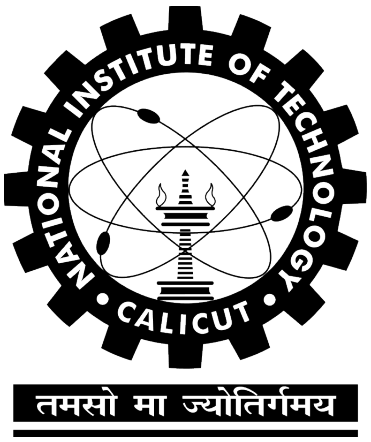
TEMPER RESISTANCE

A
Seminar Report

by

Manpal Singh
B130093CS



Department of Computer Science and Engineering
National Institute of Technology, Calicut
Monsoon-2016

National Institute of Technology, Calicut
Department of Computer Science and Engineering

Certified that this Seminar Report entitled

TEMPER RESISTANCE

is a bonafide record of the Seminar presented by

Manpal Singh
B130093CS

in partial fulfillment of
the requirements for the award of the degree of
Bachelor in Technology
in
Computer Science and Engineering

Srinivasa T M
(Group Seminar Coordinator)
Department of Computer Science and Engineering

Acknowledgement

It is my proud privilege and duty to acknowledge the kind of help and guidance received from several people in preparation of this report. It would not have been possible to prepare this report in this form without their valuable help, cooperation and guidance.

First and foremost, I wish to record my sincere gratitude to my coordinator Srinivasa TM, National Institute of Technology, Calicut for his constant support in preparation of this report.

The seminar on Temper Resistance was very helpful to us in giving the necessary background information and inspiration in choosing this topic for the seminar. My sincere thanks to Prof. Sumesh TA and Prof. Srinivasa TM Project/Seminar Coordinator for having supported the work related to this Seminar. Their contributions and technical support in preparing this report are greatly acknowledged.

Last but not the least, I wish to thank my parents for financing my studies in this college as ill as for constantly encouraging us to learn engineering. Their personal sacrifice in providing this opportunity to learn engineering is gratefully acknowledged

Abstract

To secure digital assets, a tasteful integration of a variety of technologies and processes is necessary. Tamper-resistance is being increasingly used as an important piece of a more comprehensive security system. It provides an eïňĂective barrier to entry and protects digital assets in the embedded systems from most people and most attacks. Embedded systems pose unique security challenges because they are largely used in relationships where one party wants to put a secure, embedded device, in the hands of another, with the assurance that the second party cannot modify and hack the device. First, it outline the major attacks that threaten the security of an embedded system. Tamper-resistant storage techniques provide varying degrees of authenticity and integrity for data. This paper surveys ïňĄve implemented tamper-resistant storage systems that use encryption, cryptographic hashes, digital signatures and error-correction primitives to provide varying levels of data protection.

Contents

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 dollar 29 billion. As soon as computers started to become 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) âĂŞ 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.

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 order to prevent tampering attacks, tamper-proofing 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. 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 as-

pects. 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.

2 Types of Attacks

2.1 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.

2.2 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.

2.3 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.

2.4 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.

2.5 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 mech-

anism detected the change.

3 Methods to Prevent Tempering

3.1 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 attempts to transform a program into an equivalent one that is more difficult to manipulate .One example is the Java byte code obfuscators. A major drawback of all obfuscation approaches is that they are of necessity adhoc. Unless provably effective techniques can be developed, each obfuscation is almost always immediately followed by countermeasures. In fact, Baraket al. 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, 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 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:

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

2- 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.

3- It maximizes stealth properties, i.e., P' should exhibit similar statistical properties, when compared to P.

4- 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. 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. Cohen suggested a approach in the early 1990s, employing obfuscation among other mechanisms as a defense against computer viruses. CohenâĂŹs 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 .Wang provides an important security result substantiating this general approach. The idea involves incorporating program transformations to exploit the hardness of precise inter procedural static analysis in the presence of aliased variables, combined with transformations degenerating program flow control. Collberg 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 met-

rics 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).

3.2 Tamper Resistance

Software obfuscation provides protection against reverse engineering, 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. Aucsmith defines tamper resistant software as software which is resistant to observation 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. 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 attackerâĂŹs difficulty of disabling testers.

4 Integrity Verification

In the last few years, there have been active development of software obfuscation and watermarking 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. 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. 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. 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 function. In order to detect run-time attacks, Chen and Venkatesan proposed oblivious hashing 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 proposed another method that enhances the runtime 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 discussed an innovative idea that suggests encrypting the executable, thereby preventing anyone from modifying it successfully unless the adversary is able to decrypt it. 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.

5 Proposed Scheme

We propose the use of multi-blocking encryption for the integrity verification of software.
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.

5.1 Design Objective

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.

5.2 Proposed Multi-blocking Encryption

Based on the concept of multi-blocking encryption, 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 independent. 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.

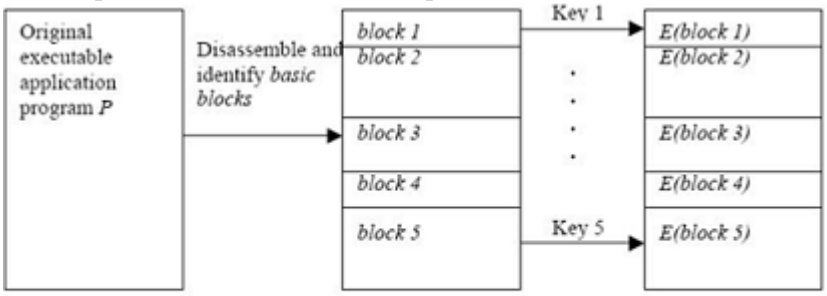


Figure 3.1: Different sized blocks encrypted by different keys

In general, it is convenient and feasible to store those keys inside the hardware token 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.

