A Thesis for the Degree of Master of Science

Tamper Resistance for Software Protection

Ping Wang

School of Engineering

Information and Communications University

2005

Tamper Resistance for Software Protection

Tamper Resistance for Software Protection

Advisor : Professor Kwangjo Kim

by

Ping Wang
School of Engineering
Information and Communications University

A thesis submitted to the faculty of Information and Communications University in partial fulfillment of the requirements for the degree of Master of Science in the School of Engineering

Daejeon, Korea
Dec. 24. 2004
Approved by

(signed)

Professor Kwangjo Kim
Major Advisor

Tamper Resistance for Software Protection

Ping Wang

We certify that this work has passed the scholastic standards required by Information and Communications University as a thesis for the degree of Master of Science

Dec. 24. 2004

Approved:

Chairman of the Committee
Kwangjo Kim, Professor
School of Engineering

Committee Member
Daeyoung Kim, Assistant Professor
School of Engineering

Committee Member
Jae Choon Cha, Assistant Professor
School of Engineering

M.S. Ping Wang
20032100

Tamper Resistance for Software Protection

School of Engineering, 2005, 39p.
Major Advisor : Prof. Kwangjo Kim.
Text in English

Abstract

Protection of software code against illegitimate modifications by its users is a pressing issue to many software developers. Many software-based mechanisms for protecting program code are too weak (e.g., they have single points of failure) or too expensive to apply (e.g., they incur heavy runtime performance penalty to the protected programs). In this thesis, We present and explore a methodology that we believe can protect program integrity in a more tamper-resilient and flexible manner. we describe a dynamic integrity verification mechanism designed to prevent modification of software. The mechanism makes use of multi-blocking encryption technique so that no hash value comparison is needed and if the program was altered, the program will not exit in a traceable way. We also make use of common virus techniques to enhance our security. Our mechanism operates on binaries that can be applied to all PE format files like EXE and DLL. The overhead in runtime execution and program size is reasonable as illustrated by real implementation.

Contents

List of Figures

List of Abbreviations

PE format Portable Executable Format

BBS Bulletin Board Systems

DRM Digital Rights Management

MD5 Message-Digest Algorithm 5

CRC Cyclic Redundancy Checks

NP Nondeterministic Polynomial Time

IVK Integrity Verification Kernel

API Application Programming Interface

PKCS Public-Key Cryptography Standards

gzip GNU zip

EXE Executable Program

DLL Dynamic Link Library

Chapter 1
Introduction

Every year software industry has to face a cost of several billion dollars due to software piracy. Thirty-six percent of the software installed on computers worldwide was pirated in 2003, representing a loss of nearly $29 billion. As soon as computers started to became popular unauthorized copying of software started to be considered an important problem. Development of computer communications brought the growth of BBS services distributing pirated software. Today, other circumstances like the advances in code analysis tools and the popularity of Internet creates new opportunities to steal software. Some of the money lost because of the software piracy is included in the cost of legal software and therefore pirate copies are partially paid by the legal users.

Software protection has recently attracted tremendous commercial interest, from major software vendors to content providers including the movie and music recording industries. Their digital content is either at tremendous risk of arriving free on the desktops of millions of former paying customers, or on the verge of driving even greater profits through new markets. The outcome may depend in large part on technical innovations in software protection, and related mechanisms for digital rights management (DRM) - controlling digital information once it resides on a platform beyond the direct control of the originator. Privacy advocates are interested in similar mechanisms for protecting personal information given to others in digital format. Related activities include Microsoft's heavy investment in a next generation trusted

hardware platform (Palladium) [40], and the recent award by the U.S. Air Force Research Laboratory of a US$1.8M research contract involving software obfuscation [38].

Today's complex software is of much value to its creator. Whether that be a company with many products, or the only product of a small company. Software piracy is a major economic problem. Great losses to software producers are due to unauthorized use and distribution of their products. Of much concern is the protection of this software, such that it will always retain the functionality which its creators intended, always protect the intellectual property embedded in the program, and thwart attempts to make illegal copies of the program. Typically, the adversary creates modified versions of the software, which disable authentication code like serial number checking, and then illegally redistributed to the public.

Most of the software that is produced today has either weak protection mechanisms (serial numbers, user/password, etc.) or no protection mechanisms at all. This lack of protection is essentially derived from the user resistance to accept protection mechanisms that are inconvenient and inefficient. In Bruce Schneier words: "The problem with bad security is that it looks just like good security". Many commercial software protection tools claim to achieve total security with software techniques. Most of these tools sold without consideration of its quality or its ability to fulfil its vendor's claims. Theoretic approaches to the formalization of the problem have demonstrated that a solution that is exclusively based in software is unfeasible [1, 18].

In order to prevent tampering attacks, tamper-proofing [2, 5, 6, 12, 16, 22, 25, 26, 37] code should detect if the program has been altered and causes the program to fail when tampering is evident. There have been several different approaches proposed in an effort to deal with such attacks, but most of the commercially available defenses rely on reactive measures. Almost anyone will agree that software should be protected, but little is agreed upon as to how this should be done. At the root of the problem is the need for a solution

that relies on proactive measures, which ultimately means modifications to the way software is made. Furthermore, a solution that requires no new software design paradigms (from a software engineer's viewpoint) and is fully automated is highly desirable.

Software protection falls between the gaps of security, cryptography and engineering, among other disciplines. Despite its name, software protection involves many assumptions related to hardware and other environmental aspects. A significant gulf currently exists between theory and practice. Inconsistencies have arisen in the relatively sparse (but growing) open literature as a result of differences in objectives, definitions and viewpoints. All of these issues provide research opportunities.

1.1 Proactive Solutions

In order to take a proactive approach, one must be able to either determine that the program has been altered, or make any alterations impossible. Tamper resistance and code obfuscation attempt to address these requirements. What is of much interest, as we will see later, is that these two approaches can be combined for a robust and tamper-resistant solution.

In order for a program to detect whether or not it has been altered requires that the program check itself every time it is run. Because it must be checked at runtime, the most obvious solution is to insert tamper-proofing code into the program itself.

The other approach is to render a program unintelligible to its adversary through obfuscation. The basic premise being, that if you don't know what you are looking at, then it is impossible to intelligently alter the code. One problem that arises from this approach is that it is possible to obfuscate a program to such a degree, that even the creator can no longer tell what he/she is looking at. This can cause major problems when one must debug software through the use of stack dumps, assembly traces, and/or memory inspection.

One would argue that the solution to this problem would be to debug the code prior to obfuscation, but this may not be possible. After a piece of software is obfuscated and deployed, the end user may experience bugs that were not seen during testing of the original code. Tracking down problems at this stage may become impossible.

A tamper resistant approach that detects and/or subverts/corrects the tampering actions in real time (concurrently with the program execution) is desirable. Ultimately, a technique that will protect the software transparently, without the user even knowing such actions are taking place, will succeed.

1.2 What We Are Up Against

Observation of the historical trends suggest that the attack methods appear to be more mature than (lead in time) the security methods. Attacks use many readily available tools which allow them to monitor network connections, monitor a program's instructions with debuggers, modify an operating system's kernel1, monitor address and memory busses, etc. It seems somewhat ironic that the tools used to help design and implement complex software, are the same tools used to attack it.

Much has been done to thwart network originated attacks, but little has been done to thwart hardware and software based attacks on the intellectual property embedded within a program. These attacks include modifications to a program to skip crucial checks (such as license file/servers), or reverse engineering of a key piece of a program's functionality.

The anti-tamper techniques in general are designed to detect or sense any type of tampering of a program. Once such tampering is detected, one of many possible actions could be taken by the anti-tamper part of the software. These actions could include disabling the software, deleting the software, or making the software generate invalid results rendering it useless to the tampering adversary.

1.3 Our Contributions

In this thesis, we initiate the use of multi-blocking encryption technique [2], which was originally used to resist code observation, in integrity checking and propose a multi-block hashing scheme. The mechanism behind multi-blocking encryption is to break up a binary program into individually encrypted segments. We base on this mechanism to perform the integrity checking. Roughly speaking, we will take the hash value of a block as the secret key for decrypting the next block. The advantages of our approach include the followings: No hash value checking is needed. If the program was altered, the hash value is changed and therefore the next block cannot be decrypted properly. Due to the corruption of the next block, the program cannot continue to run.

Also, using this dynamic multi-block hashing scheme, the integrity of the software is kept checking during the program run-time execution. The adversary (*i.e.*, the software pirate) is unable to obtain a single point of failure. Unlike the previous approaches that rely on branching instructions for checking the hash values, in our approach, we do not have any such instructions. The technique of bypassing the checking instructions is no longer feasible in our approach. On the other hand, if the program was altered, the program will not exit immediately. Therefore the adversary is very difficult to trace back the problem. The feasibility of our suggested approach is realized by a real implementation. Experimental results show that the overhead in runtime execution as well as the increase in program size is reasonable.

1.4 Outline of The Thesis

The rest of the thesis is organized as follows.

In Chapter 2, we review some early literature on software protection and discusses a selection of software protection approaches including software wa-

termarking, software obfuscation, software tamper resistance, integrity verification and software diversity.

Then, we discusses the idea of our multi-block hashing scheme and its security issues in Chapter 3.

In Chapter 4, we show the experimental results according to the implementation of our multi-blocking encryption technique to a software application.

Finally, we discuss the generalization of our approach and the conclusion in Chapter 5.

Chapter 2
Background and Related Work

There exists a wide range of tamper resistance methodologies. The following discusses some of the more widely know approaches.

It is also important to keep in mind, that in order to increase the effectiveness of tamper resistance, multiple approaches can be combined. One should think carefully about how to combine different approaches, and strive to mask the weaknesses of one, with the strengths of another. For example, combining control ow monitoring with obfuscation can lead to a monitored program that requires significant effort to reverse engineer(NP-complete) [37].

2.1 Watermarking

Watermarking consists of statically, or dynamically inserting 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 a program's image, or from the program execution itself. Watermarking, as mention previously, is a reactive measure. Hence, we will not be looking into watermarking as an effective technique. While this performs a valuable function, the idea is to avoid the need for this all together by making the program impossible to tamper with in the first place. Good representatives of software watermarking methods are [11, 16, 32, 33, 34].

