NAIST-IS-DD0361009

Doctoral Dissertation

Protecting Secret Information in Software Processes and Products

Yuichiro Kanzaki

February 2, 2006

Department of Information Systems
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Yuichiro Kanzaki

Protecting Secret Information in Software Processes and Products *

Yuichiro Kanzaki

Abstract

Recently, many software development processes and products involve secret information, which is valuable or related to system security, such as the cipher keys of a digital rights management system, conditional branch instructions for license checking, and algorithms that are commercially valuable. The goal of this dissertation is to prevent such secret information from being revealed to users. Two cases are considered where secret information is revealed: (1) through work products (e.g., source code, specification) leaked by insiders (developers) who take part in a software development process, and (2) by reverse engineering of software products (i.e., software implementation).

First, in order to tackle (1), a method is proposed which is useful for constructing secure (i.e., little risk of leakage by insiders) software development processes. Knowledge about work products in a software process is transferred among developers in a person-to-person manner, and the probability of each developer knowing each product depends on the assignment of developers and the structure of the software process. Based on these facts, the mechanism of knowledge transfer within a software process for evaluating the risk of leaking security-sensitive products by insiders is formulated.

* Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0361009, February 2, 2006.

Second, in order to tackle (2), a method for increasing the cost of reverse engineering attacks is proposed. This method first overwrites program instructions with dummy instructions. Then, the program itself restores (i.e., replaces the dummy instructions with the original instructions) within a certain period of execution using a self-modification mechanism. This drastically increases the complexity of program understanding since the understanding fails if the dummy instructions are inspected as they are.

This dissertation is organized as follows. In Chapter 1, the background and outline of the dissertation is summarized. In Chapter 2, a method to evaluate the risk of leaking security-sensitive work products by insiders, for a given software process, is presented. In Chapter 3, a method for increasing the cost of reverse engineering attacks using the self-modification mechanism is presented. Chapter 4 concludes the dissertation and presents directions for future work.

Keywords:

software security, software protection, information leakage, software development process, tamper-resistant software

List of Major Publications

Journal Papers

- Yuichiro Kanzaki, Hiroshi Igaki, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto, "Quantitative Analysis of Information Leakage in Security-Sensitive Software Processes," *IPSJ Journal*, Vol.46, No.8, pp.2129–2141, Information Processing Society of Japan, August 2005.

- Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto, "A Software Protection Method Based on Instruction Camouflage," *The IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences (Japanese Edition)*, Vol.J87-A, No.6, pp.755–767, June 2004. (in Japanese)

International Conference Papers

- Yuichiro Kanzaki, Hiroshi Igaki, Masahide Nakamura, Akito Monden, and Ken- ichi Matsumoto, "Characterizing Dynamics of Information Leakage in Software Process," In *Proc. 3rd Australasian Information Security Workshop (AISW2005)*, Vol.44, pp.145–151, January 2005.

- Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto, “Exploiting Self-Modification Mechanism for Program Protection,” In *Proc. 27th Computer Software and Applications Conference (COMPSAC2003)*, pp.170–179, November 2003.

Domestic Conference Papers

- Yuichiro Kanzaki, Hiroshi Igaki, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto, “Evaluating the Risk of Information Leakage in Software Process,” *Computer Security Symposium 2004 (CSS2004)*, Vol.2, pp.775–780, October 2004. (in Japanese)
- Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto, “Protecting Software Programs by Replacing Instructions at Runtime,” *Technical Report of IEICE*, ISEC2002-98, pp.13–19, December 2002. (in Japanese)

Patent Applications

- Akito Monden and Yuichiro Kanzaki, “Program, Apparatus and Method for Adding Self-Modifying Code,” *Japan Patent Pending*, 2002-355881, December 2002. (in Japanese)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1. Background

As software involving security-sensitive *secret information* increases, protection of internal secrets from being leaked out to software users has become an overarching issue in today's software development. A wide variety of secret information is present in today's software systems as follows:

Security-Sensitive Data: Security-sensitive data are constant values (e.g., numeric values and alphabet strings) that are related to system security and/or user privacy. For instance, in CPPM (Content Protection for Prerecorded Media)/ CPRM (Content Protection for Recordable Media) standards, a cipher key called a "device key" and related constant values called an "S-Box" must be secretly implemented in a digital contents player software and must be concealed from its users [1]. If a user obtains such secret values, illegal copying or use of digital contents could occur [65].

Security-Sensitive Algorithm: A security-sensitive algorithm is a logical sequence of instructions that is related to system security or is commercially valuable. For instance, the decryption algorithm of CSS (Contents Scrambling System) standards, which is still commonly used for DVD media protection, should have been concealed from users [21], although the algorithm was revealed by a “crack” in 1999. As a result, programs which subvert DVD copy protection are widely distributed these days [54].

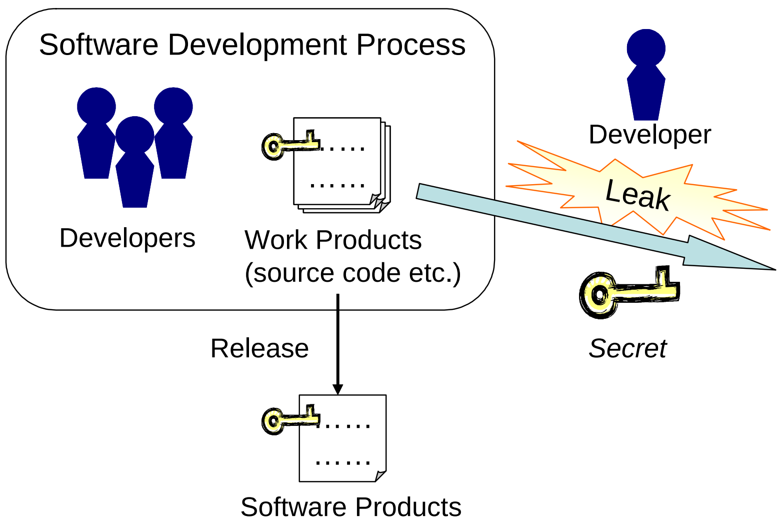
Security-Sensitive Branch Points: A security-sensitive branch point is a set of conditional branch instructions related to system security. For instance, it must be *extremely difficult* for users to find and modify conditional branch instructions that check whether a user is licensed or not, to avoid illegal use or copying of software [21, 23, 26].

Illegal activities caused by the revealment of such secret information are fatal threats to the software industry since those activities bring about serious financial damage to software suppliers and vendors [49]. In order to prevent such secret information from being revealed to users, the following two avenues of obtaining secrets must be obstructed:

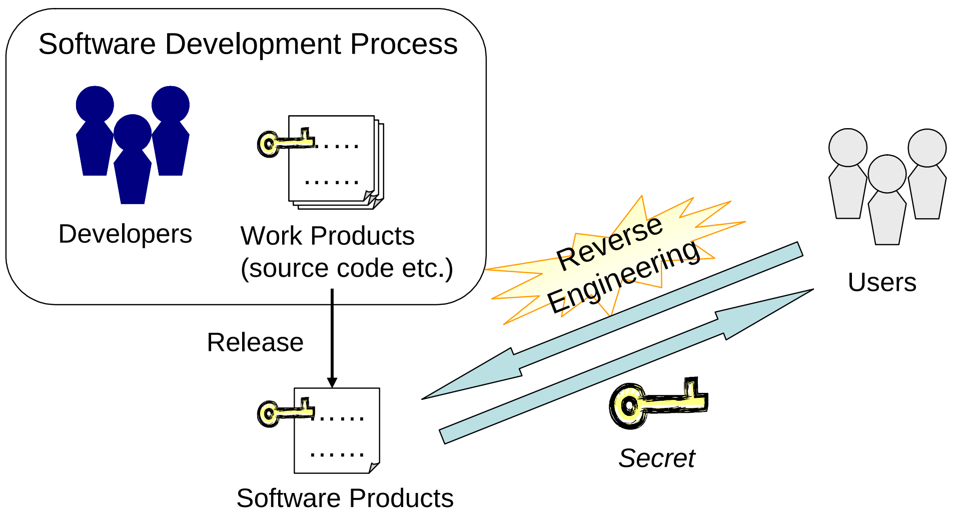
1. Obtaining secret information through leaked work products of a software process (Figure 1.1(a)).

2. Obtaining secret information by reverse engineering of software products (Figure 1.1(b)).

The two avenues of obtaining secrets are described in detail below.



(a) Obtaining secret information through leaked work products of a software process



(b) Obtaining secret information by reverse engineering of software products

Figure 1.1. Two avenues of obtaining secrets

1. Obtaining secret information through leaked work products of a software process

There is a possibility that work products (e.g., source code, specification) of a software development process leak out to users. The leakage of work products could have many different causes, such as having a computer that has secret information stolen or misplaced, unauthorized access to the computer via the Internet, and illegal sales of secret information by insiders [3]. For example, the source code of Microsoft Windows NT and Windows 2000 has been leaked onto the Internet [7]. Certain evidence seems to point to a Microsoft partner company as the source of the leaked code [36,38]. Microsoft is concerned about the potential theft of its handiwork because this theft calls into question intellectual property rights [37].

2. Obtaining secret information by reverse engineering of software products

Even if a user cannot obtain leaked work products that include secret information, he/she might be able to obtain the secret information by the *reverse engineering* of software products (i.e., software implementation). Reverse engineering is the process of taking a software program apart and analyzing its workings in detail. While reverse engineering commonly takes place in software maintenance tasks such as the investigation of interoperability and security auditing, it also has been used for obtaining secret information involved in software by end users.

On open systems such as Windows and Unix, information about the API of the OS and the architecture of CPU [59] is open to the public. In addition, tools for analyzing software such as disassemblers, binary editors and debuggers are easily available. Since it is easy for users to reverse engineer compared to conventional embedded devices [25], software running on open systems is always exposed to the menace of reverse engineering attacks. In fact, many illegal activities have been and are currently occurring through reverse engineering, and many software developers worry about their applications being reverse engineered [18, 52, 57].

1.2. Goal and Approach

The goal of this dissertation is to explain how to prevent secret information from being revealed to users. Assuming that a user can obtain secret information through leaked work products of software processes, and by reverse engineering of software products, two methods for protecting secrets are presented:

1. A method for evaluating the risk of leaking security-sensitive work products
2. A method for protecting software against reverse engineering attacks

The approaches of each method are described below.

A method for evaluating the risk of leaking security-sensitive work products

Since leakage of security-sensitive work products is caused by human errors in many cases [3], minimizing the amount of *knowledge transfer* of secret information among software developers to prevent the information from leaking out is important. However, from the viewpoint of efficiency, knowledge transfer should be improved, since it helps developers acquire a similar understanding of the process [32, 35]. Thus, people who design a security-sensitive software process (e.g., software process analyst, software process designer) should carefully consider the balance between the productivity and the risk of leakage, when they determine the structure of the software process and the assignment of developers to each process.

As a method that is useful for preventing leakage of security-sensitive software process, a framework to quantitatively evaluate the risk of *information leakage*, that is, the knowledge transfer with irrelevant products, for a given software process is presented. To achieve this, first the problem of information leakage is formulated by introducing a formal software process model. Next, assuming that the knowledge of the irrelevant products can be transferred, a method to compute the probability of each developer knowing each product is presented. The probability reflects the risk that someone leaked the product to the developer. This probability is derived from the given software process model using a recurrence formula.

This method is introduced in detail in Chapter 2.

A method for protecting software against reverse engineering attacks

First, how typical reverse engineering attacks are performed to obtain a solution for protecting software against the attacks is considered. Although it is difficult to clearly answer how software is reverse engineered since there are many ways to attack software programs, it is certain that an attacker (i.e., a user who performs reverse engineering attacks) has to understand a program to reverse engineer that program successfully.

Figure 1.2 shows a scenario, where an attacker obtains clues to nullify the password-checking routine. First, the attacker disassembles the binary program into an assembly program to make it easy to understand. Then, the attacker often narrows the range of analysis to reduce the cost of program understanding. Then, a program fragment containing the checking routine is identified and understood by the attacker. As a result, the checking routine may be canceled or cropped,

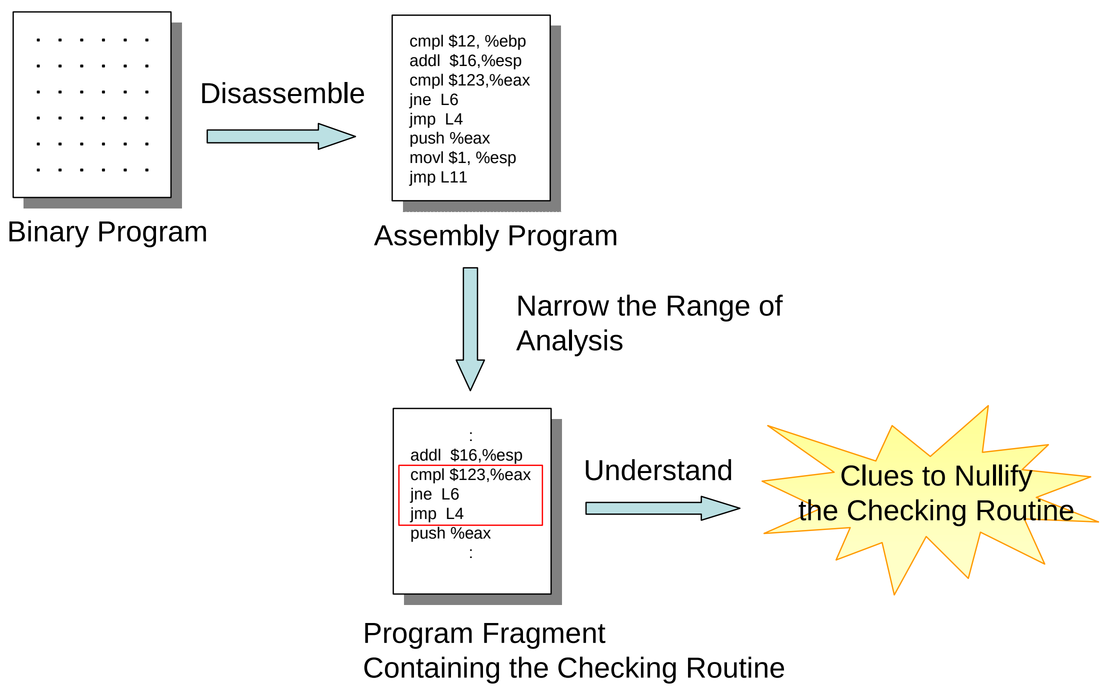


Figure 1.2. A scenario of obtaining clues to nullify the password-checking routine

and the program may be used without password authentication. As seen in this example, whenever software is reverse engineered, there must be a process where an attacker reads the program and tries to understand the program.

An effective method to protect software against reverse engineering attacks is to increase the cost of program understanding. The key idea of the method presented in this dissertation is to *camouflage* a significant amount of instructions with dummy instructions in a program. Our method first overwrites the instructions with dummy instructions. Then, the program itself restores the instructions within a certain period of execution using a self-modification mechanism. This drastically increases the complexity of program understanding, since the understanding fails if the dummy instructions are inspected as they are.

This method is presented in detail in Chapter 3.

Chapter 2

Quantitative Analysis of Information Leakage in Security-Sensitive Software Process

2.1. Introduction

Information leakage is a serious problem in today's advanced information society. Many incidents have been reported recently, including leakage of user accounts and passwords [61], medical records [47], and product source codes [7]. Because of its impact on trust and security, minimizing the *risk* of information leakage is now a social responsibility for every organization.

Although information leakage occurs in many domains, the leakage in a software development process is especially focused on. The development of a complex and large-scale software system requires the collaborative effort of many people with an elaborate *software process* [30]. A (whole) software process is composed of partially-ordered *sub-processes* (simply called *processes*) such as design, coding and testing. Each process has *work products* (simply *products*) as either the input or output of the process.

Typically, a number of developers including manages, designers and programmers, participate in a common process. Given input products, the developers collaborate with each other, and produce output products. Through the collaboration, they usually share their knowledge of the products in order to achieve the process efficiently. Thus, when multiple developers participate in a process, certain *knowledge transfer* may occur in the process. From the viewpoint of efficiency, knowledge transfer should be improved, since it helps developers to acquire a similar understanding of the process [35].

