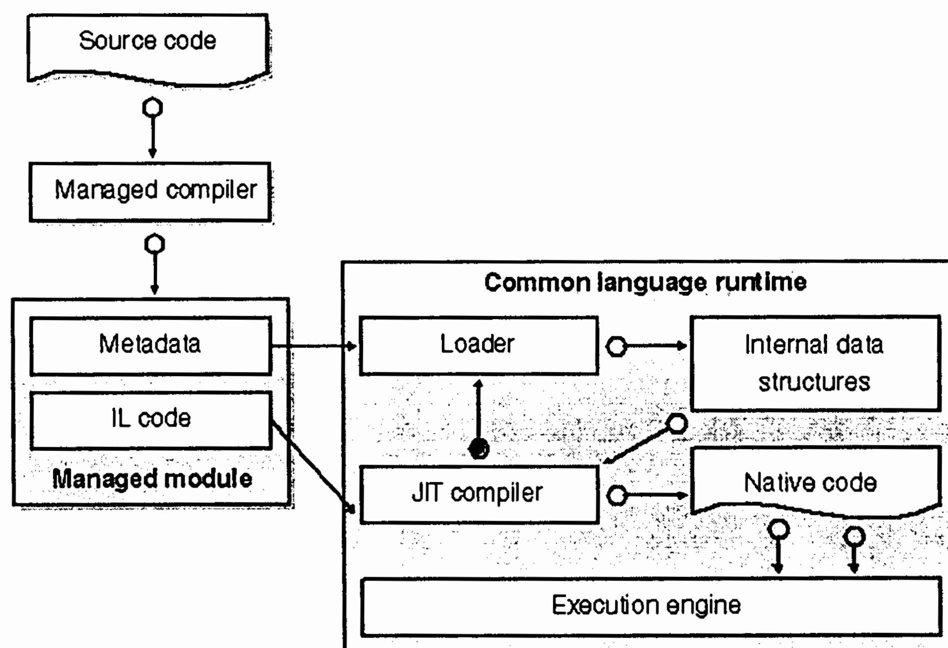


Figure C.1 Execution of managed .NET application [18]

MSIL is a stack-based set of instructions designed to be easily generated from higher level language source code by .NET compilers and other compliant tools. Several kinds of instructions are provided, including instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and method invocation. There is also a set of IL instructions for implementing object-oriented programming constructs such as virtual method calls, field access, array access, and object allocation and initialization. The IL instruction set can be directly interpreted by simply tracking the data types on the stack and emulating the IL instructions.

Here we examine how IL code will look like. We will observe a trivial "HelloWorld" example for both of its C# version and equivalent MSIL version.

C#

`public static void main(string[] args)`
`{`
`System.out.println("Hello world!");`
`}`

MSIL

`.method static void main(string[] args) cil managed`
`{`
`.entrypoint`
`.maxstack 8`
 	
`IL_0000: nop`
`IL_0001: ldstr "Hello world!"`
`IL_0002: call void [mscorlib]System.Console::WriteLine(string)`
`IL_0003: nop`
`IL_0004: ret`
`}`

Table C.1 Code Snippet of *HelloWorld* Program

We can observe that a single line method written in C# is compiled into several IL instructions. There will be a single IL code file of any .NET executable. Its code starts by defining the application metadata (like assembly information) that is followed by IL code in hierarchical structure (like namespaces containing classes and classes containing methods' code). Above code snippet is showing the inner most block of code (i.e. disassembly of a method '*main ()*'). This is default method from which .NET applications starts running. This is why the first line in IL code we see the word '*.entry point*' which helps .NET compiler to start application running from this method. Second line of code '*.maxstack 8*' defines the number of slots to be reserved in memory which the identifiers of this method will use at execution time. Subsequent instructions comprises of functional code of the method, each of these lines of code start with a label like 'IL_0000'. .NET compiler only requires these labels for those instructions which are targets of some jump call. As there is no jump calls in this snippet so these labels only serve as statement identifiers. Visual Studio compiler starts compiling from first instruction and stops where finds the first occurrence of '*ret*' statement in a method. As

we see the last statement in above code is '*ret*' so compiler will transfer control to the caller of this method.

We have presented a simple introduction of this IL code block over here where as more comprehensive details can be viewed at [18].

Appendix D

USING *ildasm* AND *ilasm*

This appendix presents a simplified scenario illustrating how to use Microsoft IL assembler and disassembler through a trivial *HelloWorld* example.