5.3 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.

5.4 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.

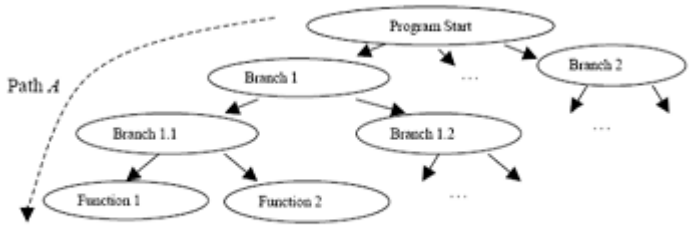


Figure 3.2: Tree structure of the program

Let the program be P and we consider any single path A .Let the path be broken down in n basic blocks,b1,b2âĂę,bn such that the control flow of the path A starts at b1 followed by b2 and then b3, etc. The blocks are in encrypted form except the starting block b1. 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

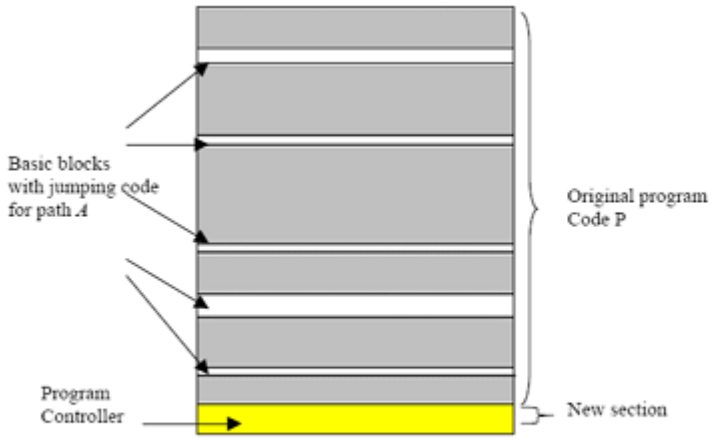


Figure 3.3: Structure of protected program

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 bi, 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 Derive Key API function. C Derive Key can derive a secret key from a known data, Hi 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 b2 starts to execute, the program jumps to the program controller to calculate H1 = Hash(b1), where b1 is in plaintext.
2. The secret key K1 is then derived from H1, e.g., K1 = H1.
3. The second block E(b2) is decrypted by K1 : b2 = DK1(E(b2)).
4. The decrypted block is moved to its original place, followed by the execution of the decrypted codes.

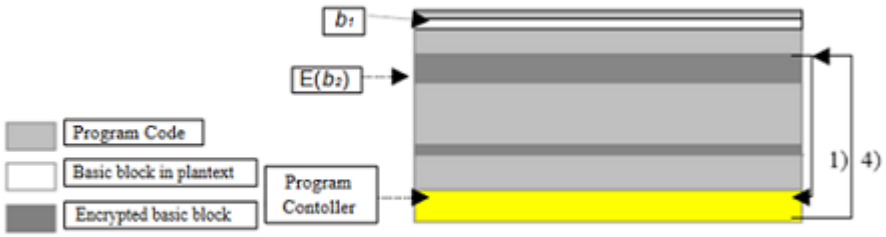


Figure 3.4: Program Progress 1 - 4

5. Before b3 starts to execute, the program jumps to the program controller to calculate H2 = Hash(b2).
6. The secret key K2 is derived from H2, e.g., K2 = H2.
7. The third block E(b3) is decrypted by K2 : b3 = DK2(E(b3)).
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 key'

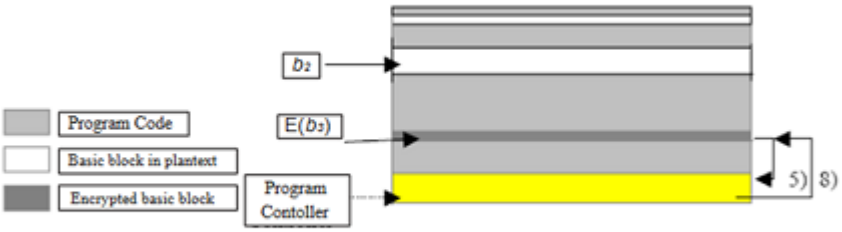


Figure 3.5: Program Progress 5 - 8

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 bi which will produce a different hash value Hi. Before the execution of bi+1, the hash value Hi

is not the proper decryption key Ki for block bi+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.

5.5 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.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 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 bi

after it has been executed and thereby changing its hash value. We can use the new hash value to re-encrypt the next block bi+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.

5.6 Comparison with Previous Schemes

Recently, there have been active development on software dynamic integrity verification. To our knowledge, Aucsmith 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 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 âĂŸstorage of keysâĂŹ 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, 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 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.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

6 Reference

1. D. Aucsmith, Tamper Resistant Software: An Implementation Information Hiding, First IntâĂŹl Workshop, R.J. Anderson, (Ed.), pp. 317-333, May 1996.
2. https://en.wikipedia.org/wiki/Tamperresistance
3. http://data.jsa.or.jp/stdz/instac/committe/tamperresistance/TSRCreport061002:pdf
4. 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.