Static watermarks evolved from the area of digital imaging. Watermarks for images have been around for quite some time, and are fairly mature. This idea has been used in program watermarking in a very simple way. First watermark a small image. Then embed the image into the data section of a program. This way, the image can be extracted from the program, and then the watermark from the image. It is rather apparent that such a simplistic approach is easily broken using standard binary editing tools.

Of more interest are the dynamic watermarks. For example, and Easter Egg watermark is a watermark that is embedded into the functionality of the program. When the program is given a particular input set, it performs some action that is immediately visible to the user. A typical Easter Egg watermark might display a logo, or force a program into a particular mode. For example, the following will turn Microsoft Word97 into a pinball game:

1. Open a new document
2. Type the word "Blue"
3. Highlight the word "Blue"
4. Using the Format menu select Font
5. Choose Font Style Bold, Color Blue
6. Type " " (space) after word "Blue"
7. Using the Help menu select About
8. Ctrl-Shift-Left click the Word icon/banner
9. Use Z for left flipper, M for right flipper, and ESC to exit

Other dynamic watermarks include execution tracing and data structure analysis, both producing no immediate output for the user, but instead relying on monitoring a particular property of the program when given special input. Because of the nature of these two watermarks, they do not work well with most types of code obfuscation.

2.2 Obfuscation

One way to protect a software program is to prevent tampering by increasing the difficulty for hackers to attack the software. There are several techniques that have been proposed in this direction. Code obfuscation [37] attempts to transform a program into an equivalent one that is more difficult to manipulate and reverse engineer. One example is the Java byte code obfuscators. A major drawback of all obfuscation approaches is that they are of necessity ad hoc. Unless provably effective techniques can be developed, each obfuscation is almost always immediately followed by countermeasures. In fact, Barak *et al*. showed in [3] that it is inherently impossible to systematically obfuscate program codes. Another technique that can provide provable protection against tampering is to encrypt programs and the program can be executed without needing to decrypt them first. Sander and Tschudin proposed cryptographic function [28], a way to compute with encrypted functions. They identified a class of such encrypted functions, namely polynomials and rational functions. Clearly not all programs fit into this category.

Code obfuscation attempts to make the task of reverse engineering a program daunting and time consuming. This is done by transforming the original program into an equivalent program, which is much harder to understand, using static analysis [7, 13, 15].

More formally, code obfuscation involves transforming the original program P' into a new program P' with the same black box functionality. P' should be built such that [12]:

I It maximizes obscurity, *i.e.*, it is far more time consuming to reverse engineer P' when compared to P.

II It maximizes resilience, *i.e.*, P' is resilient to automated attacks. Either they will not work at all, or they will be so time consuming that they will not be practical.

III It maximizes stealth properties, *i.e.*, P' should exhibit similar statis-

tical properties, when compared to P.

IV It minimizes cost, *i.e.*, the performance degradation caused by adding obfuscation techniques to P' should be minimized.

Obfuscation techniques involve lexical, control and data transformations. Lexical transformations alter the actual source code, such as Java code. This transforms the original source code into a lexically equivalent form by mangling names and scrambling identifiers. Such transformations make it a daunting task to reverse engineer a program. A simple example would be to swap the names of the functions *add*() and *subtract*(). (This would also involve swapping every reference to these functions as well.) An even more interesting approach would to replace *add*() with the function *tcartbus*() and replace *subtract*() with the function *sunim*().

Control transformations alter the control flow of the program by changing branch targets to an ambiguous state. The code for the program is shuffled such that the original branch targets are no longer correct. During this shuffling, the new targets are calculated, and code is inserted in place of the old branch instruction to acquire its new target address.

Data transformations rearrange data structures such that they are not contiguous. Data can be transformed all the way down to the bit level. Bit interleaving is one example.

One particular obfuscation technique of interest is obscuring control flow of a program. By obscuring branch target addresses, static analysis of a control flow graph can be shown to be NP-hard [37]. Program address based obfuscation is presented in [3].

Several researchers have published papers on software obfuscation using automated tools and code transformations [14, 15]. One idea is to use language-based tools to transform a program (most easily from source code) to a functionally equivalent program which presents greater reverse engineering barriers. If implemented in the form of a pre-compiler, the usual portability issues can be addressed by the back-end of standard compilers. For design

descriptions of such language-based tools see [19, 27, 32]. Cohen [10] suggested a similar approach already in the early 1990s, employing obfuscation among other mechanisms as a defense against computer viruses. Cohen's early paper, which is strongly recommended for anyone working in the area of software obfuscation and code transformations, contains an extensive discussion of suggested code transformations [12, 13]. Wang [35] provides an important security result substantiating this general approach. The idea involves incorporating program transformations to exploit the hardness of precise interprocedural static analysis in the presence of aliased variables [13], combined with transformations degenerating program flow control. Wang shows that static analysis of suitably transformed programs is NP-hard. Collberg *et al.* [13] contains a wealth of additional information on software obfuscation, including notes on: a proposed classification of code transformations (*e.g.*, control flow obfuscation, data obfuscation, layout obfuscation, preventive transformations); the use of opaque predicates for control flow transformations (expressions difficult for an attacker to deduce, but whose value is known at compilation or obfuscation time); initial ideas on metrics for code transformations; program slicing tools (for isolating program statements on which the value of a variable at a particular program point is potentially dependent); and the use of (de)aggregation of flow control or data (constructing bogus relationships by grouping unrelated program aspects, or disrupting legitimate relationships in the original program).

2.3 Tamper Resistance

Software obfuscation provides protection against reverse engineering [8, 9, 23, 31], the goal of which is to understand a program. Reverse engineering is a typical first step prior to an attacker making software modifications which they find to their advantage. Detecting such integrity violations of original software is the purpose of software tamper resistance techniques. Software

tamper resistance has been less studied in the open literature than software obfuscation, although the past few years has seen the emergence of a number of interesting proposals.

Tamper resistance is the art and science of protecting software or hardware from unauthorized modification and distribution. Although hard to characterize or measure, effective protection appears to require a set of tamper resistance techniques working together to confound an adversary. Algorithms like MD5 and CRC are commonly used for integrity checking of the software. A common approach is to hash the whole block of software to obtain a hash value. To check the integrity of that software, this hash value will be compared with the hash value calculated based on the current copy of the software before running. If the two hash values do not match, the software has probably been modified and the program will be terminated. However, this static hash value checking is easily bypassed by locating the hash value comparison instruction and modifying the binary program code with existing software debugging tools. This branching instruction that performs the hash values comparison becomes a single point of failure.

Self-checking means (also called self-validation or integrity checking), while running, program checks itself to verify that it has not been modified. We distinguish between static self-checking, in which the program checks its integrity only once, during start-up, and dynamic self-checking, in which the program repeatedly verifies its integrity as it is running. Self-checking alone is not sufficient to robustly protect software. The level of protection from tampering can be improved by using techniques that thwart reverse engineering, such as customization and obfuscation, techniques that thwart debuggers and emulators, and methods for marking or identifying code, such as watermarking or fingerprinting. These techniques reinforce each other, making the whole protection mechanism much greater than the sum of its parts.

Fundamental contributions in this area were made by Aucsmith [2]. Aucsmith defines tamper resistant software as software which is resistant to obser-

vation and modification, and can be relied upon to function properly in hostile environments. An architecture is provided based on an Integrity Verification Kernel (IVK) which verifies the integrity of critical code segments. The IVK architecture is self-decrypting and involves self-modifying code. Working under similar design criteria (e.g. to detect single bit changes in software), Horne *et al.* [21] also discuss self-checking code for software tamper resistance. At run time, a large number of embedded code fragments called testers each test assigned small segments of code for integrity (using a linear hash function and an expected hash value); if integrity fails, an appropriate response is pursued. The use of a number of testers increases the attacker's difficulty of disabling testers.

2.4 Integrity Verification

In the last few years, there have been active development of software obfuscation [13] and watermarking [11] but only a few researches have been done in software integrity verification. Computing a hash value of code bytes is a common integrity verification method [4, 5, 6, 24]. It examines the current copy of the executable program to see if it is identical to the original one by checking the hash values. The main drawback of this approach is that the adversary can easily bypass the verification by locating the hash value comparison instruction. Furthermore, since this method only verifies the static shape of the code, it cannot detect run-time attacks, where the debugger tools monitor the program execution and the adversary can identify the instructions that are being executed and then modify them.

Hashing functions scan a block of the program, and use the data contained therein as input to a mathematical equation. The simplest way of which would be to sum all the numbers in a given block. When done, the output must agree with the previously determined result. If they are not the same, this block of the program has been altered.

It is fairly obvious that such a scheme is easily broken, which has led to more complex techniques. Some techniques may use values on the stack at a crucial instant in time, or values in a register. Others perform complex mathematical equations on overlapping sections of code. Horne *et al.* [21] have implemented such a system, using linear hash functions, which overlap and also hash the hashing functions as well. One of the strong points of hashing is that you can have as many hashing functions as you want, all performing a different hash fuction.

In order to detect run-time attacks, Chen and Venkatesan proposed oblivious hashing [6] based on the actual execution of the code. This method examines the validity of intermediate results produced by the program. It is accomplish by injecting additional hashing codes into the software. These hash codes are calculated by taking the results of previous instructions from the memory. In other words, the hash values are calculated based on the dynamic shape (run-time states) of the program so as to make it more difficult to attack. However, there is a practical constraint for binary-level code injection and if the adversary can locate the instructions for hash value comparison, bypassing is still possible.

Chang and Atallah [5] proposed another method that enhances the run-time protection in which protection is provided by a network of execution unit call guards. A guard regarded as a small code segment which performs checksums on part of the binary code to detect if the software has been modified. The guards are inserted into the software with different locations. They are inter-related so as to form a network of guards that reinforce the protection of one another by creating mutual-protection.

They also proposed the use of guards that actually repair attacked code. Although a group of guards is more resilient against attacks than a single branching instruction for comparing hash values, it only spreads out the single attack point into different locations of the program. In other words, although more complicated, bypassing instructions performed by the guards is still

possible.