However, in the development of security-sensitive software such as DRM applications [11, 41], the transfer of product knowledge is not always encouraged. To understand this better, a simple example is introduced.

Figure 2.1 shows a software process consisting of two processes P1 and P2. P1 is a design process conducted by developer Alice, where Alice produces a product *Specification* from two given products, *Requirement* and *SecretInfo*. It is here assumed that *SecretInfo* is confidential information (e.g., device keys or S-BOX of CPRM systems [1]) that only Alice is authorized to see, and that must appear in *Specification* in a certain encoded form. P2 is a coding process in which Alice, Bob, and Chris participate. The three developers collaborate with each other and produce *Code* from *Specification*. Since Alice knows *Specification*, Alice may explain it to Bob and Chris in the collaboration. This knowledge transfer is reasonable, since *Specification* is necessary for Bob and Chris to perform P2 together. Even if Alice does not give any explanation, Bob and Chris must know *Specification*. On the other hand, both *Requirement* and *SecretInfo* are *irrelevant*

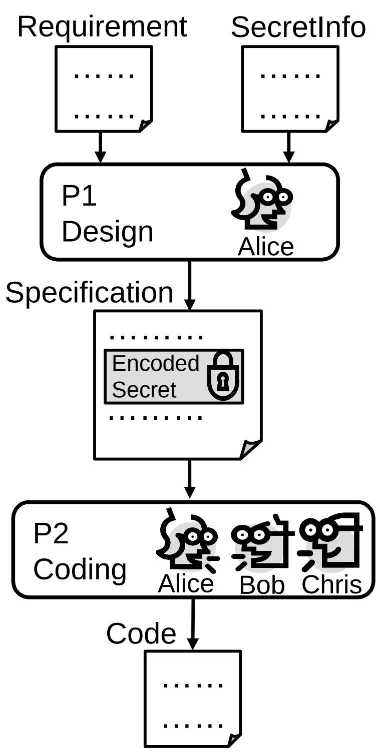


Figure 2.1. Security-sensitive software process

to P2, since they are not directly connected to P2. However, what happens if Alice tells Bob or Chris the knowledge about *SecretInfo* in P2? In this case, Bob or Chris gets to know *SecretInfo*, which ruins the security scheme.

From the above observation, in a security-sensitive software process, the knowledge transfer with any such irrelevant products should carry a warning such as *information leakage*. Note that the risk of information leakage varies, depending on the structure of the software process and the assignment of developers to each process. For example, in Figure 2.1 if Alice is not assigned to P2, no leakage occurs.

The goal of this chapter is to propose a framework to evaluate quantitatively the risk of information leakage for a given software process. To achieve this, first the problem of information leakage by introducing a formal software process model is formulated. The model is based on the conventional *process-centered software engineering environment* [19, 22]. The contribution is to formulate *product knowledge* of each developer on top of the model, focusing on the process structure and the developer assignment.

Next, assuming that the knowledge of the irrelevant products can be transferred (i.e., leaked), a method to compute the probability of each developer knowing each product is presented. The probability reflects the risk that someone leaked the product to the developer. The probability is derived from the given software process model using a recurrence formula.

To show applicability to practical or actual settings, three case studies are conducted. The first case study demonstrates how the information leakage varies depending on the assignment of developers. In the second case study, an application to find an optimal assignment of developers for a constrained software process is presented. In the last case study, the relationship between the process structure and the information leakage is investigated. The proposed method provides a simple but powerful means to perform quantitative analysis on information leakage in a security-sensitive software process.

The rest of this chapter is organized as follows: In Section 2.2, definitions of the software process model are given. Section 2.3 describes the proposed method for characterizing the dynamics of information leakage. In Section 2.4, the case studies are conducted. In Section 2.5, two issues are discussed: setting probability of leakage among developers, and leakage in a deterministic manner. The related work is also reviewed in the section.

2.2. Preliminaries

2.2.1 Software Process Model

First, a definition of a software process model is presented. The model is based on the conventional *process-centered software engineering environment* [19, 22], where the software development is modeled by partially-ordered activities (*processes*) operating with given or intermediate *working products.* In addition to the conventional model, the proposed model involves the *assignment of developers* to specify explicitly who participates in each process.

Definition 1 (Software Process Model) A software process model is defined by $P = (U, WP, PC, I, O, AS)$, where:

- U is a set of all *developers* participating in the development.
- WP is a set of all *work products.*
- PC is a set of all *processes.*
- I is an *input* function $PC \rightarrow 2^{WP}$ that maps each process $p \in PC$ onto a set $IP(\subseteq WP)$ of *input products* of p.
- O is an *output* function $PC \rightarrow 2^{WP}$ that maps each process $p \in PC$ onto a set $OP(\subseteq WP)$ of *output products* of p.
- AS is an *developer assignment function* $PC \rightarrow 2^{U}$ that maps each process $p \in PC$ onto a set of developers participating in the process p.

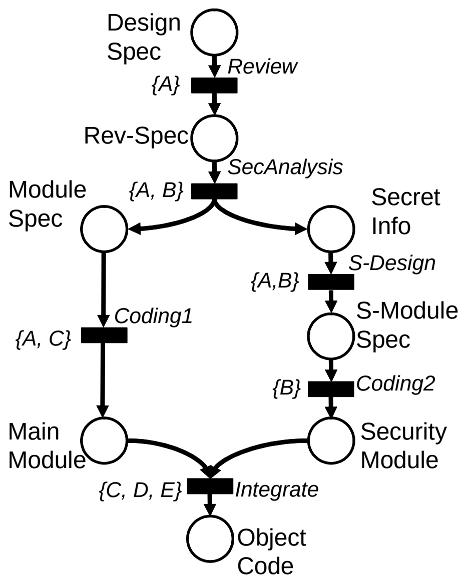
Figure 2.2(a) shows an example of the software process model, which simplifies an implementation stage of a security-sensitive application. The model contains

```
U = { A, B, C, D, E}
PD = { DesignSpec, Rev-Spec, SecretInfo,
         ModuleSpec, S-ModuleSpec, MainModule,
         SecurityModule, ObjectCode }
PC = {Review, SecAnalysis, S-Design, Coding1,
        Coding2, Integrate  }
I(Review)={DesignSpec}
I(SecAnalysis)={Rev-Spec}
I(S-Design)={SecretInfo}
I(Coding1)={ModuleSpec}
I(Coding2)={S-ModuleSpec}
I(Integrate)={MainModule, SecurityModule}

O(Review)={Rev-Spec}
O(SecAnalysis)={ModuleSpec,SecretInfo}
O(S-Design) = {S-ModuleSpec}
O(Coding1)={MainModule}
O(Coding2)={SecurityModule}
O(Integrate)={ObjectCode}

AS(Review)={A}
AS(SecAnalysis)={A, B}
AS(S-Design)= {A, B}
AS(Coding1)={A, C}
AS(Coding2)={B}
AS(Integrate)={C, D, E}
```

(a) Software Process Model



(b) Petri-Net Representation

Figure 2.2. Process model example

five developers, eight work products, and six processes. The scenario assumed in the model is briefly explained as follows:

Example Scenario: The implementation stage produces an object code from a given design specification. In this stage, the design specification is revised by a review process. Next, by applying a security analysis to the reviewed specification, all security-sensitive information is isolated from the specification. The rest of the specification is called *ModuleSpec*. From the security information, authorized developers design a specification, called a *S-ModuleSpec* for an independent security module in which the raw security information is encoded. A main module and the security module are coded respectively from *ModuleSpec* and the *S-ModuleSpec*. Finally, these two modules are integrated as the object code.

In Figure 2.2(a), let us consider the process *Review*. This process models the review of the design specification. Review takes *DesignSpec* as an input product, and outputs a reviewed specification (*Rev-Spec*). In this example, only developer A is responsible for conducting the process. Next, the process *SecAnalysis* is considered. This process takes *Rev-Spec* as an input, and outputs *ModuleSpec* and *SecretInfo*. Two developers A and B participate in the process. Through a similar discussion, how the process model achieves the example scenario can be seen.

By definition, each process has a set of input products and a set of output products. This definition allows us to draw a given process model by a *Petri net* [43], by associating WP with places, PC with transitions, and by connecting a place and a transition with an arc according to I and O. Figure 2.2(b) shows a schematic representation of the example process with Petri net. Also, a set of developers is associated with each corresponding transition based on AS, as depicted in the left of the transition. Note that the use of the Petri net is just for better comprehension of the overview of the process structure, but is not essential to the methodology here.

2.2.2 Order among Processes

Suppose that $P = (U, WP, PC, I, O, AS)$ is given. For $p \in PC$, $w \in I(p)$ and $w' \in O(p)$, a triple (w, p, w') is used to represent a *product dependency* of process p, where a work product w' is produced from w via p. The product dependencies implicitly specify a *partial order* between processes, since a process needs input products that have been previously generated by other processes.

Definition 2 (Order of Processes) For processes p and p', p is *executed before* p' (denoted by $p < p'$) iff there exists a sequence (w_0, p, w_1) (w_1, p_1, w_2) ... (w_{n-1}, p', w_n) of product dependencies. For processes q and q', if any $<$ is not defined between q and q', q and q' are *independent*.

Let us consider the previous example. As depicted in Figure 2.2(b), the order among the six processes can be seen, i.e., *Review* $<$ *SecAnalysis* $<$ *Coding1* $<$ *Integrate*, and *Review* $<$ *SecAnalysis* $<$ *S-Design* $<$ *Coding2* $<$ *Integrate*. Note that the order is *partial* at this moment. Indeed, no order between *Coding1* and *S-Design* (or *Coding2*) is defined, thus they are independent. The independent processes can be executed in any order, even concurrently.

2.2.3 Assumption on Software Process Model

The following two assumptions are used for a given process model $P = (U, WP, PC, I, O, AS)$.

Assumption A1: There exists no sequence (w_0, p_0, w_1) (w_1, p_1, w_2) ... (w_{n-1}, p_n, w_n) of product dependencies such that $w_0 = w_n$.

Assumption A2: For any pair of independent processes p and p', if $AS(p) \cap AS(p') \neq \phi$, then an order between p and p' must be given.

Assumption A1 states that the product dependencies never form a loop. This is quite reasonable for general software processes. Indeed, it is unrealistic to assume that a work product newly obtained can be used as the input of the processes that have been completed previously. By this assumption, there is a consistent partial order among processes for a given sequence of product dependencies.

Assumption A2 states that independent processes p and p' must be ordered when the same developer is assigned to both p and p'. This assumption is based on the observation that a developer cannot engage in more than one process simultaneously. Let us consider the process model in Figure 2.2. In this example, processes *Coding1* and *S-Design* are independent. However, they cannot be executed simultaneously, since the same developer A is assigned to both processes (i.e., $AS(S - Design) \cap AS(Coding1) = \{A\}$). Hence, it is required to give an order between these processes, for instance, *S-Design* $<$ *Coding1*, so that A conducts *S-Design* first.

By these assumptions, if a developer u is fixed, then the processes in which u participates are totally-ordered.

Proposition 1 Let $P = (U, WP, PC, I, O, AS)$ be a given process model with Assumptions A1 and A2. For a developer $u \in U$, let $PC_u = \{p | p \in PC \wedge u \in AS(p)\}$ be a set of all processes to which u is assigned. Then, PC_u is *totally-ordered.*

Consider the process model in Figure 2.2 with *S-Design* $<$ *Coding1*. Then, the processes to be conducted by each user are ordered as follows:

PC_A : *Review* $<$ *SecAnalysis* $<$ *S-Design* $<$ *Coding1*

PC_B : *SecAnalysis* $<$ *S-Design* $<$ *Coding2*

PC_C : *Coding1* $<$ *Integrate*

PC_D : *Integrate*

PC_E : *Integrate*

Since PC_u are totally-ordered, any process in PC_u has at most one *immediate predecessor.*

Definition 3 (Predecessor of Process) Let $p_{u_1}, p_{u_2}, ..., p_{u_k}$ be all processes in PC_u such that $p_{u_1} < p_{u_2} < ... < p_{u_k}$. For $p_{u_i} \in PC_u$, $p_{u_{i-1}}$ is called *immediate predecessor* of p_{u_i} with respect to u, which is denoted by $pred_u(p_{u_i})$. Also, $pred_u(p_{u_1})$ is defined to be ϵ (empty).

In the above example, there is $pred_A(Coding1) = S\text{-}design$, which means that A participates in *S-design* immediately before $Coding1$. Also, there is $pred_C$ $(Coding1) = \epsilon$ meaning that $Coding1$ is the first process that C engages in.

2.3. Characterizing Dynamics of Information Leakage

2.3.1 Product Knowledge of Developers

To perform a process p, developers engaging in p must know all the input products of p. Based on the input products, they develop the output products. Hence, when finishing p, developers should be acquainted with the output products as well. Thus, when a process is performed, the developers acquire knowledge about the related (i.e., input/output) products. For each developer, the knowledge is accumulated in the sequence of completed processes. This dynamics depends on the given process model, specifically, I, O, and AS.

For example, consider the example in Figure 2.2. Developer A participates in process *Review*. Hence, when *Review* is finished, A must know the products *DesignSpec* and *Rev-Spec*. Similarly, the completion of *SecAnalysis* provides the knowledge of *Rev-Spec*, *ModuleSpec*, and *SecretInfo* for both A and B. Thus, when A completes *SecAnalysis*, A knows four products; *DesignSpec, Rev-Spec, ModuleSpec, SecretInfo.*

Definition 4 (Product Knowledge) Let $P = (U, WP, PC, I, O, AS)$ be a given software process model. For $u \in U$ and $p \in PC$, a set of working products $Know(u, p)$ $(\subseteq WP)$ is defined s.t.

$$Know(u, p) = \bigcup_{u \in AS(p') \wedge p' \leq p} (I(p') \cup O(p')) \tag{2.1}$$

Table 2.1. $Know(u, Integrate)$ ($u \in \{A, B, C, D, E\}$)

u	*DSpc*	*RSpc*	*SInfo*	*MSpc*	*SSpc*	*MMo*	*SMo*	*OCd*
A	1	1	1	1	1	1	0	0
B	0	1	1	1	1	0	1	0
C	0	0	0	1	0	1	1	1
D	0	0	0	0	0	1	1	1
E	0	0	0	0	0	1	1	1

$Know(u, p)$ is called the *product knowledge* of developer u at the completion of process p.

The term "knowledge" is used in an abstract sense, which can be refined in terms of, for instance, the essential idea or mechanism, the product's document itself, or the access method to the product.

Let us compute $Know(B, Coding2)$ with Figure 2.2. Before *Coding2*, B has participated in *SecAnalysis* and *S-Design*. Hence, accumulating the input/output products of these three processes, there is $Know(B, Coding2)$ = { *Rev-Spec, SecretInfo, ModuleSpec, S-ModuleSpec, SecurityModule* }.

For convenience, $Know(u, p)$ is represented with a *binary vector*. Let w_1, w_2, ..., w_n be all work products in *WP*. Then, $Know(u, p) = [wp_1, wp_2, ..., wp_n]$ is denoted, where $wp_i = 1$ iff $w_i \in Know(u, p)$, otherwise $wp_i = 0$. Then, the product knowledge of all users at the completion of the last process (i.e., *Integrate*) can be represented in Table 2.1.

2.3.2 Leakage of Product Knowledge

Now suppose a situation such that a developer may *share* his/her product knowledge to other developers sharing the same process.

As an example, consider *Coding1* in Figure 2.2. This process is shared by A and C. Assuming an order *S-Design* $<$ *Coding1*, the product knowledge of A and C at *Coding1* are computed as follows:

$$\begin{array}{rccccccccc} & \mathit{DSpc} & \mathit{RSpc} & \mathit{SInfo} & \mathit{MSpc} & \mathit{SSpc} & \mathit{MMo} & \mathit{SMo} & \mathit{OCd} & \\ Know(A, Coding1) = [& 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 &] \\ Know(C, Coding1) = [& 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 &] \end{array}$$

Coding1 is the first process that C participates in. Hence, at this moment, C is supposed to know only *ModuleSpec* and *MainModule*. C does not need to know all the rest of the products. On the other hand, A has more product knowledge than C, because A has previously participated in three other processes.

