# Tamper Resistant Software by Integrity-Based Encryption

Jaewon Lee, Heeyoul Kim, and Hyunsoo Yoon

Division of Computer Science, Department of EECS, KAIST, Deajeon, Korea
{jaewon, hykim, hyoon}@camars.kaist.ac.kr

**Abstract.** There are many situations in which it is desirable to protect a piece of software from illegitimate tampering once it gets distributed to the users. Protecting the software code means some level of assurance that the program will execute as expected even if it encounters the illegitimated modifications. We provide the method of protecting software from unauthorized modification. One important technique is an *integrity-based encryption*, by which a program, while running, checks itself to verify that it has not been modified and conceals some privacy sensitive parts of program.

**Keywords:** Security, Tamper Resistant Software, Software Protection.

## 1 Introduction

A fundamental limitation faced by designers of e-commerce and network security application is that software is easy to reverse engineer in order to determine how the software works and discover embedded secrets and intellectual properties and to make unauthorized changes for the functionality of the software. Therefore, unprotected software deployed on suspicious hosts cannot be trusted by the other hosts even the server. This situation is namely called a *malicious host problem* [1]. This problem is central in the cases that the client programs are executed in arbitrary user environment, such as DRM (Digital Right Management), e-commerce, on-line game, etc.

Our goal for tamper resistant software is to defend up to the level of dynamic modification for program, and can be summarized as following objectives: 1) (**Confidentiality**) Privacy sensitive algorithms which have been implemented into executable binary code shall be protected so that they may be concealed from competitors of program producer or analysis of adversary and 2) (**Integrity**) The dynamic modification by its user shall be detected by program itself, and it helps to cope with some proper reaction.

To fulfill these goals, we propose an *integrity-based encryption* scheme, which is composed of self-decrypting and self-integrity checking methods. The self-decryption makes a program enable itself to decrypt own parts of encrypted code. The decryption key is extracted by hashing other parts of program, so we can also preserve the integrity of those parts. Furthermore, we offer interleaving and

K.-M. Liew et al. (Eds.): PDCAT 2004, LNCS 3320, pp. 608–612, 2004.
© Springer-Verlag Berlin Heidelberg 2004

mutual guarding mechanism to enhance the security. With our scheme, neither specialized compiler nor hardware is needed and additional implementation in source code is minimal, so it makes efficient to use in practice.

## 2 Related Work

Previous work to response malicious host problem can be classified into two categories as passive and active prevention. *Passive prevention* refers making software to be resistant against static analysis. Obfuscation [2, 3] is a major example and it attempts to thwart reverse engineering by making it hard to understand the behavior of a program. Also, software watermark and fingerprint [4, 5] allow tracking of misused program copies by providing an additional deterrent to tampering. However, it may be effective for the cases that decompilation of binary code produces some high level of human recognition such as Java. Also, it requires specialized compiler and may result in degradation of performance.

Secondly, *active prevention* protects the software from dynamic analysis, which uses debuggers or processor emulators [1, 6]. However, in an absolute sense, this type of prevention is impossible on the PC due to the characteristics of open architecture. Any defense against this type of attacks must, at best, merely deter a perpetrator by providing a poor return on their investment.

## 3 Our Approach

In our scheme, a program code is classified into three classes, i.e. algorithm private, integrity sensitive, and normal classes of code. Algorithm privacy is realized by the code encryption with guarding chain. It does not require an explicit decryption key while program running. Furthermore, decryption routines and integrity sensitive codes are protected by guarding chain, as shown in Fig. 1, to guarantee that those parts are not illegitimately modified by malicious user.

### 3.1 Notations and Assumptions

**Program Structural Notations**

$A||B$ : Concatenation of program fragments $A$ and $B$.
$q_i$ : The integrity sensitive class of program.
$Q_i,\ 1 \leq i \leq n$ : The concatenation of integrity sensitive fragments, i.e., $Q_i = q_{i_1}||\cdots||q_{i_m}$, for example in Fig. 1, $Q_1 = q_1,\ Q_2 = q_1||q_2,\ Q_3 = q_1||q_3||q_4$.
$P_i,\ 1 \leq i \leq n$ : The algorithm private class of program in plain form.
$C_i,\ 1 \leq i \leq n$ : The algorithm private class of program in encrypted form.
$D_i,\ 1 \leq i \leq n$ : The decryption routines to decrypt $C_i$.

**Cryptographic Notations**

$\mathcal{H}(m)$ : A one-way, collision-resistant hash function.
$Enc_k(m)$ : Symmetric encrypt function on message $m$ with the key $k$.

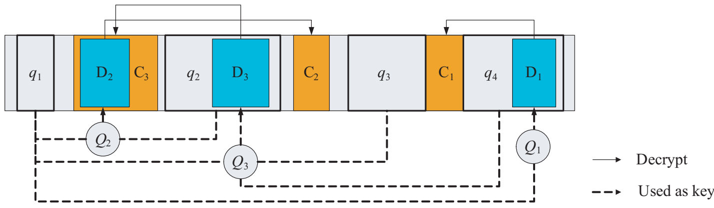


**Fig. 1.** Program image of mutual protecting cipher code

$Dec_k(c)$ : Symmetric decrypt function on ciphertext $c$ with the key $k$.
$C : A \rightarrow B$ : Replace $A$ with $B$ by $C$.