The above integrity checking techniques all involve hash value comparison, which can be quite easily bypassed. Recently, Collberg and Thomborson [12] discussed an innovative idea that suggests encrypting the executable, thereby preventing anyone from modifying it successfully unless the adversary is able to decrypt it. However, the details are not discussed in their paper and only very little researches focusing on this kind of technology can be found. Our suggested approach can be regarded as the same direction as the idea proposed by them in which encryption is used instead of hash value comparison.

2.5 Diversity

Diversity is an idea which is part of the software folklore, but it appears to only recently have received significant attention in the security community. The fundamental idea is simple: in nature, genetic diversity provides protection against an entire species being wiped out by a single virus or disease. The same idea applies for software, with respect to resistance to the exploitation of software vulnerabilities and program-based attacks. In this regard, however, the trend towards homogeneity in software is worrisome: consider the very small number of different Internet browsers now in use; and the number of major operating systems in use, which is considerably smaller than 10 years ago.

The value of software diversity as a protection mechanism against computer viruses and other software attacks was well documented by Cohen [10] in 1992-1993. The architecture of automated code transformation tools to provide software obfuscation (see Section 2.2) can be modified slightly to provide software diversity: rather than creating one new instance of a program which is functionally equivalent to the original (and hard to reverse engineer), create several or many. Here the difficulty of reverse engineering or tampering with a single program instance is one security factor, but a

more important factor is that an exploit crafted to succeed on one instance will not necessarily work against a second. Forrest *et al.* [17] pursue this idea in several directions, including simple randomizations of stack memory to derail high-profile buffer-overflow attacks.

The idea of relying on diversity for improving the reliability and survivability of networks (*e.g.*, see Wang [35] for contributions and references) has gained recent popularity, subsequent to incidents of global terrorism. The value of diversity for security and survivability was also recognized in the 1999 Trust in Cyberspace report [30], among others.

2.6 Types of Attacks

When one decides to prevent attacks on their software, they must first decide what type of attack is of most concern to them. The following shows attacks classified into three basic categories. The main distinction being that each of these depend on the relative location of the origination of the attack [14, 20, 36, 39].

I Outside attackers attempting to gain entry over a networked connection. This is the most common type of attack today, and several preventive measures are already in place.

II Executable code that is run on a target system, but not under the direct control of the attacker, such as viruses and Trojan horses. This is a fairly common attack which has several preventive measures already in place as well.

III God Mode attacks: The attacker owns a copy of the software, and has complete control over the system it is run on. This is one of most damaging attacks in that it allows the theft of Intellectual Property, and the execution of pirated software.

The God Mode attack model assumes that the attacker has full control over the system, *i.e.*, the attacker owns the system the program is running on,

and has total access to the software and hardware in the system. The attacker may choose to run binary analysis tools, software and hardware debuggers, logic analyzers, etc..

IV Modifying the Testers One possible disabling attack is to modify one or more testers so that they fail to signal a modification of the tested code section.

V Temporary Modifications A dynamic attack might modify the code so that it behaves anomalously and then restore the code to its original form before the self-checking mechanism detected the change.

Chapter 3
Proposed Scheme

We propose the use of multi-blocking encryption for the integrity verification of software. In this chapter, we will first introduce the proposed multi-blocking encryption, then followed by the hashing scheme. Its security issues will also be discussed.

Multi-blocking encryption breaks up a binary program into individual encrypted blocks. The program is executed by decrypting and jumping to the executable block during the run-time process. When not being executed, blocks are in encrypted form after applying this program protection technique, therefore the adversary cannot modify the code statically, where the program being disassembled is examined by the disassembler which is not able to interpret the encrypted version of it.

3.1 Design Objectives

The fundamental purpose of a dynamic program self-checking mechanism is to detect any modification to the program as it is running, and upon detection to trigger an appropriate response. We sought a self-checking mechanism that would be as robust as possible against various attacks while fulfilling various non-security objectives.

Comprehensive and Timely Dynamic Detection The mechanism should detect the change of a single bit in any non-modifiable part of the program, as the program is running and soon after the change occurs. This

helps to prevent an attack in which the program is modified temporarily and then restored after deviant behavior occurs.

Separate, Flexible Response Separating the response mechanism from the detection mechanism allows customization of the response depending upon the circumstances, and makes it more difficult to locate the entire mechanism having found any part.

Modular Components The components of the mechanism are modular and can be independently replaced or modified, making future experimentation and enhancements easier, and making extensions to other executables and executable formats easier.

Platform Independence Although the initial implementation of our self-checking technology is Intel x86-specific, the general mechanism can be adapted to any platform.

Insignificant Performance Degradation The self-checking mechanism should not noticeably slow down the execution of the original code and should not add significantly to the size of the code.

Easy Integration We designed our self-checking technology to work in conjunction with copy-specific static water marking and with other tamper resistance methods such as customization. Embedding the self-checking technology in a program relies on source-level program insertions as well as object code manipulations.

3.2 Proposed Multi-blocking Encryption

Based on the concept of multi-blocking encryption in [2], we propose to apply this technique in the following way: We propose to divide a program into several different sized blocks (instead of equal sized blocks) according to the flow of the program. Each block is encrypted with a different key, which is illustrated in Figure 3.1. Let the program be P. If it can be broken down in several blocks, each basic block has the property of being indepen-

dent. This means that the block does not have any jump/branch instructions, jumping/branching to other blocks. In other words, all the targets of the jump/branch instructions are local within the block. In order to find out the basic blocks, we first need to disassemble the executable program P to its machine code instructions accurately. There are two generally used techniques for this: linear sweep and recursive traversal [31].

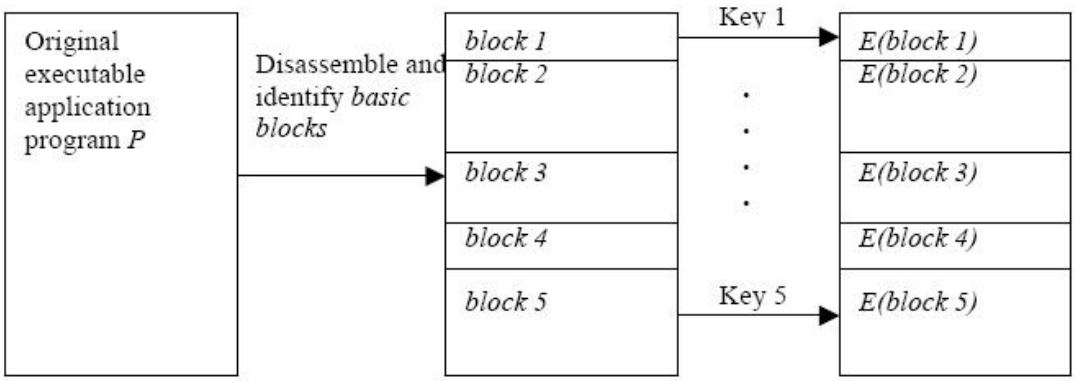

Figure 3.1: Different sized blocks encrypted by different keys

In general, it is convenient and feasible to store those keys inside the hardware token [39] so that dumping of keys from the main memory is impossible. The encryption can also be done inside the token. In our approach, we make use of the hash values of the blocks to be the encryption keys, thus further eliminate the necessity of storing the keys (see Section 3.4 for details).

3.3 Determining the Number of Blocks

The number of blocks ranges from the whole program (one-time encryption) to one instruction per block. It is clear to see that in the extreme environment, of which each block is one instruction, it can achieve the maximum-security level. The instruction can be decrypted inside a special CPU as well. However, it is infeasible to put such heavy workload into the CPU itself. Thus,

the number of blocks should be determined by striking a balance between the level of security to be achieved and the speed of the program. Note also that the blocks also depend on the actual control flow of the program.

3.4 Multi-block Hashing Scheme

In this section, we will describe our proposed multi-block hashing scheme. Our objective is to prevent an adversary from modifying the software. Assuming that a user has legal access to the software, he may try to tamper it to remove authentication code so that it can be freely distributed for illegal use.

Our scheme works as follows: We take the hash value of a basic block as the secret session key for decrypting the next basic block according to the flow of the program. As an initiate study, we focus on the programs of which the control flow is a tree-like structure (as shown in Figure 3.2). We remark that this kind of control flow is very common.

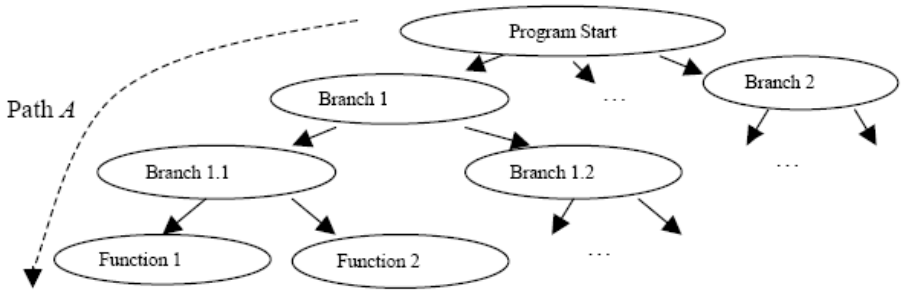

Figure 3.2: Tree structure of the program

Let the program be P and we consider any single path A. Let the path be broken down in n basic blocks, $b_1, b_2, \cdots, b_n$ such that the control flow of the path A starts at b_1 followed by b_2 and then b_3, etc. The blocks are in encrypted form except the starting block b_1. The jumping code to the decryption routine is placed inside the basic blocks. The program controller,

which implements the dynamic integrity verification, is stored at the end of the original program as illustrated in Figure 3.3.