Assume now that during *Coding1*, A tells C the product knowledge that C does not know, for example *SecretInfo*, with some probability. As a result, C learns *SecretInfo* although C has never directly touched it before. Once C knows *SecretInfo*, the knowledge would be *propagated* to D and E, since C shares the subsequent process, *Integrate*, with D and E. As a result, the isolation of security information would be in vain.

Thus, when multiple developers work in the same process, the product knowledge can be spread from the developer who knows the product to developers who do not know. This is regarded as *information leakage in the software process*, which is specifically defined as follows.

Definition 5 (Leakage) For developers u, $u' \in D$, a work product $w \in WP$ and a process $p \in PC$, u *may leak* w to u' at p iff $\{u, u'\} \subseteq AS(p)$ and both $w \in Know(u, p)$ and $w \notin Know(u', p)$.

The above definition of leakage might be a bit broad. Indeed, this definition covers a case such that a security product w is known to an unauthorized developer u'. On the other hand, someone may say that it is not leakage if w is not a security-sensitive product, or if u and u' work for the same company. However, for simplicity and generality of the model, this broad definition is kept. A more detailed criteria of the leakage should be tuned depending on the target software process.

2.3.3 Stochastic Product Knowledge

Now, let us take the leakage of product knowledge into account in the model. Specifically, the following assumption for a given process model $P = (\, U\, , \mathit{WP}, \mathit{PC}, I, O, \mathit{AS}\,)$ is introduced:

Assumption A3: For $u, u' \in U$ and $w \in \mathit{WP}$, let $\mathit{leak}(u, w, u')$ be the probability that u leaks w to u'. It is assumed that $\mathit{leak}(u, w, u')$ is given for any u, u' and w.

Then, in a process p, a developer u may happen to know a product w such that $w \notin \mathit{Know}(u, p)$, since someone could leak w to u with a certain probability. This motivates us to deal with product knowledge in a *stochastic* manner.

Let us consider a probability that a developer u knows a work product w at the completion of process p, which $\mathit{Pkn}(u, p, w)$ is denoted. When u knows w at the completion of p, two cases can be considered.

Case C1: $w \in \mathit{Know}(u, p)$, or

Case C2: $w \notin \mathit{Know}(u, p)$ and some developers leak (or leaked) w to u.

Case C1 means that w is already counted in u's product knowledge. For this case, $\mathit{Pkn}(u, p, w) = 1.0$. Case C2 can be further divided into two sub-cases.

Case C2a: u knew w before p (via someone else), or

Case C2b: [$u \in AS(p)$] and [u did not know w before p] and [in p some developers sharing p with u leak w to u].

The probability that Case C2a holds is

$$P(C2a) = Pkn(u, pred_u(p), w) \tag{2.2}$$

which means that u knew w in the predecessor process. Next, the probability for Case C2b can be formulated by

$$P(C2b) = C(u,p) * \ (1 - Pkn(u, pred_u(p), w)) * \ P_{leak} \tag{2.3}$$

where $C : U \times PC \rightarrow \{0,1\}$ such that $C(u,p) = 1$ iff $u \in AS(p)$; otherwise $C(u,p) = 0$, and P_{leak} is the probability that some developers sharing p leak w to u.

Next, P_{leak} is formulated. Let $u_1, u_2, ..., u_j$ be developers who share p with u (i.e., $\{u_1, u_2, ..., u_j\} = AS(p) - \{u\}$). In order for u_i to leak w in p, two conditions are required: (1) u_i needs to have known w before p, and (2) u_i leaks w to u. Therefore, the probability that u_i leaks w to u in p is

$$Pkn(u_i, pred_{u_i}(p), w) * leak(u_i, w, u)$$

Moreover, u knows w iff at least one of $u_1, u_2, ..., u_j$ leaks w to u in p, which is the complement of "none of $u_1, u_2, ..., u_j$ leaks w to u in p". Hence,

$$P_{leak} = 1 - \prod_{u_i \in AS(p) - \{u\}} \{1 - Pkn(u_i, pred_{u_i}(p), w) * \ leak(u_i, w, u)\} \tag{2.4}$$

Combining all formulas together, $Pkn(u,p,w)$ is finally derived, which is the probability that u knows w at the completion of p :

$$Pkn(u,p,w) = \begin{cases} 1.0 \quad (\cdots \textbf{if} \quad w \in Know(u,p)) \\ \\ \quad Pkn(u, pred_u(p), w) \\ + \; C(u,p) \\ * \;\; (1 - Pkn(u, pred_u(p), w)) \\ * \;\; [1 - \prod_{u_i \in AS(p)-\{u\}} \{1 - Pkn(u_i, pred_{u_i}(p), w) \\ \qquad\qquad\qquad\qquad\qquad\qquad * \; leak(u_i, w, u)\}] \\ \qquad (\cdots \textbf{if} \quad w \notin Know(u,p)) \end{cases} \tag{2.5}$$

Note that $Pkn(u,p,w)$ is specified as a recurrence formula with respect to the process p. According to Assumptions A1 and A2, the set of processes that u participates in is totally-ordered. Hence, $pred_u(p)$ is uniquely obtained. Also, by Assumption A3, $leak(u_i,w,u)$ is given. Therefore, the value of $Pkn(u,p,w)$ can be calculated deterministically.

$Pkn(u,p,w)$ is now defined as *stochastic product knowledge.*

Definition 6 (Stochastic Product Knowledge) Let $P = (\ U,\ WP,\ PC,\ I,\ O,\ AS)$ be a given software process model with Assumptions A1, A2 and A3. Let $w_1, w_2, ..., w_n$ be all work products in *WP*. For $u \in U$, $p \in PC$, a vector $PKnow(u,p)$ is defined s.t.

$$PKnow(u,p) = [Pkn(u,p,w_1), Pkn(u,p,w_2),\ \ldots, Pkn(u,p,w_n)] \tag{2.6}$$

$PKnow(u,p)$ is called *stochastic product knowledge* of u at the completion of p.

Consider the example in Figure 2.2 with *S-Design* < *Coding1.* For the sake of simplicity, let us assume a fixed probability $leak(u,w,u') = 0.01$ for all $u, u' \in U$

Table 2.2. $PKnow(u, Integrate)$ ($u \in \{A, B, C, D, E\}$)

u	*DSpc*	*RSpc*	*SInfo*	*MSpc*	*SSpc*	*MMo*	*SMo*	*OCd*
A	1.0	1.0	1.0	1.0	1.0	1.0	0.0	0.0
B	0.0199	1.0	1.0	1.0	1.0	0.0	1.0	0.0
C	0.01	0.01	0.01	1.0	0.01	1.0	1.0	1.0
D	0.0001	0.0001	0.0001	0.01	0.0001	1.0	1.0	1.0
E	0.0001	0.0001	0.0001	0.01	0.0001	1.0	1.0	1.0

and $w \in WP$. Therefore, the stochastic product knowledge of all users at the completion of the last process (i.e., *Integrate*) can be obtained as shown in Table 2.2.

2.4. Case Studies

To show the applicability of the proposed method to practical software processes, this section conducts three case studies.

A software tool which automatically derives the stochastic product knowledge is implemented. The tool is written in C++, comprising about 600 lines of codes. The tool computes the stochastic product knowledge from a given text file, which includes software process model and values of $leak(u, w, u')$ in a text format. As for the performance, the tool required 0.07 seconds to compute the result for the example process shown in Figure 2.2, on a Pentium 4 PC (3.60GHz).

2.4.1 Case Study 1: Impact of Collaboration among Developers

The aim of this case study is to demonstrate how the *collaboration* among developers influences the *risk* of information leakage. Here, the software process model shown in Section 2.2.1 (see Figure 2.2) is further investigated. The stochastic product knowledge with varying AS on some processes is computed.

Let us recall the scenario of the process model. In the scenario, a work product *SecretInfo* is assumed to be confidential. Also, only developers A and B are authorized to access *SecretInfo*. When *S-Design* is completed, A and B are the only developers that know *SecretInfo*.

The interest is to evaluate the risk that *SecretInfo* is leaked to unauthorized developers C, D or E. For this evaluation, The developers assignment AS is varied in the subsequent three processes *Coding1*, *Coding2* and *Integrate*.

For convenience, first the following parameters is defined:

- $U_{aut} = \{A, B\}$: authorized developers.
- $U_{uaut} = \{C, D, E\}$: unauthorized developers.
- $PC_{tgt} = \{Coding1, Coding2, Integrate\}$: target processes where the developers assignment is varied.

The risk of the leakage depends heavily on how the authorized developers (A, B) collaborate with the unauthorized ones (C, D, E) in the target processes (PC_{tgt}). To characterize the collaboration, the following parameter for a process p is defined:

$$Col(p) = |U_{aut} \cap AS(p)| * |U_{uaut} \cap AS(p)| \tag{2.7}$$

Also,

$$Col = \sum_{p \in PC_{tgt}} Col(p) \tag{2.8}$$

is defined.

$Col(p)$ represents the number of combinations of authorized and unauthorized developers in a process p. This intuitively characterizes the *degree of collaboration* where an authorized developer *interacts* with an unauthorized one in p. For example, if $AS(p) = \{A, B, C, D\}$ with $U_{aut} = \{A, B\}$, $U_{uaut} = \{C, D\}$, then $Col(p) = 4$, which implies that there are 4 patterns where an authorized A or B interacts with an unauthorized C or D. Col is the total number of such interactions in the target processes. Hence, with the greater value of Col, the more the authorized developers can collaborate with the unauthorized ones.

For a fixed developers assignment $as = [AS(Coding1), AS(Coding2), AS(Integrate)]$, the risk that *SecretInfo* is leaked to unauthorized developers is formulated by:

$$Risk_{as} = \sum_{u \in U_{uaut}} Pkn(u, integrate, SecretInfo) \tag{2.9}$$

$Risk_{as}$ is characterized as the expected number of unauthorized developers knowing $SecretInfo$.

Using the developed tool, $Risk_{as}$ is computed for all possible assignments $as \in 2^U \times 2^U \times 2^U$. In the computation, it is assumed that $leak(u, w, u') = 0.01$ for every $u, u' \in U$ and $w \in WP$.

Figure 2.3 depicts the result. In this scattered plot, the horizontal axis represents Col, while the vertical axis plots $Risk_{as}$. Table 2.3 shows the average value of $Risk_{as}$ with respect to Col [1] .

[1] Due to the structure of the given process model, there is no developer assignment such that $Col = 17$.

Table 2.3. Average of *Risk* with respect to *Col*

Col	Risk
0	0.000000000
1	0.010040352
2	0.020082816
3	0.030094976
4	0.040112381
5	0.050125019
6	0.060119186
7	0.070156853
8	0.080101327
9	0.090217868
10	0.100071167
11	0.110321914
12	0.120011146
13	0.130566804
14	0.139962521
15	0.151329923
16	0.160040993
17	—
18	0.180650505

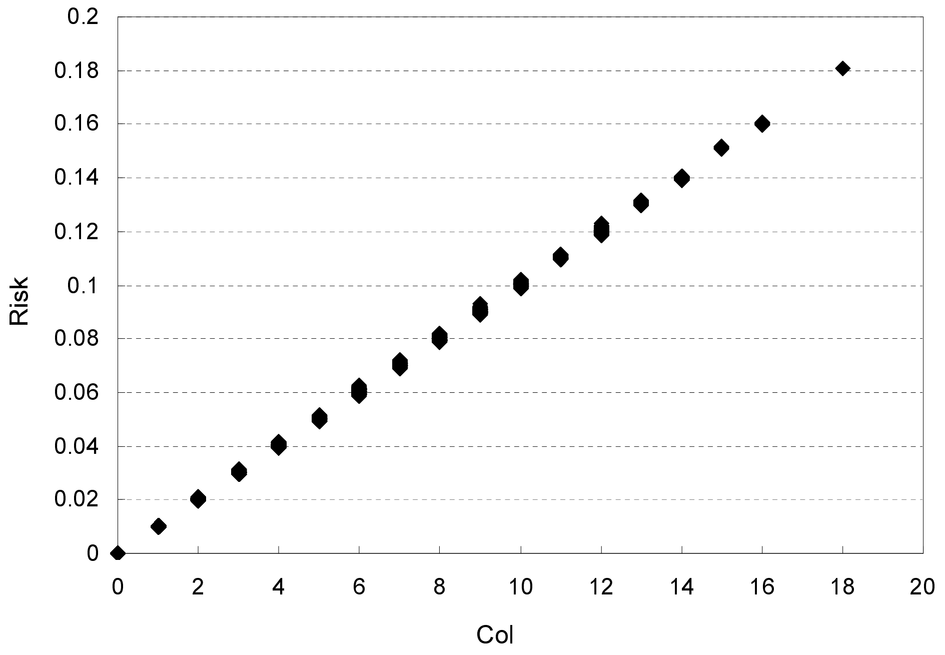


Figure 2.3. Computation result of *Risk*

As seen in the result, the risk that *SecretInfo* is leaked grows as Col increases. This increase implies that more collaboration among authorized and unauthorized developers causes the higher risk of information leakage. In this case study, each probability that a developer u leaks a product w to another u' is relatively small (i.e., $leak(u, w, u') = 0.01 = 1\%$). However, if the developers share many processes, the total probability of the leakage becomes significantly large. For $Col = 18$ where all of five developers are assigned to every target process, the risk becomes as large as 18%.

2.4.2 Case Study 2: Optimal Developers Assignment

This case study demonstrates how the proposed method can be used to find an optimal assignment of developers. For a software process model (with certain constraints), an assignment of developers AS is *optimal* if the risk of the leakage is minimized by AS.

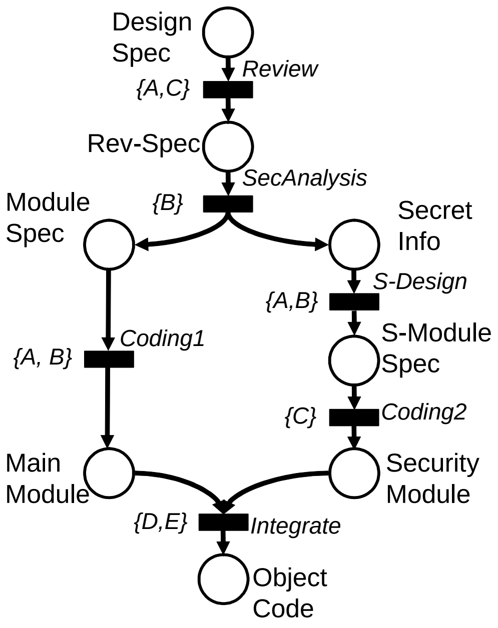
In this case study, a software process model $P = (U, WP, PC, I, O, AS)$ is used, where U, WP, PC, I and O are the same as those in Figure 2.2 and AS is not determined yet. Also, the same security scheme is assumed as the one in the previous case study. In general, a software process has certain constraints with respect to human resources, developing time, etc. As a typical example, the following constraints is imposed on P:

- All processes in WP must be performed by the effort of the total 10 developers (with overlaps).
- At most two developers can conduct each process.
- Both *SecAnalysis* and *S-Design* must be done only by the authorized developers ($U_{aut} = \{A, B\}$).
- $leak(u, w, u') = 0.01$ for all $u, u' \in U$, and $w \in WP$.

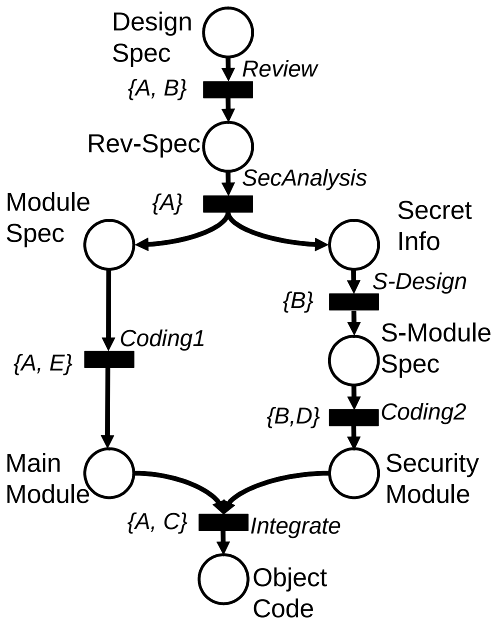
With the above constraints, the optimal assignment is computed, where the risk that *SecretInfo* is leaked is minimized.