D.1 Disassembling .NET Application

Microsoft .NET framework is shipped with two built-in tools called *ilasm* (IL Assembler) and *ildasm* (IL Disassembler). *ildasm* disassembles an executable file like .exe or .dll and produce its IL code. It can be found in directory *[DIR]:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\Bin* on a system where .NET framework is installed. In order to execute this tool go to above directory and run *ildasm.exe*. It will open the disassembler window as shown in Figure D.1.

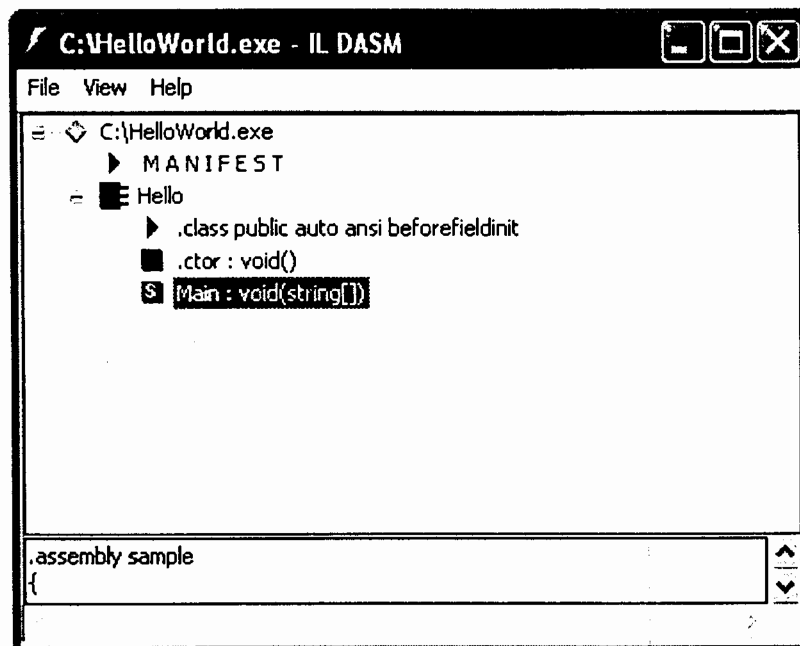


Figure D.1 Disassembling *HelloWorld.exe* through *Ildasm*

We opened *HelloWorld.exe* in *ildasm* and its disassembly is shown in above figure. If we observe different nodes in the tree, we see the root node '*C:\HelloWorld.exe*' is displaying the name of executable file that is disassembled. Second node titled as '*Manifest*', contains the program metadata as assembly information, which is shown in Figure D.2. It defines references to external assemblies which this application imports

like *mscorlib*, etc. Moreover it defines certain properties of this assembly which are required by .NET runtime environment.

```
MANIFEST
Find  Find Next
// Metadata version: v2.0.50727
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )                         // .z
  .ver 2:0:0:0
}
.assembly sample
{
  .custom instance void [mscorlib]System.Runtime.CompilerServices.Compilatio
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module sample.exe
// MVID: {CF554140-297D-4A5A-B81D-6B68933B50B3}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x03450000
```

Figure D.2 Metadata Information as Contained in Program Manifest

Third node is the name of assembly 'Hello' and rests of the nodes are classes and methods it contains. If we double click the last node titled '*Main: void(string[])*', it will open the IL code of this method as shown in Figure D.3.

```
Hello::Main : void(string[])
Find  Find Next
.method public hidebysig static void  Main(string[] args) cil managed
{
  .entrypoint
  // Code size       13 (0xd)
  .maxstack  8
  IL_0000:  nop
  IL_0001:  ldstr      "Hello World!"
  IL_0006:  call       void [mscorlib]System.Console::WriteLine(string)
  IL_000b:  nop
  IL_000c:  ret
} // end of method Hello::Main
```