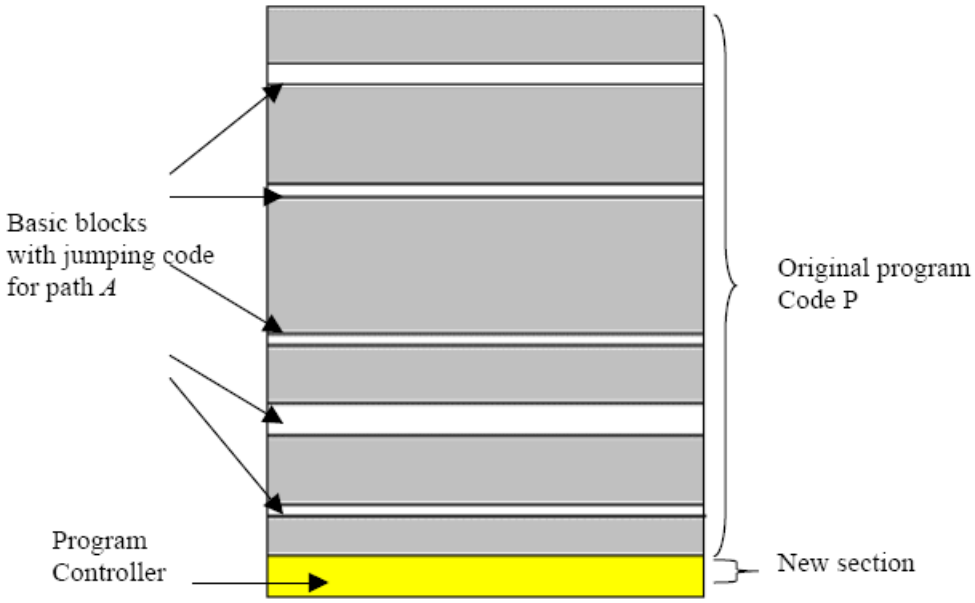

Figure 3.3: Structure of protected program

The entry point of the protected program is now set at the program controller. Once the initial state has been set up, the original program begins execution. Before the execution of b_i, calculate the hash value of $b_{i-1}, H_{i-1} = Hash(b_{i-1})$. We treat the hash value H_{i-1} as the secret session key $K_{i-1}(K_{i-1} = H_{i-1})$ for the decryption of the block b_i, provided that the number of bits for K_{i-1} is compatible with the encryption algorithm. If a hardware token was used, this can be implemented by using the $C\_DeriveKey$ API function of PKCS #11 [41]. $C\_DeriveKey$ can derive a secret key from a known data, H_i in our case. In order to illustrate the algorithm, we take $n = 3$ in the following:

Algorithm : $Multi - blocking$ *integrity check* (*during program execution*)

1. Before b_2 starts to execute, the program jumps to the program con-

troller to calculate $H_1 = Hash(b_1)$, where b_1 is in plaintext.

2. The secret key K_1 is then derived from H_1, *e.g.*, $K_1 = H_1$.

3. The second block $E(b_2)$ is decrypted by $K_1 : b_2 = D_{K1}(E(b_2))$.

4. The decrypted block is moved to its original place, followed by the execution of the decrypted codes.

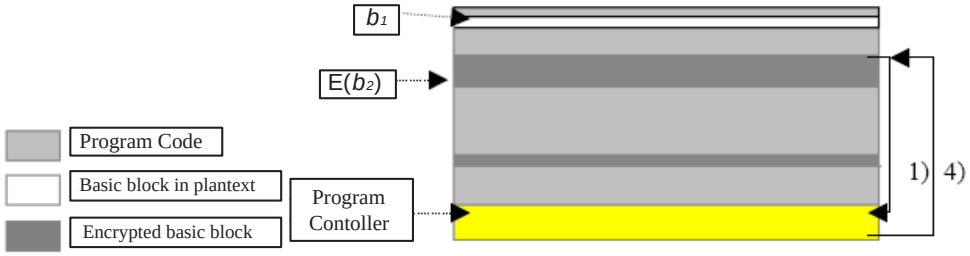

Figure 3.4: Program Progress 1 - 4

5. Before b_3 starts to execute, the program jumps to the program controller to calculate $H_2 = Hash(b_2)$.

6. The secret key K_2 is derived from H_2, *e.g.*, $K_2 = H_2$.

7. The third block $E(b_3)$ is decrypted by $K_2 : b_3 = D_{K2}(E(b_3))$.

8. The decrypted block is moved to its original place, followed by the execution of the decrypted codes.

In order to create different keys for encrypting different blocks, we use different hash values to achieve this property. There is no 'storage of keys'

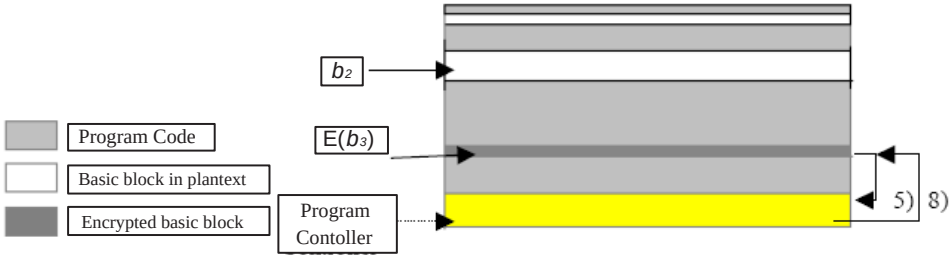

Figure 3.5: Program Progress 5 - 8

problem as the hash values are calculated dynamically during program execution. The above completes the treatment of a single path. For the whole tree structure program P, we can apply the algorithm on all paths inside the tree.

The adversary cannot modify the software statically as binary codes are in encrypted form after protection. For dynamic modification, suppose an adversary alters the running program in block b_i which will produce a different hash value H_i. Before the execution of b_{i+1}, the hash value H_i is not the proper decryption key K_i for block b_{i+1}, the result from the decryption will then produce rubbish code. Due to the corruption of the next block, the program cannot continue to run properly and crashes. It is a great advantage that the program will not halt its execution immediately after code modification. When the tampered program crashes, the adversary will find it very difficult to trace back the exit point.

Using this multi-block hashing method, no hash value comparison is present and bypassing the checking is impossible. The scheme is constructed so that any program state is in a function of all previous states. Therefore, the program is guaranteed to fail if one bit of the protected program is tampered with. The point of failure also occurs far away from the point of detection, so that the adversary does not know how it has taken place.

3.5 Analysis of Multi-block Hashing Scheme and Enhancement

We have described a dynamic software integrity verification scheme that made use of multi-block encryption technique. In contract to common hash value comparison schemes, this scheme does not use a single code block for integrity checking. This makes the adversary difficult to bypass the checking in the program.

To attack the scheme, the adversary may find out the hash value of each block by dynamic analysis. After finding out those hash values, he or she can replace the tampered hash value with the correct one during program execution and the program can decrypt the next basic block properly. However, the time taken by dynamic analysis is typically at least proportional to the number of instructions executed by the program at runtime. In other words, to attack our scheme, it takes a lot more effort than the attack for the previous schemes. In fact, we can further enhance the security of our approach in order to prevent the adversary to find out the hash values. One effective way to achieve this is to obfuscate the program codes. Some techniques are discussed in [12]. The use of multi-blocking encryption in our mechanism is, in fact, also one kind of obfuscation techniques.

On the other hand, we can also use the technique of code polymorphism [19] to prevent this problem, which means that the program code is mutated after each execution while preserving its semantics. Many computer viruses use this technique to prevent the anti-virus engines from finding them out. We use this idea to make our protection not noticeable by the adversary. One possible implementation is to mutate each basic block b_i after it has been executed and thereby changing its hash value. We can use the new hash value to re-encrypt the next block b_{i+1} and so on. This creates a new version of the same program with identical block decomposition. Even if the adversary can identify the hash values at the first time, those values cannot be used for the next execution of the program as the hash values have been mutated after the previous execution.

3.6 Comparison with Previous Schemes

Recently, there have been active development on software dynamic integrity verification. To our knowledge, Aucsmith [2] was the first to introduce the concept of multi-blocking encryption. The armored segment of code, which

call integrity verification kernel (IVK). The IVK is divided into several equal size blocks, which are encrypted by the same key. Blocks are exposed by decryption as it is executed. The multi-blocking encryption technique used in [2] is mainly used to resist code observation. In contrast to this approach, we divided the original program into different size blocks, and calculated the keys dynamically during program execution, encrypted the different blocks with different keys. There is no 'storage of keys' problem as the hash values are calculated dynamically during program execution. We make use of this technique to resist code modification during program execution.

Unlike mechanism in [21], which consists of a number of testers that redundantly test for changes in the executable code as it is running and report modifications, our scheme detects the modification of codes by decrypting the next block code, not depending on any tester. Whether encrypted blocks can be decrypted properly, entirely depends on whether there is no modification on software. Otherwise, the software can not run properly, sometimes crashed, rather than exit in a traceable way. With this property, our scheme can prevent single check point failure attack.

Also, Chang and Atallah [5] proposed another approach based on a distributed scheme, in which protection and tamper-resistance of program code is achieved, not by a single security module, but by a network of (smaller) security units that work together in the program. These security units, or guards, can be programmed to do certain tasks (check summing the program code is one example) and a network of them can reinforce the protection of each other by creating mutual-protection. Although a group of guards are more resilient against attacks than a single branching instruction for comparing hash values, it only spreads out the single attack point into different locations of the program. In other words, although more complicated, bypassing instructions performed by the guards is still possible. Our scheme have similar inter-related structure, but achieved higher level security by preventing static code analysis attack.

In [6], Chen and Venkatesan proposed a novel software integrity verification primitive, Oblivious hashing, which implicitly computes a fingerprint of a code fragment based on its actual execution. Its construction makes it possible to thwart attacks using automatic program analysis tools or other static methods. This new method verifies the intended behavior of a piece of code by running it and obtaining the resulting fingerprint. However, there is a practical constraint for binary-level code injection and if the adversary can locate the instructions for hash value comparison, bypassing is still possible. Since our multi-block hashing scheme decrypts necessary block during program execution, it can avoid run-time attack.

We compared the above three schemes on software dynamic integrity verification with our scheme, with respect to the ability against certain attacks, such as single check point failure attack, static code analysis attack, dump memory attack and run-time attack. The results illustrated in Figure 3.6.

	Dynamic Self-Checking	Protection by Guards	Oblivious Hashing	Our Scheme
Single Check Point Failure Attack	--	--	O	O
Static Code Analysis Attack	O	--	O	O
Dump Memory Attack	O	O	O	O
Run-time Attack	X	X	X	--

O: Completely against certain attack
--: Partially against certain attack
X: Vulnerable to certain attack

Figure 3.6: Comparison with Previous Schemes

Chapter 4
Experimental Results

In this chapter, we want to find out the overhead in terms of program size and execution time. We have implemented our multi-blocking encryption technique to a software application (see Appendix for the main code of this experiment): gzip [42] is a compression utility designed to be a replacement for compress and the control flow is in a tree structure. The corresponding binary object code is analyzed to determine the number of basic blocks to be used and the blocks information is passed to the installation program, as shown in Figure 4.1.