The computation is rather straightforward. For every possible assignment as that satisfies the constraints, $Risk_{as}$ is computed (see Case Study 1). The optimal assignment is shown in Figure 2.4(a). In this assignment, there is no risk that *SecretInfo* is leaked (i.e., $Risk_{as} = 0.0000$). For a just comparison, the assignment is also computed where $Risk_{as}$ is maximized within the constraints. Figure 2.4(b) shows one of such cases, where $Risk_{as}$ is as large as 0.0300).

As can be seen in the optimal assignment (Figure 2.4(a)), after *S-design* there is no process involving authorized (A, B) and unauthorized developers (C, D, E), simultaneously. In contrast, as for the risky assignment (Figure 2.4(b)), in every



(a) Assignment that $Risk_{as}$ is minimized ($Risk_{as} = 0.0000$)



(b) Assignment that $Risk_{as}$ is maximized ($Risk_{as} = 0.0300$)

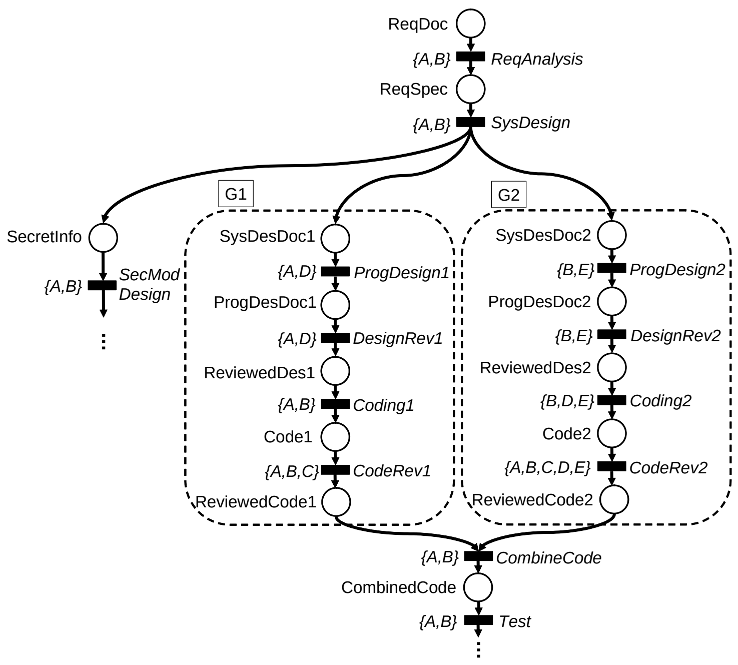
Figure 2.4. Optimal assignment and risky assignment

process after *S-design* either A or B shares the process with an unauthorized developer.

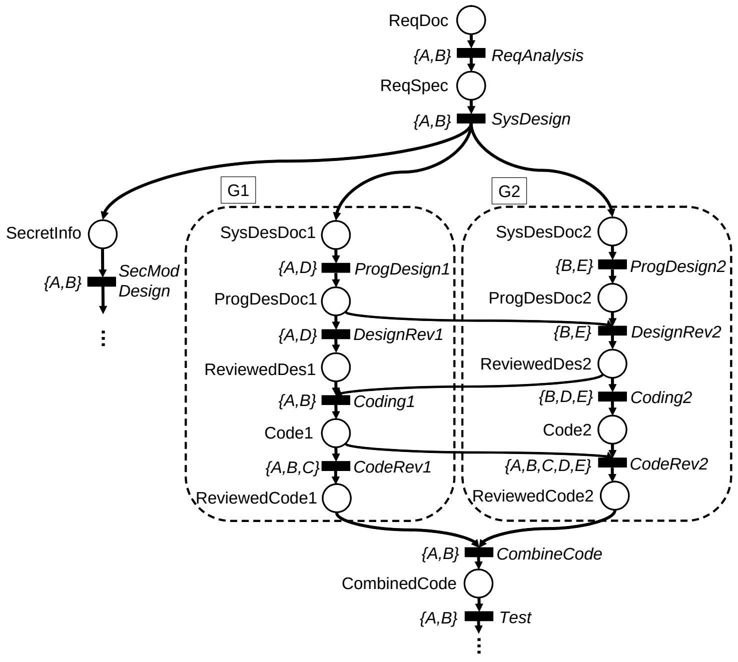
Thus, the proposed method can be also used as a powerful means to perform optimal tunings of the process configuration.

2.4.3 Case Study 3: Influence of Process Structure

The previous two case studies showed that the assignment of developers to each process is an essential factor for controlling the risk of the information leakage. The next interest is to examine the influence of the *process structure* (i.e., the shape of the Petri Net, intuitively) on information leakage.



(a) Structure $S1$



(b) Structure $S2$

Figure 2.5. Target software process model

For this, two software process models $S1$ and $S2$ shown in Figure 2.5 are introduced. The scenario of $S1$ is summarized as follows:

- Five developers A, B, C, D, and E participate in the software process.
- A and B first make the requirement specification (*ReqSpec*) from the given requirement document (*ReqDoc*) by the requirement analysis (*ReqAnalysis*).
- A and B conduct the system design (*SysDesign*) to divide the whole system into three sub-systems: a security module and two sub-modules.
- In *SysDesign*, A and B carefully isolate the confidential information (*SecretInfo*) from the rest of the system, to design the security module. For this isolation, it is assumed that only A and B are authorized to access *SecretInfo*. That is, $U_{aut} = \{A, B\}$ and $U_{uaut} = \{C, D, E\}$.
- The two sub-modules are developed in two concurrent processes $G1$ and $G2$ (shown as dotted boxes in Figure 2.5), each of which consists of four processes (program design, design review, coding, and code review).
- The reviewed codes of the two sub-modules are combined into one. The resultant code is applied to the subsequent test process.

Model $S2$ is almost the same as $S1$, but is somewhat ill structured. $S1$ and $S2$ have the same sets of products and processes. The developers assignment for $S1$ is also equal to the one for $S2$. However, for $S2$, some processes in $G1$ require some products in $G2$ (vice versa), which is represented by the three extra arcs between $G1$ and $G2$. These arcs suppress the execution order of some processes. For instance, *DesignRev2* can be performed only after *ProgDesign1* is completed.

For both $S1$ and $S2$, authorized developers A and B share some processes with the unauthorized ones C, D and E. Therefore, the knowledge of *SecretInfo*

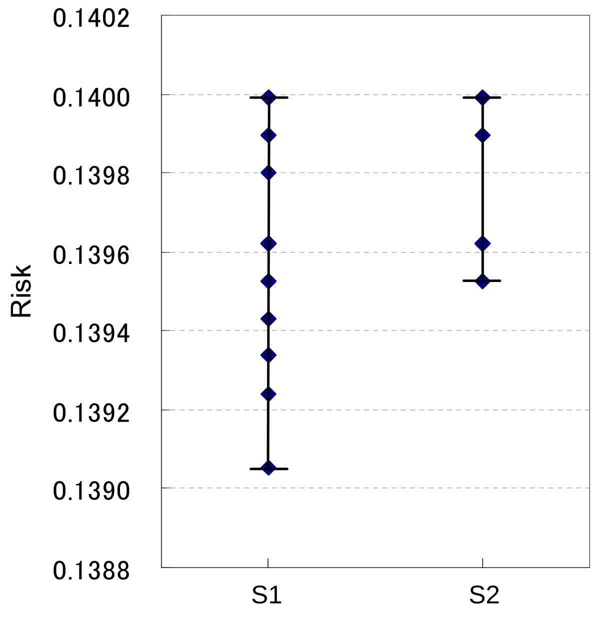


Figure 2.6. Influence of process structure

may leak. The interest is to see how the structural difference between $S1$ and $S2$ (i.e., the three extra arcs) impacts the risk of the leakage. As in a similar discussion in the previous case studies, the risk is formulated by:

$$Risk_{str} = \sum_{u \in \{C,D,E\}} Pkn(u, Test, SecretInfo) \tag{2.10}$$

Figure 2.6 illustrates the result, where the horizontal axis represents the name of the model, while the vertical axis plots $Risk_{str}$ for all possible execution orders of the processes. As seen in the result, $S2$ tends to have a higher risk of information leakage.

According to Assumption $A2$, it is required to give an order for between any independent processes. A simple solution is to give an order so that $Risk_{str}$ is minimized. For $S1$, the optimal order of processes is:

$ProgDesign2 < DesignRev2 < Coding2 < CodeRev2 < ProgDesign1 < DesignRev1 < Coding1 < CodeRev1$

where $Risk_{str} = 0.139052$. On the other hand, for $S2$, the optimal execution order of processes is:

$ProgDesign1 < ProgDesign2 < DesignRev2 < Coding2 < DesignRev1 < Coding1 < CodeRev2 < CodeRev1$
where $Risk_{str} = 0.139524$.

In $S1$, each process of $G1$ is completely independent of another process in $G2$. Therefore, the processes in $G1$ can be executed without concern for the progress of $G2$. On the other hand, for $S2$, the extra arcs suppress some execution order of the processes (for example, the order $Coding1 < DesignRev2$ cannot be assumed). Therefore, $S2$ cannot take a wide range of execution orders, which results in a higher risk of leakage in this experiment.

Thus, the structure of the process controls the execution order of processes, which significantly influences the risk of information leakage.

2.5. Discussion

2.5.1 Setting Value of $leak(u, w, u')$

The proposed framework requires the user to give an absolute probability $leak$ (u, w, u') for every u, w and u'. Although $leak(u, w, u')$ is assumed to be given (see Assumption A3), here the idea of how to determine the value in a practical setting is discussed.

By definition, the value $leak(u, w, u')$ characterizes the probability that a developer u leaks the product knowledge w to another developer u'. In reality, since this action of the leakage involves many *human factors*, it would be difficult to estimate an *exact* value of $leak(u, w, u')$ for each individual developer.

However, as demonstrated in Section 2.4, even if the user simply determines a *uniform value* of $leak(u, w, u')$ for all u, w and u', the user can analyze extensively the security aspects of the given software process. In this case, the user assumes that all developers are *equally likely* to leak their own product knowledge. Since varying the uniform value is easy, this type of analysis is useful for examining the process structure and developer assignment in the process planning stage.

On the other hand, if the user desires to estimate a more *realistic* value of $leak(u, w, u')$ for each individual developer, the user should refer to the *profile* information of developers, products, and working environment, which are supposed to be available in the organization. The profile information is used to derive objective attributes for each developer and product, involving; the age, the position, working experience, and security awareness of the developer, as well as organizational policies for confidential products and the trust of companies in collaboration. Based on the derived attributes, the user would be able to estimate $leak(u, w, u')$ in a more credible way.

A practical reasonable solution would be to introduce a *multi-grade* system with respect to the risk of leakage. For instance, the user evaluates each developer according to a three-grade system: dangerous, moderate, or safe. Then, the user assigns 0.1, 0.01 or 0.001 to $leak(u, w, u')$ for the dangerous, moderate, or safe developer u. Applying these settings to the proposed framework, the user can simulate a more realistic situation of information leakage.

In addition, a user who is required to precisely evaluate the risk may desire to vary a value of $leak(u, w, u')$ according to the content of the process. For example, in the example shown in Figure 2.2, a value of $leak(u, SecretInfo, u')$ in *Coding2*, which uses *S-ModuleSpec* (the product related to *SecretInfo*) can be different from one in processes that do not use *S-ModuleSpec* (e.g., *Coding1*). In order to perform such analysis, the parameter of *leak* needs to be extended as follows:

$leak(u, w, u', p)$: the probability that u leaks w to u' in p

Although this extension makes the way for setting *leak* more complexly, the user can evaluate the risk in a more precise way.

More sophisticated methods and their evaluation are left as a challenging issue in future work.

2.5.2 Deterministic Model

The information leakage has been characterized with the stochastic model so far. However, the leakage might be considered in a *deterministic* manner, assuming that any *potential* leakage actually occurs. Such a deterministic model could be used to determine the *safest* way to avoid the leakage.

Indeed, this deterministic model can be constructed within the proposed framework by setting every parameter $leak(u, w, u')$ to be either 0.0 or 1.0.

The deterministic model has the great advantage of simplicity in determining the value of $leak(u, w, u')$, which can be used to analyze some *special* cases. For example, suppose that a software process in a company X is performed by the collaboration with another external company Y. For every pair of developers x and y from X and Y, respectively, a deterministic model can be constructed, assuming that $leak(x, w, y) = 1.0$, and that no leakage occurs between developers within the same company. Then, the model can compute the set of product knowledge that could be transfered (or leaked) from the company X to Y.

Note, however, that the deterministic model can capture only some special situations according to the "always or never" basis. In the practical software process, *every* developer can have a possibility of leaking his product knowledge. The deterministic model cannot deal with the *degree* of the potential leakage that is significantly characterized by the process structure and the developer assignment.

In the above example, as long as x and y shares at least a certain process, the deterministic model always concludes that the leakage occurs. This conclusion is just by the worst-case analysis, and is *independent* of the process structure and the number of collaborations among x and y. Thus, the deterministic model tends to omit the detailed characteristics of the given software process model itself.

On the other hand, based on the assumption that every developer has a potential of leakage, the proposed stochastic model can quantitatively derive the degree of leakage of product knowledge, taking both the process structure and the developer assignment into account, as illustrated in Section 2.4. Although the stochastic model has a difficulty in justification of the probability setting on $leak(x, w, y)$, it is expected to provide a reasonable and useful metric for many process improvement tasks, such as constructing optimal software process under a constraint, and examining differences among multiple software process models.

Finally, note again that the proposed framework can deal with both the deterministic and stochastic models. So, the user can choose either model at his discretion.

2.5.3 Related Work

To the author's knowledge, no research study on a software process involving the leakage of product knowledge from one person to another exists. Chou et al., [10] presented a model for access control named *WfACL*, which aims to prevent information leakage within work flows that may execute among competing organizations. Chou et al., address issues related to the management of dynamic role change and access control. However, the model includes no concrete method to *evaluate* the risk of leakage quantitatively.

A numerical approach to compute information leakage might be to use *Gen-*

eralized Stochastic Petri Net (GSPN) [43] extensively. This approach is first examined. To do this, however, both the structure of process and the dynamics of leakage must be modeled in one GSPN. This complicates the net structure, and the state space becomes so large that the GSPN solver cannot compute the probability within a reasonable time. Therefore, it is decided to treat the process description and the leakage computation separately.

In addition, much research has been focused on different kinds of access control methods, such as *role-based access control* [20, 58], and *task-based access control* [62]. The goal of access control is to ensure that only authorized people are given access to certain resources (i.e., products in this chapter). However, the aim of the proposed method is not to control the access authority, but to evaluate the risk of leakage as unexpected knowledge transfer among developers.

Chapter 3

Protecting Software Based on Instruction Camouflage

3.1. Introduction

With the spreading use of networks, there has been remarkable progress in the flow configuration of programs and digital content. Accompanying this is an increasing demand for techniques which can prevent internal analysis and tampering with programs by end users. In programs containing digital rights management (DRM), for example, preventing the interception of the internal decryption key [11, 65] is necessary. In a program built into the hardware of cell phones and set-top boxes, preventing analysis or tampering by the user [53] is also required. An example of the problems caused by analysis is the phenomenon in which a decryption tool for DVD data was disseminated [21, 51]. This tool was based on an analysis of a DVD playback program, and greatly facilitated illegal copying of DVDs.

In this chapter, *analysis* is meant as an act of reverse engineering to acquire secret information (such as a secret key or algorithm) in a program. Typically, such an action is assumed to involve the following steps. First the attacker disassembles the program and tries to understand the resulting assembly program [40]. However, a tremendous amount of labor and time is required to understand the entirety of a large-scale program, and this amount of labor and time is not realistic. Consequently, the attacker restricts the range to be considered (the range which seems to be related to the secret information), and tries to understand only that range [8, 9]. The restriction and understanding of the range is repeated until the desired secret information is acquired.