*Magic codes* are used to denote the beginning and ending positions of each $P_i$ and to store some meta data. But, they do not affect the program execution. $Q_1, \ldots,$ and $Q_n$ are the integrity sensitive parts of program, whose hash values are taken as the keys for the both of encryption of $P_i$ and decryption of $C_i$.

### 3.2 Code Encryption

To realize the algorithm privacy, we adopt a conventional symmetric encryption algorithm to encrypt privacy sensitive fragments of program, but use the hash value of other parts of program as key. The following steps are appended to the original flow of software development.

Step 1) Initialize : Complete the source code with insertion of magic codes
Step 2) Compile : Compile the source code with ordinary compiler.
Step 3) Encrypt : Encrypt the $P_i$ into $C_i$ using external encrypting utility.

$$\mathsf{Utility} : P_i \rightarrow C_i, \text{ where } C_i = Enc_{\mathcal{H}(Q_i)}(P_i), \ 1 \leq i \leq n \qquad (1)$$

### 3.3 Code Decryption and Integrity Checking

In the beginning of program execution, the encrypted segment, $C_i$, should be decrypted into executable code, $P_i$, as follows:

$$D_i : C_i \rightarrow Dec_{\mathcal{H}(Q_i)}(C_i), \ 1 \leq i \leq n \qquad (2)$$

If a $Q_i$ has been tampered into $Q_i'$, decryption key will be $\mathcal{H}(Q_i') \neq \mathcal{H}(Q_i)$, so $D_i$ cannot properly decrypt $C_i$.

## 4 Security Analysis

*Reverse engineering* is the process of analyzing a subject system 1) to identify the system's components and their interrelationships and 2) to create representations of the system in another form or at a higher level of abstraction [7]. It can be broadly classified into the static analysis and dynamic analysis, according to the attacker's aptitude for analysis.

Our goal for tamper resistant software is to defend against static analysis and dynamic analysis up to the level of dynamic substitution for the program instruction. The security of our scheme is supported by two points of view. The first is the integrity and secrecy of program and the second is the guarding chain.

### 4.1 Program Execution Integrity and Secrecy

Program execution integrity is guaranteed by the hashing used to generate decryption key. If the integrity sensitive code $Q_i$ is modified into $Q'_i$ by the perpetrator, it's hash value $\mathcal{H}(Q'_i)$ will not correspond to it's origin $\mathcal{H}(Q_i)$ which was used in encryption. After all, $C_i$ cannot be properly decrypted, therefore program execution will be obstructed. Furthermore, through the encryption of privacy sensitive part of program, algorithm privacy can be preserved against static analysis. Also, we can deter a perpetrator against dynamic analysis by providing a fine granularity of encrypted fragment.

### 4.2 Guarding Chain

As shown in Fig. 1, the chain of key usage can provide sophisticated protection scheme that is more resilient against attacks. For example, if an adversary wishes to modify some codes in $q_1$, $D_1, D_2$, and $D_3$ cannot properly decrypt $C_1, C_2$, and $C_3$, respectively. Also, $D_2$ is affected by the modification of $q_2$ and $D_3$ is affected by $q_3$ and $q_4$. In another sense, the corruption of $D_3$ by modification of $q_1, q_3$, or $q_4$ will cause the failure of decrypting $C_3$. It also affects decrypting $C_2$. In this manner, the adversary would have a difficulty to track down the dependency.

## 5 Conclusion

This paper presents and discusses the techniques for protecting software from the malicious users trying to reverse engineer and modify the code on their own purpose. The integrity-based encryption scheme is proposed to guarantee the code integrity and to promise algorithm privacy with minimal effort in development of software. Furthermore, the guarding chain would make the adversary more difficult to analyze the dependency of protection mechanism. Our scheme defend against static analysis for algorithm privacy and dynamic analysis for execution integrity.

## Acknowledgement

This work was supported by the Korea Science and Engineering Foundation (KOSEF) through the Advanced Information Technology Research Center(AITrc) and University IT Research Center (ITRC) Project.

## References

1. Sander, T., Tschudin, C.F.: Protecting Mobile Agents Against Malicious Hosts. In: Proceedings of Mobile Agents and Security, LNCS 1419. (1998) 44–60
2. Collberg, C., Thomborson, C., Low, D.: A Taxonomy of Obfuscating Transformations. Technical Report Technical Report 161, Department of Computer Science, The University of Auckland, New Zealand (1997)
3. Ogiso, T., Sakabe, Y., Soshi, M., Miyaji, A.: Software Obfuscation on a Theoretical Basis and Its Implementation. IEICE Trans. Fundamentals **E86-A** (2003) 176–186
4. Esparza, O., Fernandez, M., Soriano, M., Muñoz, J.L., Forné, J.: Mobile Agent Watermarking and Fingerprinting: Tracing Malicious Hosts. In: Proceedings of DEXA 2003, LNCS 2736. (2003) 927–936
5. Myles, G., Collberg, C.: Software Watermarking Through Register Allocation: Implementation, Analysis, and Attacks. In: Proceedings of 6th International Conference on Information Security and Cryptology -ICISC 2003. (2003) 274 – 293
6. Aucsmith, D.: Tamper Resistant Software: An Implementation. In: Proceedings of First International Workshop on Information Hiding, LNCS 1174. (1996) 317–333
7. Chikofsky, E.J., II, J.H.C.: Reverse Engineering and Design Recovey: A Taxonomy. IEEE Software **7** (1990) 13–17