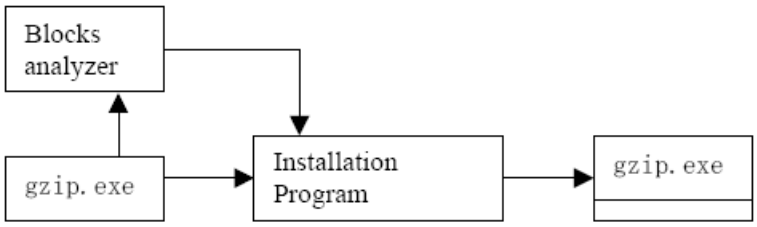

Figure 4.1: Installation of dynamic integrity verification scheme

We have applied the integrity verification scheme in one single path of the tree structure, which is a subset of the whole tree structure program. The program controller and information for each basic block are added to the end of the object code automatically after passing through the installation program. USB hardware token was used for encryption and hashing, which conforms to the PKCS #11 [41] standard. Those basic blocks are encrypted with AES and we use SHA-1 for hashing. The integrity verification scheme is

then installed into the program and is ready to be run. The installation process is automatic such that it is error free. The experiments were conducted on a Pentium 1.5GHz CPU clock with 256MB RAM. We have used two files to be compressed by the gzip program with the size 89.5KB and 1.01MB, respectively.

4.1 Program Performance

We want to find out if the multi-block hashing would impose unreasonable runtime overhead on the program. An experiment was conducted to measure the runtime overhead of multi-block hashing versus number of blocks used and the results are shown in Figure 4.2.

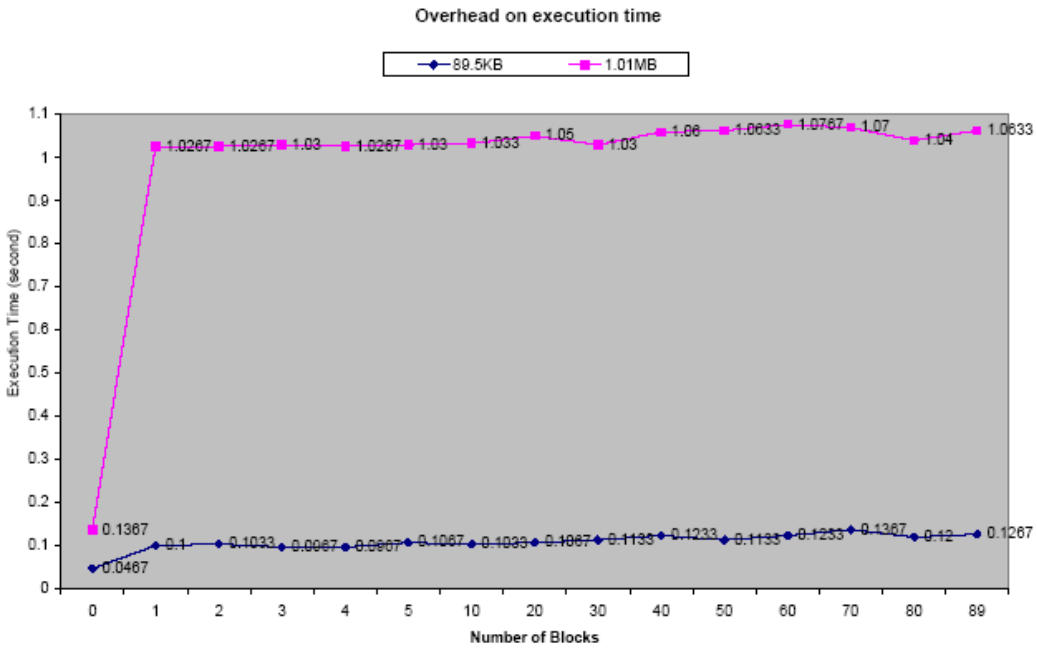

Figure 4.2: The average execution time of the protected gzip program with different number of blocks used

We have identified 89 blocks for the experiments. The execution time of

Windows programs may differ slightly every time the code is executed. We chose to find an average execution time for the gzip program. While there were many jumping between the original program and the program controller, the run time overhead was reasonable. The execution time of the original program was 0.0467s and 0.1367s, for zipping the 89.5KB and 1.01MB file, respectively. It took only 1.07s for zipping the 1.01MB file with 70 blocks used. The overhead is reasonable even the number of blocks used is large.

4.2 Program Size

The verification scheme installed gzip program size with different number of blocks installed was shown in Figure 4.3.

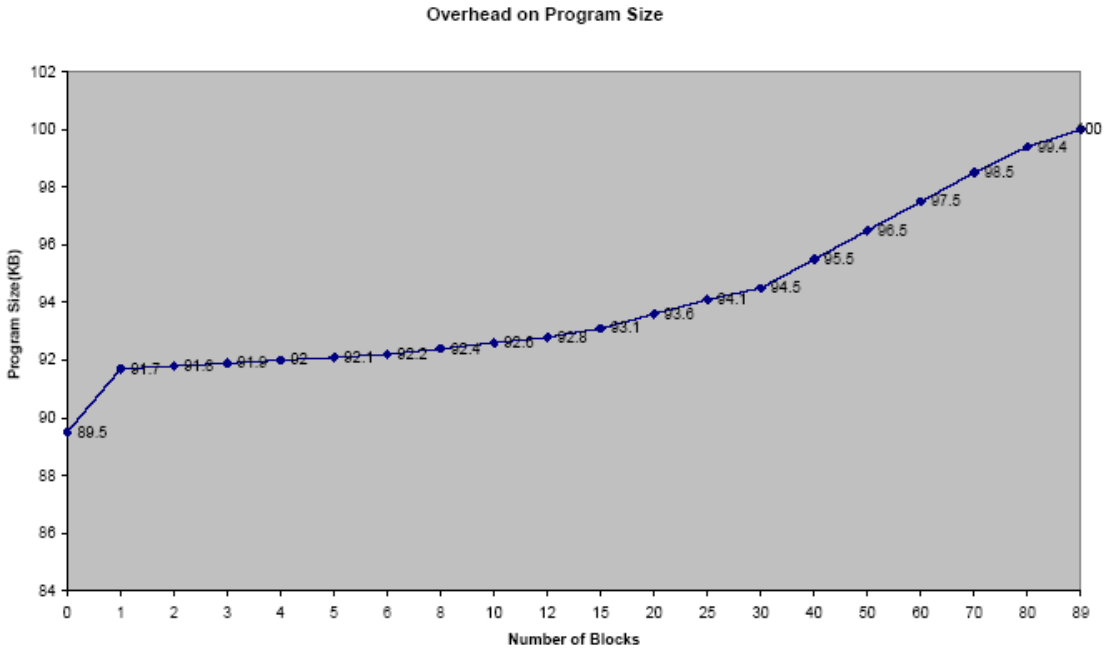

Figure 4.3: gzip program size with different number of blocks used

The increase in size for storing information of each block is proportional to the number of blocks identified in the program. It can be shown that

the size overhead for 30 blocks used is only 5.6% compared with the original program size, which is 89.5KB. It is relatively small as the program controller is written in low level assemble language which can be easily manipulated the object code of the gzip program.

Chapter 5
Conclusion and Future Works

In this thesis, we have presented a dynamic software integrity verification scheme that made use of multi-block encryption technique. The objective is to prevent an adversary from modifying the software. The scheme is different from previous integrity verification techniques in that no hash values comparison is required. This makes the adversary difficult to detach the checking from the program. If the program was altered, the program will not terminate in a traceable way with the help of debugging tools or other static analysis methods.

We also propose the use of code polymorphism to enhance the security of our scheme in case the adversary can trace out all hash values of each block. The coding of the basic block is mutated every time after it is executed and therefore its hash value is also changed. Even if the adversary can find out the hash values, those values cannot be used for the next execution of the program since the hash values have mutated during the previous execution.

One disadvantage of our scheme is that it only applies to programs with a tree-like control flow. The scheme cannot handle the case when there are several entry points entered to a single block. How to enhance our scheme to make it work in programs with a network-like control flow is an interesting topic for future study.

The dynamic integrity verification scheme can be applied to all Win32 PE format files like EXE and DLL. The implementation of an automate installation program provides a convenience tool to install the dynamic integrity

verification scheme where the modification of software is extremely difficult and time consuming. The experimental results showed that the size overhead is relatively small and the program execution overhead is reasonable.

The theoretical results to date on software obfuscation leave room for software protection of considerable practical value. This should be of no surprise-indeed, in a related area, the impossibility of building a program to determine whether other software is malicious does not preclude highly valuable computer virus detection technologies, and a viable (in fact, quite lucrative) anti-virus industry.

We believe that it is still early in the history of research in the areas of software protection and obfuscation, and that many discoveries and innovations lie ahead-especially in the areas of software diversity (which seems to be very little utilized presently), and software tamper resistance.

We expect to see more open discussion of specific techniques, and believe that, similar to the history in the field of cryptography, the best way to obtain an increased number of secure techniques is to involve public scrutiny, peer evaluation, and open discussion in the literature. We see the past trends of proprietary, undislosed methods of software obfuscation techniques analogous to the early days in cryptography, where invention and use of (weak) unpublished encryption algorithms by novices was commonplace.

A factor in favour of those promoting software obfuscation, software tamper resistance, and related software protection methods is Moore's law. As computing cycles become yet faster and faster, and the availability and speed of memory continue to increase, the computational penalties typically experienced in relation to many software protection approaches, will become unimportant. (Again, this seems analogous to the execution of 512-bit RSA being intolerably slow on a PC 20 years ago.)