This chapter proposes a method in which a large number of instructions in the program are camouflaged (hidden) in order to make it difficult to analyze an assembly language program accompanied by such range restriction. In the proposed method, an arbitrary instruction (target) in the program is camouflaged by a different instruction. By using a self-modification mechanism in the program, the original instruction is restored only in a certain period during execution [33, 34]. Even if the attacker attempts an analysis of the range containing the camouflaged instruction, it is impossible for him/her to correctly understand the original behavior of the program unless he/she notices the existence of the routine that rewrites the target (restoring routine). In order to make the analysis a success, the range containing the restoring routine must be analyzed, and the attacker is forced to analyze a wider range of the program. The proposed method can easily be automated, and the number of targets can be specified arbitrarily. By distributing a large number of targets and a large number of restoring routines in the program, it is likely that analysis by range restriction will be made very difficult.

Below, 3.2 proposes a systematic method which camouflages a large number of instructions in the program by self-modification. 3.3 discusses attacks on the proposed method, and analyzes the difficulty of the attack and its prevention. 3.4 reports a case study using the proposed method. 3.5 describes related studies.

3.2. Method of Software Protection by Instruction Camouflage

3.2.1 Attacker Model

The *attacker* is assumed to be as follows.

- The attacker has a disassembler and the ability to perform static analysis including range restriction by using the disassembler.
- The attacker has a debugger with a break-point function. By (manually) setting the break-point at an arbitrary point in the program, a snapshot at an arbitrary execution time (i.e., the content of the program which is the object of analysis loaded in the memory) can be acquired. However, he/she does not have tools by which a snapshot can be acquired automatically or by which dynamic analysis using the acquired snapshot history can be automated. He/she also lacks the ability to construct such a tool.

The above attacker corresponds to the "level 2 attacker" in the graded attacker model of Monden et al. [44, 45].

Assuming the above attacker model, the mechanism of program protection must satisfy the following requirements.

- The protection mechanism is not easily invalidated by static analysis using a disassembler.

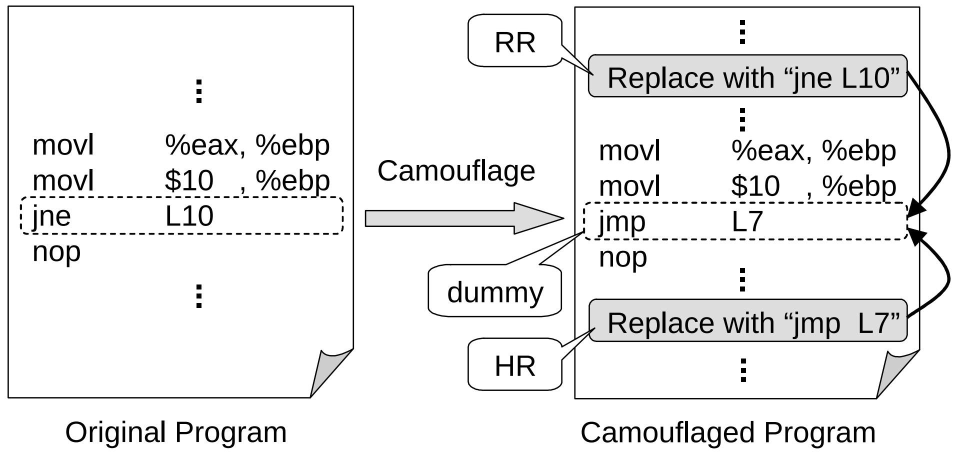


Figure 3.1. Example of camouflage

- The protection mechanism is not easily invalidated by an attack using (a few) snapshots.

In the following subsections, a protection method satisfying the above properties is proposed.

3.2.2 Key Idea

The proposed method makes it difficult for the attacker to understand the program by camouflaging program instructions. *Camouflage* means hiding the existence of the original instructions from the attacker by overwriting the instructions with dummy instructions.

Figure 3.1 shows an example of camouflage [1] . Consider the situation in which the instruction `jne L10` in the assembly program to be protected is camouflaged. First, a dummy instruction for `jne L10`, that is, an instruction with different content, is constructed. Suppose that `jmp L7` is constructed as the dummy in-

[1] An Intel X86 type CPU is assumed as an example for description. The assembly language instructions are based on the AT&T syntax.

struction. Then, overwriting with `jmp L7` is performed at the position of `jne L10`.

Next, a self-modification routine is added. *Self-modification* means the process of modifying the content of instructions in the program during execution. There are two types of self-modification routines. One is a routine that rewrites the camouflaged instruction to the original content (*RR* in Figure 3.1). This routine assures the execution of the original content of the program. In the case of Figure 3.1, the routine rewrites the camouflaged instruction as `jne L10`.

The other is a routine that again rewrites the instruction which has been returned to the original instruction by *RR*, as the dummy instruction (*HR* in Figure 3.1). This routine is intended to hinder an attacker who has a snapshot acquisition capability from determining the original instruction. In the case of Figure 3.1, a routine automatically rewrites the instruction to be camouflaged as `jmp L7`.

The dummy instruction takes the form of the original instruction only between the execution of *RR* and the execution of *HR*. Consequently, it is difficult for the attacker to determine that the original instruction is overwritten by the dummy instruction `jmp L7` by simply observing the neighborhood of the camouflaged instruction. Even if the snapshot of the program after execution of *HR* is acquired, it is impossible to determine the original instruction from the obtained snapshot.

The above instruction camouflage is repeated several times on the assembly program to be protected, in order to make the program difficult to understand. Figure 3.2 shows conceptually the program obtained after instruction camouflaging has been repeated many times. It is evident that a large number of instructions in the program have been overwritten by dummy instructions before execution (● in Figure 3.2). For each dummy, there exists a routine that rewrites the instruction to the original instruction (before it was overwritten by

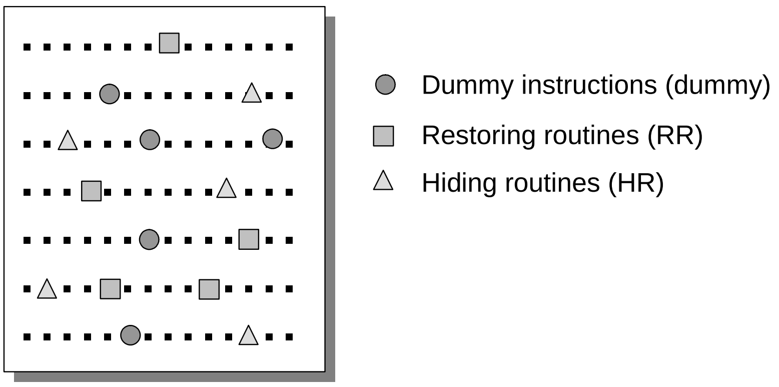


Figure 3.2. Image of a camouflaged program

the dummy instruction) (■ in Figure 3.2), and the routine that during execution takes the instruction rewritten to the original instruction by the above routine and rewrites it again as the dummy instruction (▲ in Figure 3.2).

If the part of the program which the attacker attempts to analyze contains a dummy instruction, he/she cannot correctly determine the original behavior of the program by examining only that part. In order to understand the program correctly, he/she must know that rewriting is performed, and determine the content of each dummy instruction in the part to be understood before it was overwritten. In order to obtain this information, however, he/she must locate the routine that rewrites the instruction to the original instruction within the whole program, which requires a tremendous effort.

3.2.3 Preliminary

The terminology in the proposed method is defined as follows. The *original program* O is the program before camouflaging is to be applied. The *target instruction* is the instruction which is the target of camouflaging in O. A *dummy instruction* is an instruction which is written over the target instruction in order to camouflage the target instruction. When the user defines multiple target instructions, the i-th target instruction is written as $target_i$ and the instruction used to camouflage $target_i$ is written as $dummy_i$.

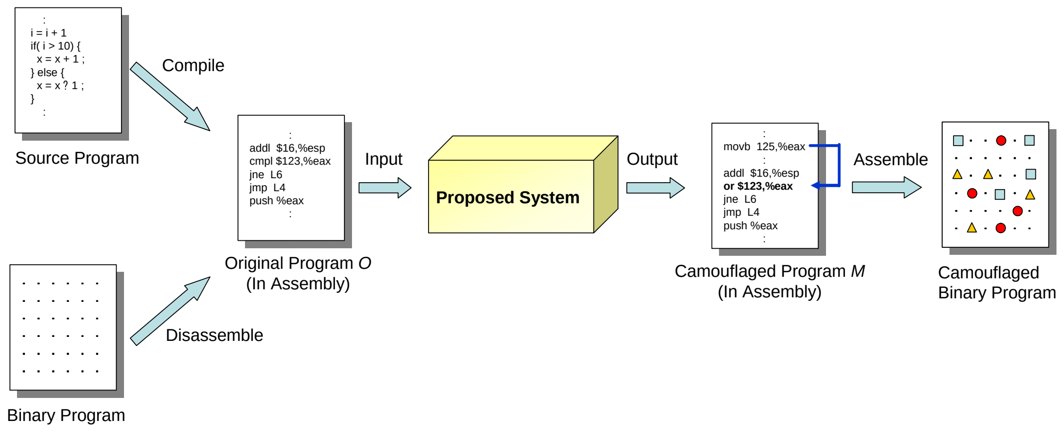
The *restoring routine* is a routine (a series of instructions) that rewrites an instruction camouflaged by the dummy instruction, thus restoring the original target instruction. The restoring routine that rewrites $dummy_i$ to $target_i$ is denoted as RR_i. The *hiding routine*, on the other hand, is a routine that rewrites the target instruction to the dummy instruction. The hiding routine that rewrites $target_i$ to $dummy_i$ is denoted as HR_i. The restoring routine and the hiding routine are together called *self-modification routines.*

A *camouflaged instruction* is an instruction whose content is changed (to $target_i$ or $dummy_i$) during execution. A *camouflaged program M* is an assembly program which contains camouflaged instructions.

3.2.4 Outline of the Proposed Method

Figure 3.3 shows an outline of the proposed method. First, a user (e.g., a program developer) who uses the proposed system prepares an assembly program (original code) O to be protected. This is normally obtained by compiling a source program or by disassembling a binary program. Then, the proposed system adds the self-modification mechanism to the assembly program, so that the original program becomes hard to be analyzed. Finally, assembling an assembly program M, which is the output of the system, the user can obtain a camouflaged program in binary that is functionally equivalent to the original one, but which is much more complex for attackers to analyze.

In the following sections, a systematic method is presented for deriving the camouflaged program M from the original program O.


Compile
Source Program
Input
Proposed System
Output
Assemble
Original Program O (In Assembly)
Camouflaged Program M (In Assembly)
Camouflaged Binary Program
Disassemble
Binary Program

```
  :
i = i + 1
if( i > 10) {
  x = x + 1 ;
} else {
  x = x ? 1 ;
}
  :
```

```
      :
addl  $16,%esp
cmpl $123,%eax
jne  L6
jmp  L4
push %eax
      :
```

```
      :
movb  125,%eax
      :
addl  $16,%esp
or $123,%eax
jne  L6
jmp  L4
push %eax
      :
```

Figure 3.3. An outline of the proposed method

3.2.5 Construction of Camouflaged Program M

M is constructed by the following steps 1 to 6.

(Step 1) Determination of the target instruction and the positions of the self-modification routines

$target_i$ and the positions of RR_i and HR_i in the program are determined. Below, the positions of RR_i and HR_i are denoted as $P(RR_i)$ and $P(HR_i)$, respectively.

First, $target_i$ is determined at random from among the instructions composing M. Or, the program developer may specify the target instruction directly. A control flow graph (directed graph) with each instruction in the assembly program is considered as a node. $P(RR_i)$ and $P(HR_i)$ are chosen so that the following four conditions are satisfied. The conditions are intended to assure that $dummy_i$ is certain to be rewritten as $target_i$ before it is executed, and is certain to be rewritten again as $dummy_i$ before the program ends.

[Condition 1] $P(RR_i)$ exists on any path from start to $target_i$.

[Condition 2] $P(HR_i)$ does not exist on any path from $P(RR_i)$ to $target_i$.

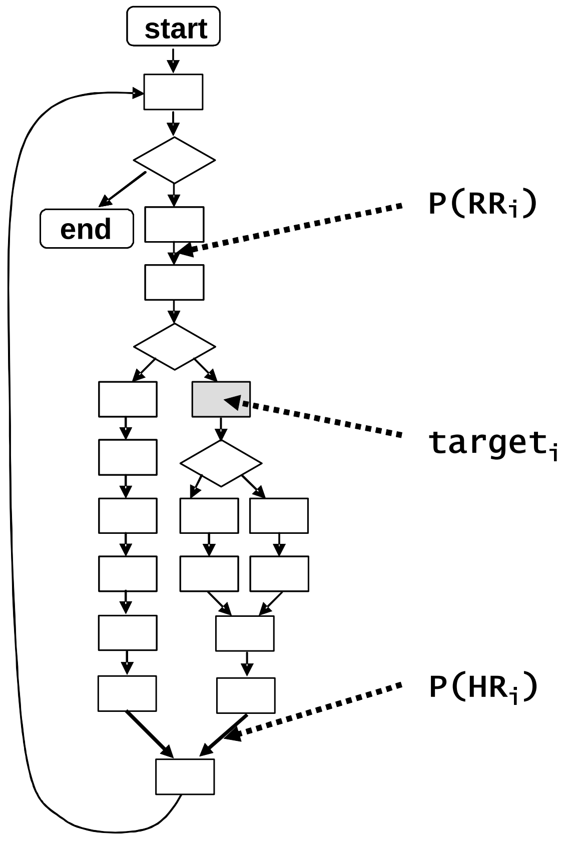


Figure 3.4. Example of $target_i$, $P(RR_i)$ and $P(HR_i)$ that satisfy four conditions

[Condition 3] $P(RR_i)$ exists on any path from $P(HR_i)$ to $target_i$.

[Condition 4] $P(HR_i)$ exists on any path from $target_i$ to the end of the program.

Figure 3.4 shows an example of $P(RR_i)$ and $P(HR_i)$ satisfying conditions 1 to 4.

Then, a procedure for choosing $P(RR_i)$ and $P(HR_i)$ satisfying conditions 1 to 4 is presented.

(1) The set T_u of paths (routes without node duplication) from *start* to $target_i$ is determined.

(2) In the nodes that are common to all paths $t \in T_u$ determined in (1), the set N_u whose incoming and outgoing orders are both 1 is determined. It is assumed that $target_i \notin N_u$. If $N_u = \emptyset$, define $target_i$ anew and go back to (1).

(3) A node $n_u \in N_u$ is selected at random. The incoming or outgoing edge of n_u is defined as $P(RR_i)$. Similarly,

(4) The set T_l of paths (routes without node duplication) from $target_i$ to *end* is determined.

(5) In the nodes contained in common to all paths $t \in T_l$ determined in (4), the set N_l of paths whose incoming and outgoing orders are both 1 is determined. It is assumed that $target_i \notin N_l$. If $N_l = \emptyset$, define $target_i$ anew, and go back to (1).

(6) A node $n_l \in N_l$ is selected at random. The incoming or outgoing edge of n_l is defined as $P(HR_i)$.

(Step 2) Determination of dummy instruction

An arbitrary instruction with the same instruction length as $target_i$ is selected and is defined as the dummy instruction $dummy_i$. An example is presented below in which the operation code composing $target_i$, or one of the operands, is modified for 1 byte, and is used as $dummy_i$. Consider the following $target_i$.

(Hex Representation)	`03 5D F4`
(Assembly Representation)	`addl -12(%ebp),%ebx`

By modifying operation code `03` to `33` in this $target_i$, the following $dummy_i$ is composed.

(Hex Representation)	`33 5D F4`
(Assembly Representation)	`xorl -12(%ebp),%ebx`

By modifying the operand `F4` to `FA` in $target_i$, the following $dummy_i$ is composed:

(Hex Representation)	`03 5D FA`
(Assembly Representation)	`addl -6(%ebp),%ebx`

(Step 3) Generation of self-modification routine

The self-modification routines RR_i and HR_i are generated by the following procedure.