Figure D.3 Disassembling Method *Main(string[])* through *Ildasm*

D.2 Modifying IL Code

Modifying program code at this level requires reasonable level of expertise in writing program in IL code. It is critical because these pseudo assembly instructions are sturdily connected to each other and altering them without consideration will result in program syntax and semantics errors. To modify this code we will take this block to some text editor and change it with valid IL code statements. Then a complete IL code file will be created that will contain rest of unchanged IL code of all methods of this application along with this modified code block. This new file is required to be saved with '*.il*' extension. Here is the complete IL code of modified HelloWorld.il.

```
// Metadata version: v2.0.50215
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )         // .z\V.4..
  .ver 2:0:0:0
}
.assembly sample
{
  .custom instance void [mscorlib]
    System.Runtime.CompilerServices.CompilationRelaxationsAttribute::
  .ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module sample.exe
// MVID: {A224F460-A049-4A03-9E71-80A36DBBBCD3}
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY

.class public auto ansi beforefieldinit Hello
       extends [mscorlib]System.Object
{
  .method public hidebysig static void  Main(string[] args) cil managed
  {
    .entrypoint
    // Code size       13 (0xd)
    .maxstack  8
    IL_0000:  nop
    IL_0001:  ldstr     "I am Modified!"     // changed from HelloWorld!
    IL_0006:  call      void [mscorlib]System.Console::WriteLine(string)
    IL_000b:  nop
    IL_000c:  ret
  } // end of method Hello::Main

  .method public hidebysig specialname rtspecialname
          instance void  .ctor() cil managed
  {
    // Code size       7 (0x7)
    .maxstack  8
    IL_0000:  ldarg.0
    IL_0001:  call      instance void [mscorlib]System.Object::.ctor()
    IL_0006:  ret
```

```
  } // end of method Hello::.ctor
} // end of class Hello
```

Figure D.4 Complete IL Code of HelloWorld.il

D.3 Re-Assembling IL Code

Above IL code file is given to *ilasm* to build a modified executable instance that will show intended modifications. *Ilasm* assembles valid IL code to build an executable file like *.exe* or *.dll*. It can be found in directory *[DIR]:\WINDOWS\Microsoft.NET\Framework\v2.0.50727*. In order to execute this tool, run system command prompt or use command prompt companioned with Visual Studio. In order to re-assemble IL code to an executable file we move to the above directory and run *ilasm* by specifying required parameters as shown in Figure D.5. The first parameter is the name of file containing complete IL code exhibiting valid syntax and semantics. Second parameter is the output switch, like '*/exe*' to get an *.exe* file or '*/dll*' to get a *.dll* file. Here we require an executable so we specify /exe over here. There are several other parameters which can be used for different purposes like displaying debug information, etc. Refer to [18] to get a complete list of *ilasm* commands and their usage details.

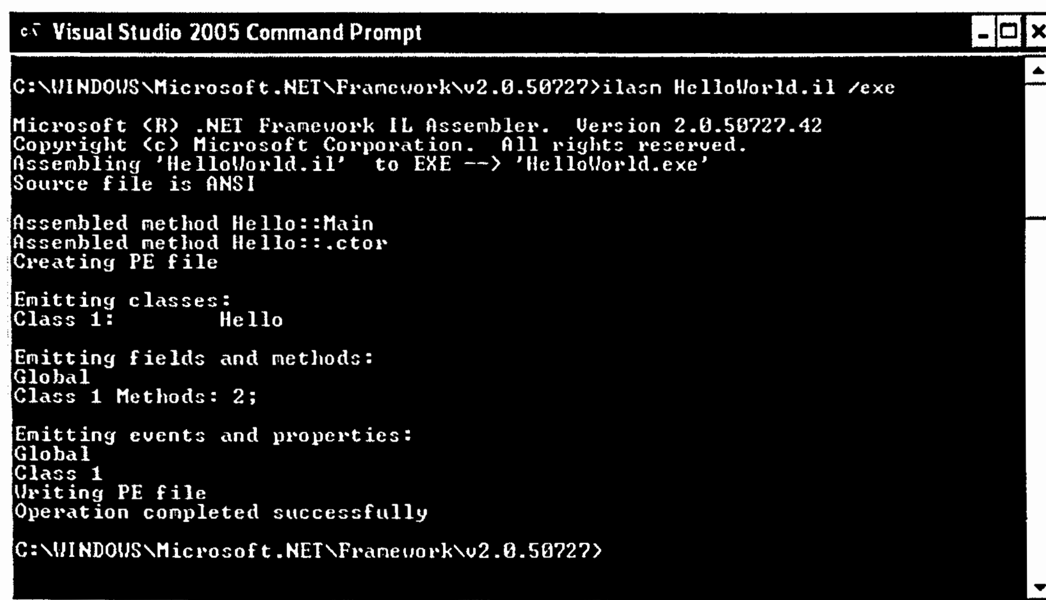


Figure D.5 Re-Assembling Modified *HelloWorld.il* through *Ilasm*

The second last line in above screen is displaying a message stating successful re-assembling of modified *HelloWorld.il* into *HelloWorld.exe*. If *ilasm* find some syntax error in IL code then this message will be changed to a failure message.