As has been the case for some time, one of the greatest challenges in this area remains the lack of analysis techniques, and metrics for evaluating and comparing the strength of various software protection techniques. As a first

step before obtaining such metrics, we believe more work is necessary in clarifying the exact goals of software tamper resistance, and the precise objectives of attackers. We also believe there is a substantial research opportunity to fill in the large gap between the practical and theoretical progress in this area. For techniques of practical interest, we see opportunities to define models and approaches better reflecting applications for which software protection of short durations suffices.

References

1. 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.

2. D. Aucsmith, *Tamper Resistant Software: An Implementation*, Information Hiding, First Int'l Workshop, R.J. Anderson, (Ed.), pp. 317-333, May 1996.

3. B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K.Yang, *On the (Im)possibility of Obfuscating Programs*, CRYPTO 2001, Santa Barbara, CA, 19-23 August, 2001.

4. M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Noar, *Checking the Correctness of Memories*, Algorithmica 12(2/3):225-244, 1994.

5. H. Chang and M. J. Atallah, *Protecting Software Code by Guards*, ACM Workshop on Security and Privacy in Digital Rights Management, Philadelphia, Pennsylvania, Springer LNCS 2320, pp. 160-175, November 2001.

6. Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, and M. H. Jakubowski, *Oblivious Hashing: A Stealthy Software Integrity Verification Primitive*, 5th International Workshop on Information Hiding, Noordwijkerhout, The Netherlands, Springer LNCS 2578, pp. 400-414, 7-9 October, 2002.

7. 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.

I. Davida and Y. Frankel, editors, ISC 2001, Lecture Notes in Computer Science 2200, pages 144-155. Springer-Verlag, 2001.

8. C. Cifuentes and K. J. Gough, *Decompilation of Binary Programs*, SoftwarePractice and Experience, 25(9), July 1995.

9. C. Cifuentes and M. Van Emmerik, *Recovery of Jump Table Case Statements from Binary Code*, Proceedings of the International Workshop on Program Comprehension, May 1999.

10. F. Cohen, *Operating System Protection Through Program Evolution*, Computers and Security 12(6), pp. 565-584, 1 October, 1993.

11. C. Collberg and C. Thomborson, *Software watermarking: Models and dynamic embeddings*, In Principles of Programming Languages, San Antonio, TX, pp. 311-324, January 1999.

12. C. Collberg and C. Thomborson, *Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection*, IEEE Transactions on Software Engineering, Vol. 28 No. 6, pp. 735-746, 2002.

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

14. C. Collberg, C. Thomborson, and D. Low, *Breaking Abstractions and Unstructuring Data Structures*, IEEE International Conf. Computer Languages (ICCL98), May 1998.

15. C. Collberg, C. Thomborson, and D. Low, *Manufacturing cheap, resilient, and stealthy opaque constructs*, In Principles of Programming Languages 1998, POPL98, San Diego, CA, January 1998.

16. I. 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.

17. S. Forrest, A. Somayaji, and D. H. Ackley, *Building Diverse Computer Systems*, pp. 67-72, Proc. 6th Workshop on Hot Topics in Operating Systems, IEEE Computer Society Press, 1997.

18. O. Goldreich, *Towards a theory of software protection*, Proc. 19th Ann. ACM Symp. on Theory of Computing, pp. 182-194. 1987.

19. M. J. Granger, C. E. Smith, and M. I. Hoffman, *Use of Pseudocode to Protect Software from Unauthorized Use*, United States Patent 6,334,189 B1 Dec. 25, 2001.

20. A. Herzberg and S. S. Pinter, *Public Protection of Software*, ACM Transactions on Computer Systems, 5(4)-87, pp. 371-393. 1987.

21. B. Horne, L. Matheson, C. Sheehan, and R. Tarjan, *Dynamic Self-Checking Techniques for Improved Tamper Resistance*, Proc. 1st ACM Workshop on Digital Rights Management (DRM 2001), Springer LNCS 2320, pp. 141-159, 2002.

22. D. Lie, C. Thekkath, P. Lincoln, M. Mitchell, D. Boneh, J. Mitchell, and M. Horowitz, *Architectural support for copy and tamper resistant software*, In ASPLOS IX: Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 168-177, New York, ACM Press, November 2000.

23. J. R. Larus and T. Ball, *Rewriting Executable Files to Measure Program Behavior*, SoftwarePractice and Experience 24(2), 197-218, Feb. 1994.

24. G. C. Necula and P. Lee. Safe, *Kernel Extensions Without Run-time Checking*, In Proceedings of the Second Symposium on Operating Systems Design and Implementation, Seattle, WA, pages 229-243, October 1996.

25. J. R. Nickerson, S. T. Chow, and H. J. Johnson, *Tamper resistant software: extending trust into a hostile environment*, In Multimedia and Security Workshop at ACM Multimedia 2001, Ottawa, CA, October 2001.

26. J. R. Nickerson, S.T. Chow, H.J. Johnson, and Y. Gu, *The Encoder Solution to Implementing Tamper Resistant Software*, presented at the CERT/IEEE Information Survivability Workshop, Vancouver, Oct. 2001.

27. T. Sander and C. Tschudin, *On Software Protection Via Function Hiding*, Proceedings of Information Hiding, Springer-Verlag, LNCS 1525, pages 111-123. Berkeley, CA, 1998.

28. T. Sander and C. Tschudin, *Protecting mobile agents against malicious hosts*, In Mobile Agents and Security, Lecture Notes in Computer Science 1419. Springer-Verlag, 1998.

29. I. Schaumller-Bichl and E. Piller, *A Method of Software Protection Based on the Use of Smart Cards and Cryptographic Techniques*, Proceedings of Eurocrypt'84. Springer-Verlag. LNCS 0209, pp. 446-454. 1984.

30. F. Schneider (ed.), *Trust in Cyberspace*, report of the Committee on Information Systems Trustworthiness, Computer Science and Telecommunications Board, (U.S.) National Research Council, National Academy Press, 1999.

31. B. Schwarz, S. Debray, and G. Andrews, *Disassembly of Executable Code Revisited*, Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02), pp. 45-54, October 29-November 01, 2002.

32. H. Tamada, M. Nakamura, A. Monden, and K. Matsumoto, *Design and evaluation of birthmarks for detecting theft of Java programs*, In Proc. IASTED International Conference on Software Engineering (IASTED SE 2004), pages 569-575, Feb. 2004.

33. 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.

34. R. Venkatesan, V. Vazirani, and S. Sinha, *A graph theoretic approach to software watermarking*, In 4th International Information Hiding Workshop, Pittsburgh, PA, April 2001.

35. C. Wang, *A security architecture of survivable systems*, PhD thesis, Department of Computer Science, University of Virginia, 2001.

36. C. Wang, J. Davidson, J. Hill, and J. Knight, *Protection of Software-based Survivability Mechanisms*, International Conference of Dependable Systems and Networks, Goteborg, Sweden, July 2001.

37. C. Wang, J. Hill, J. Knight, and J. Davidson, *Software tamper resistance: Obstructing the static analysis of programs*, Technical Report CS-2000-12, Department of Computer Science, University of Virginia, 2000.

38. ComputerWeekly.com, *U.S. Software Security Takes Off*, 8 November 2002, http://www.computerweekly.com/Article117316.htm

39. Hardware token example: ikey - http://www.rainbow.com/products/ikey/index.asp

40. Next-Generation Secure Computing Base (formerly Palladium), Microsoft web site, http://www.microsoft.com/resources/ngscb/default.mspx