(1) Label [2] L_i is inserted immediately before $target_i$. Using label L_i, $target_i$ can be referred to indirectly.

(2) Using L_i, a series of instructions for the rewriting of $dummy_i$ to $target_i$ is constructed and is defined as RR_i.

(3) Using L_i, a series of instructions for the rewriting of $target_i$ to $dummy_i$ is constructed and is defined as HR_i.

An example is presented below. `addl -12(%ebp), %ebx` is defined as $target_i$, and `xorl -12(%ebp), %ebx` is defined as $dummy_i$. Label `L1` is inserted into $target_i$.

```
L1:  addl -12(%ebp),%ebx
```

Next, RR_i is generated. RR_i has the function of modifying the first byte `33` of the instruction at `L1` to `03`:

[2] A label is a *name* in assembly language which indicates the position of the instruction (memory address) in the program.

```
    movb $0x03,L1
```

The effect of this small assembly routine composed of the above instruction is that the content of the address indicated by `L1` is to be overwritten by the immediate value `03` (hexadecimal). When RR_i is executed, $dummy_i$ is rewritten as $target_i$.

Similarly, HR_i is generated. HR_i has the function of modifying the first byte `03` of the instruction at `L1` to `33`.

```
    movb $0x33,L1
```

When HR_i is executed, $target_i$ is rewritten as $dummy_i$.

(Step 4) Write-in of dummy instruction and insertion of self-modification routine

The dummy instruction $dummy_i$ generated in Step 2 is written over $target_i$ determined in Step 1. By this process, the program before execution enters a state in which $target_i$ is camouflaged by $dummy_i$. Then the self-modification routines RR_i and HR_i which were generated in Step 3 are inserted into $P(RR_i)$ and $P(HR_i)$, respectively.

(Step 5) Complication of self-modification routine

The self-modification routine has the property that the address of $target_i$ in the program area is indicated by the label (immediate value address), and the content is rewritten. Consequently, there is a danger that an attacker may ascertain the position through the (static) analysis, and may identify the position of $target_i$.

Consider, for example, the case in which the movb instruction in the program contains the immediate address indicating the program area as the second operand. Then, that movb instruction may be inferred to be a self-modification routine.

In order to make static analysis difficult, the self-modification routine is complicated. For example, the fact that there is no write-in into the program area may be disguised by operating on the label. Or, the identification of the self-modification routine by the static pattern is made more difficult by the use of conventional techniques such as obfuscation of machine language instructions [42] and mutation [28]. An example of the modification of `movb $0x03,L1` is as follows:

```
movl $L1 + 1250, %eax
subl $1250, %eax
movb $0x03,(%eax)
```

L1 does not appear in the binary program obtained by assembling the above assembly routine (the value obtained by adding 1250 to L1 appears). This makes it difficult to identify the address L1 (the position of $target_i$) by static analysis. The address obtained by adding 1250 to L1 does not necessarily indicate the program area. Furthermore, the second operand of the movb instruction is not the immediate address, but the address indicated by the register %eax. It is difficult to determine its value by static analysis. By combining the above processing with obfuscation and mutation, an attack by pattern matching and address analysis will be made more difficult.

(Step 6) Iteration of above steps

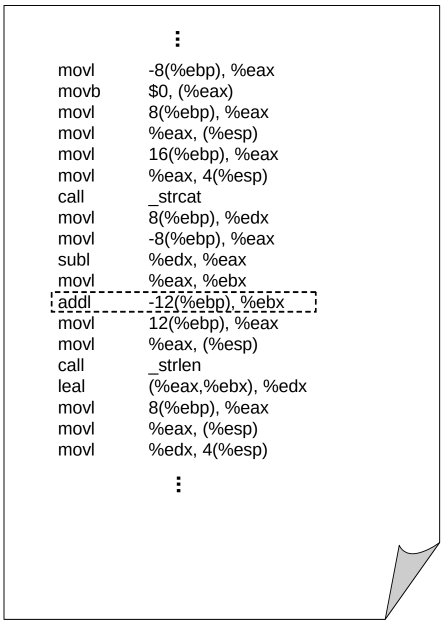
The processes from Step 1 to Step 5 are repeated. The number of camouflaged instructions is increased by each iteration. As will be discussed in 3.4, the increase in the number of camouflaged instructions and degradation of the execution efficiency are in a trade-off relation. It is thus desirable to specify the number of iterations with reference to the required degree of protection and the acceptable degradation of execution efficiency.

3.2.6 Construction Example of Camouflaged Program M

Figure 3.5 shows an example of a camouflaged program. (a) is the original program and (b) is the camouflaged program. The procedure for deriving (b) from (a) is as follows.

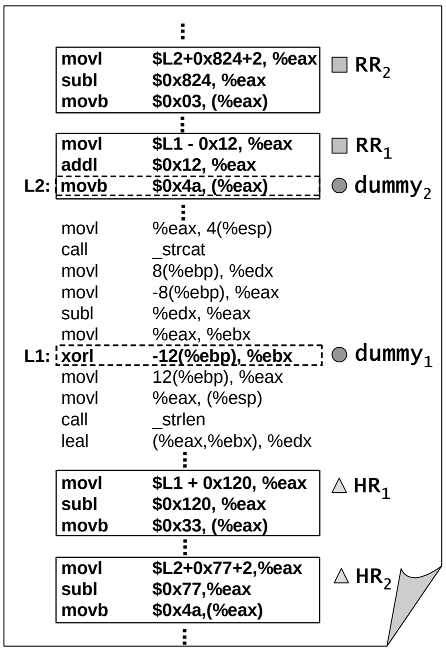
In the first camouflaging process, the instruction `addl -12(%ebp), %ebx` in the dotted frame in Figure 3.5(a) is selected as $target_1$, and is overwritten by $dummy_1$ (`xorl -12(%ebp), %ebx`), as shown in Figure 3.5(b). Then, the self-modification routines RR_1 and HR_1 for $target_1$ are generated and inserted. In the second camouflaging process, one of the instructions composing RR_1 (`movb $0x03, (%eax)` at the end of the first camouflaging process) is selected as $target_2$ and is overwritten by $dummy_2$ (`movb $0x4a, (%eax)`) as shown in Figure 3.5(b). Then, the self-modification routines RR_2 and HR_2 for $target_2$ are generated and inserted.

In this case, part of RR_1 is rewritten by $dummy_2$. Consequently, in order to ascertain the original instruction for $dummy_1$, not only RR_1 but also RR_2 must be found. In the Appendix A, a simple program containing a conditional branch is presented, together with a listing of the camouflaged program.



```
                  ⋮
movl        -8(%ebp), %eax
movb        $0, (%eax)
movl        8(%ebp), %eax
movl        %eax, (%esp)
movl        16(%ebp), %eax
movl        %eax, 4(%esp)
call        _strcat
movl        8(%ebp), %edx
movl        -8(%ebp), %eax
subl        %edx, %eax
movl        %eax, %ebx
addl        -12(%ebp), %ebx
movl        12(%ebp), %eax
movl        %eax, (%esp)
call        _strlen
leal        (%eax,%ebx), %edx
movl        8(%ebp), %eax
movl        %eax, (%esp)
movl        %edx, 4(%esp)
                  ⋮
```

(a) Original Program



```
                  ⋮
    movl        $L2+0x824+2, %eax
    subl        $0x824, %eax
    movb        $0x03, (%eax)
                  ⋮
    movl        $L1 - 0x12, %eax
    addl        $0x12, %eax
L2: movb        $0x4a, (%eax)
                  ⋮
    movl        %eax, 4(%esp)
    call        _strcat
    movl        8(%ebp), %edx
    movl        -8(%ebp), %eax
    subl        %edx, %eax
    movl        %eax, %ebx
L1: xorl        -12(%ebp), %ebx
    movl        12(%ebp), %eax
    movl        %eax, (%esp)
    call        _strlen
    leal        (%eax,%ebx), %edx
                  ⋮
    movl        $L1 + 0x120, %eax
    subl        $0x120, %eax
    movb        $0x33, (%eax)
                  ⋮
    movl        $L2+0x77+2,%eax
    subl        $0x77,%eax
    movb        $0x4a,(%eax)
                  ⋮
```

(b) Camouflaged Program

Figure 3.5. Example of a camouflaged program

3.3. Discussion of Difficulty of Analysis

3.3.1 Assumed Analysis Procedure

Consider the case in which the attacker described in 3.2.1 attempts an analysis of the secret part $C(M)$ of M. The following attack procedure is assumed. The goal of the analysis is defined as understanding $C(M)$ correctly. In order to understand $C(M)$ correctly, the original instruction corresponding to each of the dummy instructions contained in $C(M)$ must be ascertained. For this purpose, the restoring routine for each dummy instruction contained in $C(M)$ must be found from the whole program.

There are two methods of analysis that the attacker can apply, *static analysis* and *dynamic analysis*. Static analysis is a method of analysis without running the program which is the object of analysis. A typical approach is to restrict the range of $C(M)$ by keyword search, pattern matching, and other techniques, so as to understand $C(M)$. Since the analysis is concentrated on $C(M)$ without considering the whole of program M, the cost of analysis is generally lower than that of the dynamic analysis discussed later, and the method is widely applied. The first objective of the proposed method is to make static analysis difficult.

On the other hand, dynamic analysis is performed while running the program which is the object of analysis. The attacker runs M using tools such as a debugger, and tries to identify and understand $C(M)$ based on the output information from the tool. By dynamic analysis, the attacker can completely track the execution of M. However, since the analysis depends on the input and the whole program M must be run, the cost of analysis increases very rapidly as the scale of M is enlarged.

Furthermore, debugging information is generally deleted from commercial programs, or features such as the inhibition of unintentional execution are included.

Consequently, it is not necessarily true that dynamic analysis can be applied to any program. It is possible at relatively low cost to preserve a snapshot at an arbitrary point of the executed program, that is, the content of the object program loaded into memory at any point during execution, in order to facilitate static analysis. For this purpose, there must be a mechanism to prevent the invalidation of protection even if several snapshots are acquired.

The next section discusses the security of M against each method of analysis.

3.3.2 Security against Static Analysis

In order to investigate the security of M against static analysis, the probability that the attacker can correctly understand the secret part $C(M)$ is formulated.

Consider the situation in which M contains only one dummy instruction $dummy_i$. In order for the attacker to correctly understand an arbitrary code block $D(M)$ of length m in M, the following event E_i must apply.

> E_i: $dummy_i$ does not exist in $D(M)$, or $dummy_i$ exists in $D(M)$ and RR_i exists in $D(M)$.

When $dummy_i$ does not exist in $D(M)$ (i.e., there is no camouflage at all), the attacker can directly track $D(M)$ and can easily understand the original behavior of $D(M)$. When $dummy_i$ is present in camouflaged form in $D(M)$, but its restoring routine RR_i also exists in $D(M)$, $target_i$ can be identified by analysis of RR_i, and the original behavior of $D(M)$ can be discovered.

Let the number of instructions in M be L. When $dummy_i$ and RR_i are selected at random in M, the probability $P(E_i)$ that E_i is valid is expressed as follows:

$$P(E_i) \quad = \quad \frac{L-m}{L} + \frac{m}{L} \times \frac{m}{L}$$

$$= \frac{(L-m)^2 + Lm}{L^2} \tag{3.1}$$

Then, consider the case in which n dummy instructions ($dummy_1$, ... , $dummy_n$) are contained in M, and E_i must be valid for any i ($1 \leq i \leq n$). The probability $P(Success, D)$ that the analysis of $D(M)$ succeeds is roughly expressed as follows:

$$P(Success, D) = \Big(\frac{(L-m)^2 + Lm}{L^2}\Big)^n \tag{3.2}$$

Figure 3.6 shows the curve representing the relation between $P(Success, D)$ and n. The horizontal axis is the number of camouflaged instructions n in M, and the vertical axis is the probability of success of code analysis $P(Success, D)$. The number of instructions m in $D(M)$ is set as 100. The number of instructions in M is varied as 1000, 2000, and 3000. The result for each of these is shown. It is evident from Figure 3.6 that as the number of dummy instructions n is increased (and thus the extent of camouflage is raised), the probability of success of code analysis for $D(M)$ approaches 0.

When the secret part $C(M)$ agrees with (or is contained in) the code block $D(M)$ which is arbitrarily selected by the attacker, this implies that the static analysis of M is a success. Since the identification of $C(M)$ depends on the skill of the attacker, formulation by the theory of probability is difficult. Letting, as an assumption, the probability that $C(M)$ is contained in $D(M)$ be X, the probability of successful analysis $P(Success)$ is expressed as

$$\begin{aligned} P(Success) &= X \times P(Success, D) \\ &= \Big(\frac{(L-m)^2 + Lm}{L^2}\Big)^n X \end{aligned} \tag{3.3}$$

Based on the above formulation, it is evident that in order to increase the probability of successful static analysis by the attacker, it is necessary for him

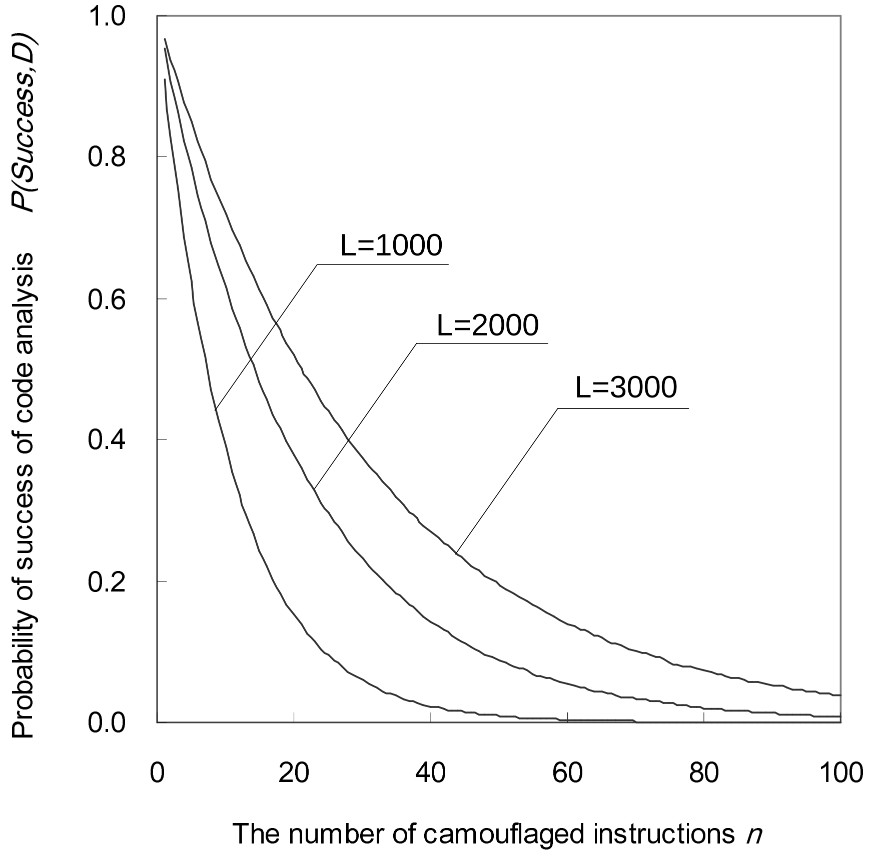


Figure 3.6. Probability of success of code analysis ($m = 100$)

either to increase X by skillfully locating $C(M)$, or to enlarge the size m of the analyzed part $D(M)$. On the other hand, the user of the proposed method can easily control $P(Success)$ by increasing the number of camouflaged instructions n.

In the above discussion, $P(Success)$ increases with the size of L. This is because $dummy_i$ is chosen at random in the formulation of the event E_i, which can prevent $dummy_i$ from being contained in $C(M)$. However, when the user knows the position of $C(M)$ beforehand, $P(E_i)$ can be decreased by inserting $dummy_i$ into $C(M)$, or by increasing the distance between RR_i and $dummy_i$ so that it is larger than the expected m. Thus, the probability of success of code analysis can be decreased.

On the other hand, when the user does not know exactly the position of $C(M)$, or wants to decrease the probability X that the attacker can locate $C(M)$, it will be effective to divide M into L/m blocks and to insert a constant number of camouflaged instructions in each block. This measure makes the analysis difficult, no matter which block $D(M)$ the attacker subjects to analysis, since the camouflaged instructions are distributed uniformly.

3.3.3 Security against Dynamic Analysis

When M is stopped at a point during execution with a debugger, some of the dummy instructions in $C(M)$ may be in the state of being rewritten as the original instructions. If the attacker acquires a snapshot and observes the part corresponding to $C(M)$ in the program loaded in memory, some of the original instructions can be determined. This poses a danger that $C(M)$ may be correctly understood.

However, in this process it is difficult to know the original content of all dummy instructions present in $C(M)$. The reason is as follows. Since the restoring routine used to rewrite the dummy instruction in $C(M)$ is scattered over the whole program, various parts of the program must be executed in order to execute all of these routines. Unless the whole program is understood, this process has a high cost. Furthermore, when the hidden routine is executed, the instruction that was present in the original content is again overwritten by the dummy instruction. Consequently, even at the point immediately before the end of the program, the attacker cannot acquire a snapshot in which most of the instructions have been restored to the original instructions.

However, especially when fewer dummy instructions are present in $C(M)$, dynamic analysis can be an effective mode of attack. Consequently, it is desirable to use other techniques to make dynamic analysis difficult by preventing the operation of a debugger that uses interrupts and other instructions [8]. This will improve the security of M against dynamic analysis.

3.4. Case Studies

3.4.1 Outline

This section describes the measurement process and the results for the following three items when the proposed method is applied to software.

(1) The distance between the target instruction and the restoring routine

(2) The change of the file size (size overhead)

(3) The change of the execution time (performance overhead)

The tool `ccrypt` was used, which encrypts and decrypts files, as the software with which to test the proposed method [60]. This program is open-source software under the GPL license.

The authors experimentally constructed a system in which the program was camouflaged by the proposed method [31]. Using that system, the proposed method was applied to the target program by the following procedure.

(1) The source file $s_1, s_2, \dots, s_n$ in the C language was compiled and the original assembly file $a_1, a_2, \dots, a_n$ was obtained.

(2) Each of $a_1, a_2, \ldots, a_n$ was camouflaged, and the camouflaged assembly file $a'_1, a'_2, \ldots, a'_n$ was obtained.

(3) $a'_1, a'_2, \ldots, a'_n$ were assembled and the execution modules $o_1, o_2, \ldots, o_n$ were obtained.

(4) $o_1, o_2, \ldots, o_n$ are linked and the executable file p is obtained.

In each trial, it is verified that the executable file p operates correctly.

In executable files running under Windows (such as the Microsoft Portable Executable format), enabling/disabling of writing to the code area is controlled by a flag in the section header in the file [39]. When the proposed method is applied, it is necessary to make the code area rewritable during execution by setting the flag beforehand.

The computer used in the experiment had Windows XP as the OS, a main memory size of 512 Mbytes, and a Pentium 4 CPU (clock frequency 1.5 GHz, primary trace cache 12kμOps, primary data cache of 8 kbytes, and the secondary cache of 256 kbytes).

3.4.2 Distance between Target Instruction and Restoring Routine

Figure 3.7 shows the distribution of the dummy instruction and the restoring routine for a camouflaged assembly language file. The file has 1490 lines and 947 instructions before camouflaging. One hundred thirty instructions are camouflaged. The vertical axis of the figure is the line number, and the horizontal axis is the line number modulo 30. By adding the value on the horizontal axis to the value on the vertical axis, the line number containing the instruction or

Table 3.1. Distance between target instructions and restoring routines

	Average	Maximum	Minimum	Standard Deviation
Distance[instructions]	151	611	1	192

the restoring routine is obtained. It is evident from Figure 3.7 that the target instructions and restoring routines are scattered over the whole program.

Table 3.1 shows the average, the maximum, the minimum, and the standard deviation of the distance between the target instruction and the restoring routine. It can be seen from Table 3.1 that in order to ascertain whether an instruction in the program is a camouflaged instruction, the restoring routine, which is at a distance of 151 instructions away on average and 611 instructions away at the maximum, must be located. Since this program is camouflaged at a rate of 1 instruction in each 7 instructions, a large number of camouflaged instructions will be encountered in the search for the restoring routine. It can also happen, as was discussed in the example in 3.2.6, that instructions constituting the restoring routines are themselves camouflaged. Thus, it is likely that the cost of the analysis intended to find the restoring routine will be high.

Since the minimum distance is 1 instruction, it can be seen that there is a case in which the target instruction is adjacent to the restoring routine. Since the position of the target instruction and the insertion position of the restoring routine are selected at random from the candidates, such a case can occur.

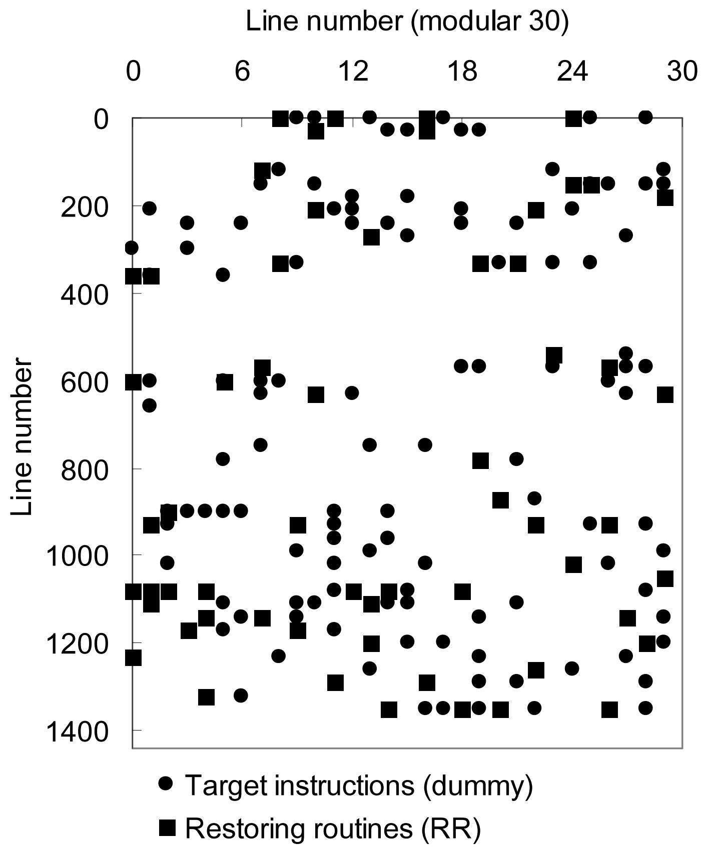


Figure 3.7. Distribution of target instructions and restoring routines

3.4.3 Size Overhead

Examining the file size of the camouflaged program, it can be seen that the file size increases in proportion to the number of camouflaged instructions. On average, each time the number of camouflaged instructions is increased by 100, the file size is enlarged by approximately 2.4 kbytes. This increase in file size is due to the increase in the number of inserted self-modification routines as the number of camouflaged instructions is increased.

Noting that the capacity of secondary memory devices is currently increasing, the enlargement of file size will not be a serious problem. However, in an environment where the file size is severely limited it may happen that the increase of the file size must be minimized. Dealing with such a situation is made possible by adjusting the number of camouflaged instructions so that the file size stays within the permissible range.

3.4.4 Performance Overhead

The time required for the camouflaged `ccrypt` to encrypt a 100-kbyte text file was measured 10 times in each session while varying the number of camouflaged instructions. The execution time was measured as the difference in the elapsed time of the system clock from immediately before the start of the camouflaged program to immediately after the termination of the program. The elapsed time of the system clock was acquired by using the `clock` function in C.

Figure 3.8 shows a plot of the results of the execution time measurement. The horizontal axis shows the number of camouflaged instructions, while the vertical axis shows the average program execution time and the proportion of camouflaged instructions (depicted by bars).

It can be seen from Figure 3.8 that the average execution time increases with the number of camouflaged instructions. When 500 instructions are camouflaged (when approximately 5% of the entire instructions are camouflaged), the average execution time is approximately 2.9 seconds. This is approximately 48 times the execution time (approximately 0.06 second) when no instruction is camouflaged. Three possible reasons for this increase in execution time exist.

(1) Inserting self-modification routines increases the number of instructions to be executed.

(2) Each time a self-modification routine writes code cached in the CPU, the

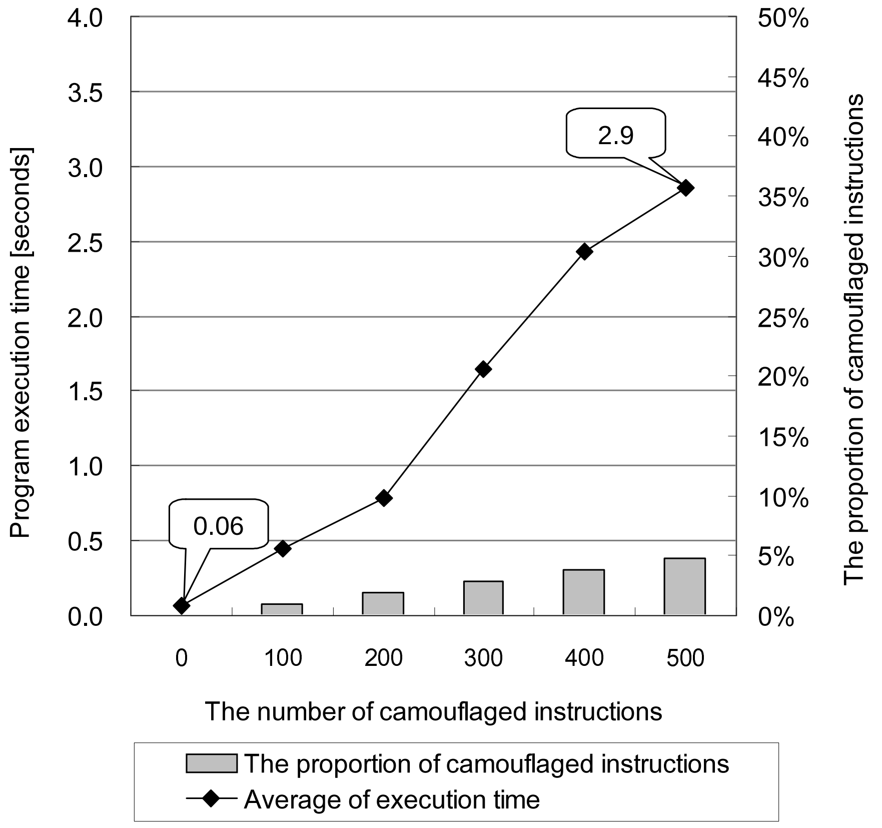


Figure 3.8. Impact on program execution time

corresponding cache line is invalidated [59].

(3) The self-modification mechanism increases the frequency of prediction failure of conditional branches in the CPU.

Increased execution time may or may not be a disadvantage. Excessive camouflaging is not recommended, for example, for an algorithm that considers the next move in a game such as shogi or chess, or for an algorithm which must operate in real time, such as a streaming playback routine for speech.

For a program in which the user is restricted by means of password authentication, on the other hand, the proposed method may be used in order to complicate the analysis of the password check routine. In such a case, if the method is applied only to the password checking part of the program, the original functioning is not degraded, except that a longer time is required for the password check. Location and degree of the camouflage should be chosen according to the properties and purpose of the program or module to which the proposed method is applied.

In addition, the probability of success of code analysis, which is formulated in 3.3.2, can help determine the degree of camouflage. Figure 3.9 shows the relationship between the probability of success of code analysis and performance overhead (about the same program). The horizontal axis shows the proportion of camouflaged instructions, while the vertical axis shows the average program execution time (depicted by the solid line) and the probability of success of code analysis (depicted by dotted lines). m/L (the proportion of instructions to be analyzed) is varied as 1%, 3% and 5%.

As can be seen, in this graph, the probability of success of code analysis decreases as the program execution time increases. Grasping this trade-off relation between the level of protection against static analysis and performance overhead can be useful in finding the optimal degree of camouflage.

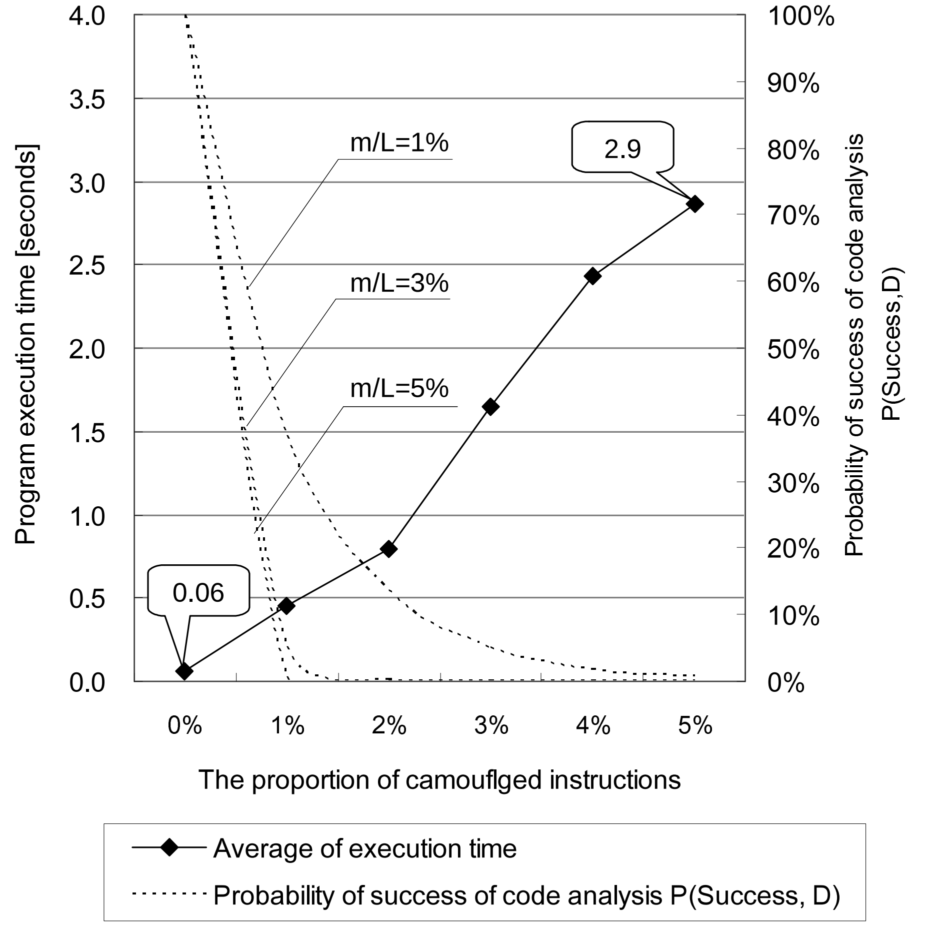


Figure 3.9. Program execution time and probability of success of code analysis (m/L = 1%, 3%, 5%)

3.5. Related Work

The self-modification mechanism itself has long been known. One of its purposes is to reduce the program size and the required memory capacity in execution [26]. Another purpose is to protect programs, as in this study, in which a program is encrypted and decrypted by self-modification [2, 4–6, 12, 17, 23, 24, 29, 48, 55]. In the latter case, the specified range of the program is encrypted beforehand and is decrypted by self-modification during execution. The instructions are encrypted again if necessary. This approach, which is called *software encryption*, is similar to the proposed method in the sense that the program is rewritten during execution, but differs in the following respects:

- Identifying the position of a camouflaged instruction by static analysis is not easy, because a camouflaged instruction cannot be distinguished from other instructions. On the other hand, the range of the part of the program which is encrypted may be easily identified, since it has features different from the other parts (such as the absence of instruction sequences and impossibility of disassembly).