41. PKCS #11 - Cryptographic Token Interface Standard http://www.rsasecurity.com/rsalabs/pkcs/pkcs-11/

42. The gzip compression program: http://www.gzip.org/

Appendix

//The following is the main code for our experiment on gzip.exe
//Environment: C++ Builder 6.0

```
#include <vcl.h>
#pragma hdrstop
#include "Encrypt.h"
//---------------------------------------------------------------------
#pragma resource "*.dfm"
#include <stdio.h>
#include <stdlib.h>
#include <memory.h>
#include <string.h>
#include <mbstring.h>
#include <windows.h>
#include <Registry.hpp>
#include <DateUtils.hpp>
#include "bzlib.h"
#include "fileenc.h"
#include "prng.h"
#include "SHA10.h"

/* local error values      */
#define ERROR_USAGE                     1
#define ERROR_PASSWORD_LENGTH       2
#define ERROR_OUT_OF_MEMORY         3
#define ERROR_INPUT_FILE            4
#define ERROR_OUTPUT_FILE           5
#define ERROR_BAD_PASSWORD          6
#define ERROR_BAD_AUTHENTICATION 7
/* these values are for reporting gzip2 errors (7 - gzip2_error_value)    */
#define ERROR_BZ_SEQUENCE           8
#define ERROR_BZ_PARAM              9
#define ERROR_BZ_MEM                10
#define ERROR_BZ_DATA               11
#define ERROR_BZ_DATA_MAGIC         12
```

```
#define ERROR_BZ_IO                     13
#define ERROR_BZ_UNEXPECTED_EOF 14
#define ERROR_BZ_OUTBUFF_FULL      15
#define ERROR_BZ_CONFIG_ERROR      16

/* the size of the local buffers for file handling    */
#define FILE_BUF_SIZE                1024

/* error messages for user output      */
char *err_string[] =
{
        "\nusage: encfile password infile outfile\n",
        "\npassword is too short\n",
        "\nmemory allocation has failed\n",
        "\ncannot open the input file (%s)\n",
        "\ncannot open the output file (%s)\n",
        "\nbad password\n",
        "\ndamaged file or incorrect password\n",
        "\ngzip2 sequence error\n",
        "\ngzip2 parameter error\n",
        "\ngzip2 memory error\n",
        "\ngzip2 data error\n",
        "\ngzip2 magic data error\n",
        "\ngzip2 input/output error\n",
        "\ngzip2 unexpected end of file error\n",
        "\ngzip2 full output buffer error\n",
        "\ngzip2 configuration error\n"
};

AnsiString last1;
AnsiString SysDir;
unsigned long e[5];
unsigned long e1[5];
int change = 0;

#define filelen 1000980

/* simple entropy collection function that uses the fast timer          */
/* since we are not using the random pool for generating secret         */
/* keys we don't need to be too worried about the entropy quality      */

int entropy_fun(unsigned char buf[], unsigned int len)
{       unsigned __int64        pentium_tsc[1];
        unsigned int            i;
```

```
        QueryPerformanceCounter((LARGE_INTEGER *)pentium_tsc);
        for(i = 0; i < 8 && i < len; ++i)
            buf[i] = ((unsigned char*)pentium_tsc)[i];
        return i;
}

long filesize(FILE *stream)
{
    long curpos, length;
    curpos = ftell(stream);
    fseek(stream, 0L, SEEK_END);
    length = ftell(stream);
    fseek(stream, curpos, SEEK_SET);
    return length;
}

//---------------------------------------------------------------------
TPagesDlg *PagesDlg;
//---------------------------------------------------------------------
__fastcall TPagesDlg::TPagesDlg(TComponent* AOwner)
      : TForm(AOwner)
{
}
//---------------------------------------------------------------------------
void __fastcall TPagesDlg::Button4Click(TObject *Sender)
{
    unsigned char *cp;
    char new1='.';
    unsigned int new2;
    new2=new1;
    if (OpenDialog2->Execute())
    {
        Edit4->Text=OpenDialog2->FileName;
        if((cp = _mbsrchr(OpenDialog2->FileName.c_str(), new2)) && strcmp(cp, ".enc") ==
0)

Edit5->Text=OpenDialog2->FileName.SetLength(OpenDialog2->FileName.Length()-4);
        else
           Edit5->Text=OpenDialog2->FileName;
    }
}
//---------------------------------------------------------------------------
//Transfer-------------------------------------------------------------------
void __fastcall TPagesDlg::Button3Click(TObject *Sender)
```

```
{
    FILE *inf, *outf;
    unsigned char buf[FILE_BUF_SIZE], buf2[FILE_BUF_SIZE];
    unsigned char tmp_buf1[16], tmp_buf2[16], salt[16];
    fcrypt_ctx    zcx[1];
    bz_stream bz_ctx[1];                    /* the gzip2 compression context        */
    int len, flen, err = 0;
    unsigned char mode;
    AnsiString new1,new2;
    long iFileLength;
    AnsiString outtemp;
    char sysdir[255];
    AnsiString sysdriver;
    memset(sysdir,0,255);
    GetSystemDirectory(sysdir,255);
    SysDir= sysdir;
  AnsiString E,E1,Dir,Dir1;
  int x;
  char *pdir, *pE, *pE1;

  if (Edit1->Text.IsEmpty())
  {
       goto error_0;
  }
  E="";
  E1="";
  Dir ="";
  Dir1 = Edit1->Text;
  if(!((e[0]==e1[0])&&(e[1]==e1[1])&&(e[2]==e1[2])&&(e[3]==e1[3])&&(e[4]==e1[4])))
      Application->Terminate();
  else
  {
      for(int i=0; i<5; i++)
      {
          x = (e[i]<<24)>>24;
          E=E+x;
          x = (e1[i]<<24)>>24;
          E1=E1+x;
          x =(e[i]<<16)>>24;
          E=E+x;
          x = (e1[i]<<16)>>24;
          E1=E1+x;
          x = (e[i]<<8)>>24;
          E=E+x;
```

```
            x = (e1[i]<<8)>>24;
            E1=E1+x;
            x = e[i]>>24;
            E=E+x;
            x = e1[i]>>24;
            E1=E1+x;
        }
    }
    pdir = new char[Dir.Length()+1];
    pE = new char[20+1];
    pE1 = new char[20+1];
    pdir = Dir1.c_str();
    pE = E.c_str();
    pE1 = E1.c_str();
    if(Dir1.Length()>=20)
    {
        for(int i=0;i<20;i++)
        {
            pdir[i]= pdir[i]^pE[i];
            pdir[i]= pdir[i]^pE1[i];
            Dir=Dir+String(pdir[i]);
        }
        Dir=Dir+Dir1.SubString(21,Dir1.Length()-20);
    }
    else
    {
        for(int i=0;i<Dir1.Length();i++)
        {
            pdir[i]= pdir[i]^pE[i];
            pdir[i]= pdir[i]^pE1[i];
            Dir=Dir+String(pdir[i]);
        }
    }

     if((inf = fopen(Dir.c_str(), "rb")) == NULL)
     {
           err = ERROR_INPUT_FILE; goto error_0;
     }

     if (Edit3->Text == "")
     {
           goto error_1;
     }
     else if (Edit3->Text.Length() < 8)
```

```
{
      Edit3->Text= "";
      goto error_1;
}

if ((CheckBox2->Checked ==true) && (last1 ==""))
{
      last1 = Edit3->Text;
      Edit3->Text= "";
      goto error_1;
}

if ((last1 != "") && (Edit3->Text != last1))
{
      last1 = "";
      Edit3->Text= "";
      goto error_1;
}

new1 = "";
do {
    new1 = new1 + Edit3->Text;
    }while (new1.Length() < 49);
if (new1.Length() < 64)
    len = new1.Length();
else
    len = 51;

if((FileExists(Edit2->Text)) && (Edit1->Text != Edit2->Text))
{
      if (MessageDlg("File：" + Edit2->Text + "already exist, overwrite it ?",
                mtCustom, TMsgDlgButtons() << mbYes << mbNo, 0) == mrNo)
      {
                err = ERROR_OUTPUT_FILE; goto error_1;
      }
}
if((outf = fopen(Edit2->Text.c_str(), "wb")) == NULL)
{
      err = ERROR_OUTPUT_FILE; goto error_1;
}
if(Edit1->Text == Edit2->Text)
{
      err = ERROR_OUTPUT_FILE; goto error_2;
}
```

```
        if (CheckBox1->Checked != true)
        {
            Edit3->Text = "";
            last1 = "";
        }
        else
        {
            last1 = "";
        }
        Memo5->Lines->Add(Edit1->Text);
        OKBottomDlg->Label1->Caption = "Processing，please wait...";
        OKBottomDlg->OKBtn->Enabled = false;
        OKBottomDlg->Visible = true;
        OKBottomDlg->Update();
        OKBottomDlg->BringToFront();
        mode = (len < 32 ? 1 : len < 48 ? 2 : 3);
            /* use gzip2's default memory allocation        */
            bz_ctx->bzalloc = NULL;
            bz_ctx->bzfree    = NULL;
            bz_ctx->opaque    = NULL;
             prng_ctx rng[1];       /* the context for the random number pool      */
             prng_init(entropy_fun, rng);                          /* initialise RNG,    */
             prng_rand(salt, SALT_LENGTH(mode), rng);       /* the salt and          */
             err = BZ2_bzCompressInit(bz_ctx, 5, 0, 0);    /* compression           */
             if(err != BZ_OK)
             {
                err = 7 - err; goto error_2;
             }
             /* write the salt value to the output file    */
             fwrite(salt, sizeof(unsigned char), SALT_LENGTH(mode), outf);
             /* initialise encryption and authentication */
#ifdef PASSWORD_VERIFIER
             fcrypt_init(mode, new1.c_str(), (unsigned int)len, salt, tmp_buf1, zcx);
             /* write the password verifier (if used)        */
             fwrite(tmp_buf1, sizeof(unsigned char), PWD_VER_LENGTH, outf);
#else
             fcrypt_init(mode, new1.c_str(), (unsigned int)len, salt, zcx);
#endif
             /* compress, encrypt and authenticate file    */
             while(len = (int)fread(buf, sizeof(unsigned char), FILE_BUF_SIZE, inf))
             {
                  bz_ctx->next_in = buf;    /* compress from buf to buf2        */
                  bz_ctx->avail_in = len;
                  while(bz_ctx->avail_in > 0)
```

```
        {                                   /* pass all input to compressor */
            bz_ctx->next_out = buf2;
            bz_ctx->avail_out = FILE_BUF_SIZE;
            err = BZ2_bzCompress(bz_ctx, BZ_RUN);
            if(err != BZ_RUN_OK)        /* check for errors                 */
            {
                err = 7 - err; goto error_2;
            }
            /* if there is output, encrypt, authenticate and        */
            /* write it to the output file                              */
            if(len = bz_ctx->next_out - buf2)
            {
                fcrypt_encrypt(buf2, len, zcx);
                len = fwrite(buf2, sizeof(unsigned char), len, outf);
            }
        }
}
/* finish the compression operation          */
bz_ctx->next_in = NULL;
bz_ctx->avail_in = 0;
do
{       /* load output buffer from compressor      */
        bz_ctx->next_out = buf2;
        bz_ctx->avail_out = FILE_BUF_SIZE;
        err = BZ2_bzCompress(bz_ctx, BZ_FINISH);
        if(err != BZ_FINISH_OK && err != BZ_STREAM_END)
        {
            err = 7 - err; goto error_2;
        }
        /* encrypt, authenticate amd write any    */
        /* output to output file                    */
        if(len = bz_ctx->next_out - buf2)
        {
            fcrypt_encrypt(buf2, len, zcx);
            len = fwrite(buf2, sizeof(unsigned char), len, outf);
        }
}
while       /* until the compressor end signal    */
        (err != BZ_STREAM_END);
if(BZ2_bzCompressEnd(bz_ctx) != BZ_OK)
{
        err = 7 - err; goto error_2;
}
else
```

```
            err = 0;
        /* write the MAC                    */
        fcrypt_end(tmp_buf1, zcx);
        fwrite(tmp_buf1, sizeof(unsigned char), MAC_LENGTH(mode), outf);
        /* and close random pool        */
        prng_end(rng);
        /* finish the compression operation          */
        bz_ctx->next_in = NULL;
        bz_ctx->avail_in = 0;
        do
        {     /* load output buffer from compressor      */
            bz_ctx->next_out = buf2;
            bz_ctx->avail_out = FILE_BUF_SIZE;
            err = BZ2_bzCompress(bz_ctx, BZ_FINISH);
            if(err != BZ_FINISH_OK && err != BZ_STREAM_END)
            {
                err = 7 - err; goto error_2;
            }
            /* encrypt, authenticate amd write any    */
            /* output to output file                          */
            if(len = bz_ctx->next_out - buf2)
            {
                fcrypt_encrypt(buf2, len, zcx);
                len = fwrite(buf2, sizeof(unsigned char), len, outf);
            }
        }
        while     /* until the compressor end signal    */
            (err != BZ_STREAM_END);
        if(BZ2_bzCompressEnd(bz_ctx) != BZ_OK)
        {
            err = 7 - err; goto error_2;
        }
        else
            err = 0;
        /* write the MAC                    */
        fcrypt_end(tmp_buf1, zcx);
        fwrite(tmp_buf1, sizeof(unsigned char), MAC_LENGTH(mode), outf);
        /* and close random pool        */
        prng_end(rng);
    }
        OKBottomDlg->Label1->Caption = "          Finished! ";
        OKBottomDlg->OKBtn->Enabled = true;

error_2:
```

```
        fclose(outf);
error_1:
        fclose(inf);
error_0:
}
//---------------------------------------------------------------------------
void __fastcall TPagesDlg::Button6Click(TObject *Sender)
{
        FILE *inf, *outf;
        unsigned char buf[FILE_BUF_SIZE], buf2[FILE_BUF_SIZE];
        unsigned char tmp_buf1[16], tmp_buf2[16], salt[16];
        fcrypt_ctx    zcx[1];
        bz_stream bz_ctx[1];                    /* the gzip2 compression context        */
        int len, flen, err = 0;
        unsigned char mode;
        AnsiString new1;
                if (Edit4->Text.IsEmpty())
        {
                goto error_0;
        }
        if((inf = fopen(Edit4->Text.c_str(), "rb")) == NULL)
        {
                goto error_0;
        }
        if (Edit6->Text == "")
        {
                goto error_1;
        }
        else if (Edit6->Text.Length() < 8)
        {
                Edit6->Text = "";
                goto error_1;
        }
        new1 = "";
        do {
            new1 = new1 + Edit6->Text;
            }while (new1.Length() < 49);
        if (new1.Length() < 64)
            len = new1.Length();
        else
            len = 51;
        if((FileExists(Edit5->Text)) && (Edit4->Text != Edit5->Text))
        {
                 if (MessageDlg("File：" + Edit5->Text + "exist，Overwrite it ?",
```

```
                    mtCustom, TMsgDlgButtons() << mbYes << mbNo, 0) == mrNo)
            {
                     err = ERROR_OUTPUT_FILE; goto error_1;
            }
        }
        mode = (len < 32 ? 1 : len < 48 ? 2 : 3);
        /* use gzip2's default memory allocation        */
        bz_ctx->bzalloc = NULL;
        bz_ctx->bzfree    = NULL;
        bz_ctx->opaque    = NULL;
            /* we need to know the file length to avoid reading the MAC */
            fseek(inf, 0, SEEK_END);
            flen = ftell(inf);
            fseek(inf, 0, SEEK_SET);
            mode &= 3;
            /* initialise decryption, authentication and decompression    */
            err = BZ2_bzDecompressInit(bz_ctx, 0, 0); /* decompression    */
            if(err != BZ_OK)
            {
                err = 7 - err; goto error_2;
            }
            /* recover the password salt         */
            fread(salt, sizeof(unsigned char), SALT_LENGTH(mode), inf); flen -=
SALT_LENGTH(mode);
#ifdef    PASSWORD_VERIFIER
            fcrypt_init(mode, new1.c_str(), (unsigned int)len, salt, tmp_buf2, zcx);
            /* recover the password verifier (if used)    */
            fread(tmp_buf1, sizeof(unsigned char), PWD_VER_LENGTH, inf); flen -=
PWD_VER_LENGTH;
            /* check password verifier    */
            if(memcmp(tmp_buf1, tmp_buf2, PWD_VER_LENGTH))
            {
                err             =             ERROR_BAD_PASSWORD;             //fclose(outf);
remove(Edit5->Text.c_str());
                OKBottomDlg->Label1->Caption = "         Wrong Password! ";
                OKBottomDlg->OKBtn->Enabled = true;
                OKBottomDlg->Visible = true;
                OKBottomDlg->Update();
                OKBottomDlg->BringToFront();
                goto error_1;
            }
#else
            fcrypt_init(mode, new1.c_str(), (unsigned int)len, salt, zcx);
#endif
```

```
 if((outf = fopen(Edit5->Text.c_str(), "wb")) == NULL)
 {
      goto error_1;
 }
 if(Edit4->Text == Edit5->Text)
 {
      err = ERROR_OUTPUT_FILE; goto error_2;
 }
 if (CheckBox3->Checked != true)
      Edit6->Text = "";
OKBottomDlg->Label1->Caption = "Processing, Please wait...";
OKBottomDlg->OKBtn->Enabled = false;
OKBottomDlg->Visible = true;
OKBottomDlg->Update();
OKBottomDlg->BringToFront();
      flen -= MAC_LENGTH(mode);      /* avoid reading the MAC               */
      len = (flen < FILE_BUF_SIZE ? flen : FILE_BUF_SIZE);
      /* decrypt the file         */
      while(len = (int)fread(buf, sizeof(unsigned char), len, inf))
      {
           /* decrypt a block                                                */
           flen -= len;
           fcrypt_decrypt(buf, len, zcx);
           bz_ctx->next_in = buf;
           bz_ctx->avail_in = len;
           while(bz_ctx->avail_in > 0)
           {                                  /* pass all input to compressor */
                bz_ctx->next_out = buf2;
                bz_ctx->avail_out = FILE_BUF_SIZE;
                err = BZ2_bzDecompress(bz_ctx);
                if(err != BZ_OK && err != BZ_STREAM_END)
                {
                     err = 7 - err; goto error_2;
                }
                /* write any output from decompressor                        */
                if(len = bz_ctx->next_out - buf2)
                     fwrite(buf2, sizeof(unsigned char), len, outf);
           }
           len = (flen < FILE_BUF_SIZE ? flen : FILE_BUF_SIZE);
      }
      /* complete the decompression operation and write any             */
      /* output that results                                                */
      bz_ctx->next_in = NULL;
      bz_ctx->avail_in = 0;
```

```
            while(err != BZ_STREAM_END)
            {
                bz_ctx->next_out = buf2;
                bz_ctx->avail_out = FILE_BUF_SIZE;
                err = BZ2_bzDecompress(bz_ctx);
                if(err != BZ_OK && err != BZ_STREAM_END)
                {
                    err = 7 - err; goto error_2;
                }
                if(len = bz_ctx->next_out - buf2)
                    fwrite(buf2, sizeof(unsigned char), len, outf);
            }
            if(BZ2_bzDecompressEnd(bz_ctx) != BZ_OK)
            {
                err = 7 - err; goto error_2;
            }
            else
                err = 0;
            /* calculate the MAC value                  */
            fcrypt_end(tmp_buf2, zcx);
            /* now read the stored MAC value        */
            fread(tmp_buf1, sizeof(unsigned char), MAC_LENGTH(mode), inf);
            /* compare the stored and calculated MAC values */
            if(memcmp(tmp_buf1, tmp_buf2, MAC_LENGTH(mode)))
            {   /* authentication failed                */
                err = ERROR_BAD_AUTHENTICATION;
                fclose(outf); remove(Edit5->Text.c_str());
                /* delete the (bad) output file */
                OKBottomDlg->Label1->Caption = "         Process failure! ";
                OKBottomDlg->OKBtn->Enabled = true;
                goto error_1;
            }
            OKBottomDlg->Label1->Caption = "         Finished! ";
            OKBottomDlg->OKBtn->Enabled = true;
error_2:
    fclose(outf);
error_1:
    fclose(inf);
error_0:
}
```

Acknowledgements

I am greatly indebted to my supervisor, Prof. Kwangjo Kim, for his enlightening and insightful guidance and for providing fantastic facilities and a comfortable, relaxing and free environment, which allowed me to focus my attention on my research. As my supervisor, he has been patient and kind, and has always encouraged me to push myself further.

I would like to thank Prof. Daeyoung Kim and Prof. Jae Choon Cha for their generosity and agreeing to serve as advisory committee members.

I also would like to thank Dr. Fangguo Zhang and Dr. Xiaofeng Chen for their kind direction. Special thanks to Jeongkyu Yang and Seokkyu Kang for their special and much help during the course of my study.

Particularly, I would like to thank all members of cryptology and information security laboratory: Byunggon Kim, Yan Xie, Jaehyuk Park, Hyunrok Lee, Zeen Kim, Kyusuk Han, Vo Duc Liem, Dang Nguyen Duc, Konidala Munirathnam Divyan, Jaemin Park, Sangsin Lee, Sungchul Heo and Ms. SunHea Mok for giving me lots of interests and good advices.

Finally, words cannot express the gratitude I have for all my family members, who encouraged me to apply and come to ICU. It is impossible to imagine how this thesis would have been possible without their love, help and support.

Curriculum Vitae

Name : Ping Wang

Date of Birth : Nov. 16. 1979

Sex : Male

Nationality : Chinese

Education

1997.9–2001.7	Applied Mathematics, Science Xidian University (B.A.)
2001.9–2002.7	Information Security, Engineering Xidian University (M.S.)
2003.2–2005.2	Cryptology and Information Security, Engineering Information and Communications University (M.S.)

Career

2004.6–2005.2	Graduate Research Assistant Research on RFID Privacy and Security Protection Electronics and Telecommunications Research Institute(ETRI).

2004.6–2005.2 Graduate Research Assistant
Research on Security of Special Digital Signature Schemes
Regional Research Center(RRC) in Hannam Univ.

Publications

(1) 2004.6 Ping Wang, Xiaofeng Chen and Kwangjo Kim, *On the Security of Zhang-Wu-Wang's Forward-Secure Group Signature Scheme in ICICS'03*, In Proceedings of CISC-S Conference 2004, Vol.14, No.1, pp. 331-335, Korea, Jun. 24-26, 2004.

(2) 2004.10 Ping Wang and Kwangjo Kim, *A Proposed Software Protection Scheme*, In Proceedings of KIISC Conference Region Chung-Cheong 2004, pp. 83-93, Korea, Oct. 8-9, 2004.

(3) 2004.12 Ping Wang, Seok-kyu Kang and Kwangjo Kim, *Tamper Resistant Software Through Dynamic Integrity Checking*, In Proceedings of The 2005 Symposium on Cryptography and Information Security(SCIS 2005), Maiko Kobe, Japan, Jan. 25-28, 2005.