- The restoring routine used to resolve the camouflage is a very ordinary short routine in main memory, with a length of 1 to several bytes. In addition, since such routines are scattered through the program, identifying their positions by static analysis is not easy. On the other hand, a routine for decryption has a large size, and hiding the identification cue is not easy. In particular, when the whole program is encrypted, the decryption starts immediately at the beginning of program execution, which makes identifying the routine for decryption easy.

The approach in which an instruction which has been overwritten by a different content beforehand is replaced by the original instruction at the time of execution has been considered in the past for software protection. However, in the past approach, an operator with sufficient knowledge, technique, and assembler resources, had to handle the protection process manually. In contrast, this chapter proposes and evaluates a systematic (formulated) method of protecting software by self-modification. The proposed method is easy to automate and to flexibly adjust the trade-off (degree of camouflage) between the degree of protection and the overhead.

The instruction code of the program that is protected by a method using the self-modification mechanism (such as software encryption methods and the proposed method) can be modified by any other executable program at run-time, since the protected program needs its *code section* to be modified for self-modification to be done. Although that restriction is normally not a disadvantage, applying the methods to a program that is running all the time (e.g., a server program) is not recommended since there is a possibility that the protected program may be attacked by malware (e.g., viruses, worms) at run-time.

Other methods of making program analysis difficult have included many methods of *software obfuscation* [13–16,27,42,46,50,63,64]. Obfuscation is a technique in which a given program is converted to a program which is more difficult to analyze (more complex) without modifying its specifications. The behavior of the obfuscated program can be understood correctly, even if only partially, by spending a long time analyzing the program. On the other hand, in a program which is protected by the proposed method, it is difficult to determine, even partially, when an instruction with different content from the original instruction is present (unless the restoring routine has been identified), even if a long time is spent. This is the difference between obfuscation and the proposed method.

The proposed method is a technique which is not controversial with respect to encryption or obfuscation. Consequently, the analysis of the program is made still more difficult by combining the proposed method with these approaches. When the method is combined with obfuscation, for example, the result is a program which cannot be analyzed successfully unless the following two stages are accomplished.

(1) The uncamouflaged state is obtained.

(2) The obfuscated program is understood.

It should be noted, however, that the program size or the execution time may be further increased by the combined use of these approaches.

Chapter 4

Conclusion

4.1. Achievements

In this dissertation, research on preventing secret information in software processes and software products from being revealed to users was addressed. Two cases were considered where secret information was revealed: (1) through work products leaked by insiders who take part in a software development process, and (2) by reverse engineering of software products. Case (1) occurs *before* a software product is released, while case (2) occurs *after* a software product is released. The method for preventing secret information in each case has been tackled successfully.

First, in Chapter 2, a method for evaluating the risk of information leakage in the software development process was presented. This method is useful for preventing secret information in case (1). Leakage was formulated as an unexpected transfer of product knowledge among developers sharing the same process. Next, a method was proposed to derive the probability that each developer will know each work product at any process of software development.

Three case studies were also conducted. The result of the first case study quantitatively showed that, more collaboration among authorized and unauthorized developers causes a higher risk of information leakage. In the second case study, the proposed method was also shown to be useful as a powerful means to perform optimal tunings of the process configuration. Finally, in the third case study, the structure of the process was shown to control the execution order of processes, which in turn, significantly influences the risk of information leakage.

Next, in Chapter 3, a method for increasing the cost of reverse engineering attacks, which aims to prevent secret information in case (2) was proposed. In particular, a systematic method of making program analysis difficult by camouflaging instructions was proposed. When an attacker attempts a static analysis of a part of the program which contains camouflaged instructions, he/she cannot understand the original behavior of that part correctly unless he/she locates the restoring routine.

To investigate the difficulty of analysis of the camouflaged program, the probability that the attacker can correctly understand the secret part of the program was analyzed. Based on the resulting equation, it was concluded that the more the instructions are camouflaged, the more the probability of success of code analysis decreases.

As a case study, a program (`ccrypt`) was camouflaged and the distance between the target instructions and the restoring routine, the file size, and the execution time overhead was measured. When 130 instructions in 947 were camouflaged, the average distance between the target instruction and the restoring routine was 151 instructions. Since many camouflaged instructions may exist between the target instruction and the restoring routine, it is likely that a costly analysis will be required to find the restoring routine. As regards overhead, it was found that the file size and the overhead of execution time are increased with an increasing number of camouflaged instructions. Choosing the position and degree of camouflage according to the properties and purposes of the program or module to which the method is applied is desirable.

Prototypes of each proposed method have been implemented. It is assumed that the prototype tool for evaluating the risk of information leakage is used by a developer who designs a software development process (e.g., software process analyst, software process designer) to construct a secure software development process, before the development process is performed. On the other hand, it is assumed that the prototype tool for increasing the cost of reverse engineering is used by a developer who designs or implements a software product (e.g., code designer, programmer), after the implementation and the test of the software is done.

4.2. Future Research

Some issues remain for future study.

The proposed method for evaluating the risk of information leakage is simple and generic; therefore, the method should not be limited to the security-sensitive software process. The method is highly feasible for other workflow-based applications, such as *medical work flows* [56] where private information must be protected. In addition, more practical methods for calculating $leak(u, w, u')$, which characterize the probability that a developer u leaks the product knowledge w to another developer u', are left as a challenging issue for future work.

As for the method of increasing the cost of reverse engineering attacks based on instruction camouflage, improving the system so that the execution time overhead will be reduced in order to make the proposed method capable of wider application is planned. Specifically, improving the algorithm for determining the inserting position of self-modification routines are planned as follows:

- Determining the inserting position of a self-modification routine considering the execution frequency of the position. Loop structure of the target program and an estimate of execution frequency of each block of the program are statically analyzed, to avoid inserting self-modification routines in the frequently-executed blocks.

- Determining the positions of self-modification routines considering the effectiveness of CPU pipeline. Since a self-modification routine interrupts branch prediction, the distance between self-modification routines influences the effectiveness of the CPU pipeline. A routine is inserted so that the routine will be a certain distance apart from the other routines.

Acknowledgements

First and foremost, I wish to express my sincere gratitude to my supervisor Professor Ken-ichi Matsumoto, for his continuous support and encouragement during this work.

I am also very grateful to the members of my thesis review committee: Professor Katsumasa Watanabe and Associate Professor Yuichi Kaji, for their valuable comments and helpful criticism of this work.

I am deeply grateful to Associate Professor Akito Monden. He gave me the opportunity to study in the field of software protection. For five years, I have received invaluable assistance from him. I will always remember his encouragement and enthusiasm.

I also want to thank Assistant Professor Masahide Nakamura, for his patient advice and guidance. I have learned a lot from his knowledge and positive attitude toward research. I could not have finished this dissertation without his encouragement.

I wish to thank Professor Hajimu Iida. I have received helpful advice and warm support from him for five years.

I also wish to thank Assistant Professor Masao Ohira. His comments and advice were very helpful in the completion of this dissertation.

I have been fortunate to have received assistance from many colleagues. I wish to thank all the members of the Software Engineering Lab., Graduate School of Information Science, Nara Institute of Science and Technology. I can only mention a few of my helpful colleagues here because the list is long. I wish to extend thanks to, Hiroshi Igaki, Haruaki Tamada, Naoki Ohsugi, Hiroki Yamauchi, Susumu Kuriyama, and Hidetaka Uwano.

Finally, I would like to express my warmest gratitude to my parents, my grandmother, my sister, and my friends for their constant encouragement and generous remarks.

The research was supported by Grant-in-Aid for 21st century COE Research (NAIST-IS Networked Media Computing).

References

[1] 4C-Entity. *Policy statement on use of content protection for recordable media (CPRM) in certain applications*, 2001. (Available online).

[2] D. J. Albert and S. P. Morse. Combating software piracy by encryption and key management. *IEEE Computer*, pages 68–73, April 1984.

[3] NPO Japan Network Security Association. Security incident report. http://www.jnsa.org/.

[4] D. W. Aucsmith. *Tamper resistant software: An implementation*, volume 1174 of *Lecture Notes in Computer Science*, pages 317–333. Springer-Verlag, 1996.

[5] D. W. Aucsmith and G. L. Graunke. Tamper resistant methods and apparatus. Assignee: Intel Corporation 5,892,899, United States Patent, April 1999.

[6] R. M. Best. Crypto microprocessor for executing enciphered programs. Technical Report 4,278,873, United States Patent, July 1981.

[7] J. Brockmeier. Conrante Tech News. Microsoft's code leakage. http://openmind.corante.com/.

[8] P. Cervan. *Crackproof your software.* No Starch Press, San Francisco, 2002.

[9] H. Chang and M. Atallah. Protecting software codes by guards. In *Proc. Workshop on Security and Privacy in Digital Rights Management 2001*, volume 2320 of *Lecture Notes in Computer Science*, pages 160–175. Springer-Verlag, 2001.

[10] S. Chou, A. Liu, and C. Wu. Preventing information leakage within workflows that execute among competing organizations. *The Journal of Systems and Software*, 2004. (Available online).

[11] S. Chow, P. Eisen, H. Johnson, and P.C. van Oorschot. A white-box DES implementation for DRM applications. In *Proc. 2nd ACM Workshop on Digital Rights Management*, pages 1–15, November 2002.

[12] F.B. Cohen. Operating system protection through program evolution. *Computers and Security*, 12(6):565–584, 1993.

[13] C. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation – tools for software protection. *IEEE Transactions on Software Engineering*, 28(8):735–746, June 2002.

[14] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical Report 148, Technical Report of Dept. of Computer Science, U. of Auckland, New Zealand, 1997.

[15] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *Proc. IEEE International Conference on Computer Languages(ICCL'98)*, Chicago, May 1998.

[16] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages(POPL98)*, pages 184–196, San Diego, California, January 1998.

[17] C. N. Drake. Computer software authentication, protection, and security system. Technical Report 6,006,328, United States Patent, December 1999.

[18] M. Fagerholm. IT Manager's Journal. Managing the insider threat through code obfuscation. http://software.itmanagersjournal.com/.

[19] P. H. Feiler and W. S. Humphrey. Software process development and enactment: Concepts and definitions. In *Proc. 2nd International Conference on Software Process*, pages 28–40, February 1993.

[20] D. Ferraiolo and R. Kuhn. Role-based access controls. In *Proc. 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[21] S. Funamoto. *Anatomy of protection technology.* Subarusha, 2002. (in Japanese).

[22] P. K. Garg and M. Jazayeri. *Process-centered software engineering environments.* IEEE Computer Society Press, 1995.

[23] D. Grover, editor. *The protection of computer software: Its technology and applications.* Cambridge University Press, 1989.

[24] B. E. Hampson. Digital computer system for executing encrypted programs. Assignee: Prime Computer, Inc. 4,847,902, United States Patent, July 1989.

[25] M. Hashimoto, K. Yamaguchi, and H. Isozaki. Software protection in open software environment. *Toshiba Review*, 58(6):20–23, June 2003. (in Japanese).

[26] T. Hidaka. *Mysteries of the Z80 machine.* Keigaku Shuppan, 1989. (in Japanese).

[27] F. Hohl. *Time limited blackbox security: Protecting mobile agents from malicious hosts*, volume 1419 of *Lecture Notes in Computer Science*, pages 92–113. Springer-Verlag, 1998.

[28] J. Irwin, D. Page, and N.P. Smart. Instruction stream mutation for non-deterministic processors. In *Proc. ASAP2002*, pages 286–295, July 2002.

[29] H. Ishima, K. Saito, M. Kamei, and K. Shin. Tamper resistant technology for software. *Fuji Xerox Technical Report*, (13):20–28, 2000.

[30] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[31] Y. Kanzaki. RINRUN: Program camouflage tool. http://se.naist.jp/rinrun/.

[32] Y. Kanzaki, H. Igaki, M. Nakamura, A. Monden, and K. Matsumoto. Quantitative analysis of information leakage in security- sensitive software processes. *IPSJ Journal, Special Issue on Research on Computer Security Characterized in the Context of Social Responsibilities*, 46(8):2129–2141, August 2005.

[33] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting self-modification mechanism for program protection. In *Proc. 27th IEEE Computer Software and Applications Conference*, pages 170–179, Dallas, USA, November 2003.

[34] Y. Kanzaki, A. Monden, M. Namamura, and K. Matsumoto. A software protection method based on instruction camouflage. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences (Japanese Edition)*, J87-A(6):755–767, June 2004. (In Japanese).

[35] F. Keller, P. Tabeling, R. Apfelbacher, B. Grone, A. Knopfel, R. Kugel, and O. Schmidt. Improving knowledge transfer at the architectural level: Concepts and notations. In *Proc. The 2002 International Conference on Software Engineering Research and Practice*, June 2002.

[36] R. Lemos. CNET News. Microsoft cracks down on source code traders. http://news.com.com/.

[37] R. Lemos. CNET News. Microsoft probes Windows code leak. http://news.com.com/.

[38] R. Lemos and I. Fried. CNET News. Search on for source of leaked Windows code. http://news.com.com/.

[39] J.R. Levine. *Linkers & loaders*. Morgan Kaufmann, 2001.

[40] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proc. 10th ACM Conference on Computer and Communications Security*, pages 290–299, October 2003.

[41] Y. Liong and S. Dixit. Digital rights management for the mobile internet. *Wireless Personal Communications*, 29(1-2):109–119, 2004.

[42] M. Mambo, T. Murayama, and E. Okamoto. A tentative approach to constructing tamper-resistant software. In *Proc. 1997 New Security Paradigm Workshop*, pages 23–33, September 1997.

[43] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. John Wiley, 1995.

[44] A. Monden, A. Monsifrot, and C. Thomborson. Obfuscated instructions for software protection. Information science technical report, NAIST-IS-TR2003013, Graduate School of Information Science, Nara Institute of Science and Technology, November 2003.

[45] A. Monden, A. Monsifrot, and C. Thomborson. Tamper-resistant software

system based on a finite state machine. *IEICE Transactions on Fundamentals*, E88-A(1):112–122, January 2005.

[46] A. Monden, Y. Takada, and K. Torii. Methods for scrambling programs containing loops. *IEICE Transactions on Information and Systems, PT.1 (Japanese Edition)*, J80-D-I(7):644–652, July 1997. (in Japanese).

[47] Monthly Information Security. Database of information leakage incidents. http://www.monthlysec.net/ (in Japanese).

[48] J. M. Nardone, R. P. Mangold, J. L. Pfotenhauer, K. L. Shippy, D. W. Aucsmith, R. L. Maliszewski, and G. L. Graunke. Tamper resistant methods and apparatus. Assignee: Intel Corporation 6,178,509, United States Patent, January 2001.

[49] G. Naumovich and N. Memon. Preventing piracy, reverse engineering, and tampering. *IEEE Computer*, 36(7):64–71, July 2003.

[50] T. Ogiso, Y. Sakabe, M. Soshi, and A. Miyaji. Software obfuscation on a theoretical basis and its implementation. *IEICE Transaction on Fundamentals*, E86-A(1):176–186, 2003.

[51] H. Okamura. *The latest cases on the cyber law.* Softbank Publishing Co., 2000. (in Japanese).

[52] A. Orlowski. The Register. iTunes DRM cracked wide open for GNU/Linux seriously. http://www.theregister.co.uk/.

[53] The United Kingdom Parliament. The mobile telephones (re-programming) bill. Technical Report 02/47, House of Commons Library Research Paper, July 2002.

[54] A. Patrizio. Wired News. DVD piracy: It *can* be done. http://www.wired.com/.

[55] W. Paulini and D. Wessel. Process for securing and for checking the integrity of the secured programs. Assignee: Siemens nixdorf informations system 5,224,160, United States Patent, June 1993.

[56] S. Quaglini, C. Mossa, C. Fassino, M. Stefanelli, A. Cavallini, and G. Micieli. *Guidelines-based workflow systems*, volume 1620/1999 of *Lecture Notes in Computer Science*, pages 65–75. Springer-Verlag, 1999.

[57] P. Samuelson. Reverse-engineering someone else's software: Is it legal? *IEEE Software*, 7(1):90–96, January 1990.

[58] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[59] *IA-32 Intel Architecture software developer's manual vol.3 System Programming Guide*. Intel Co. http://www.intel.co.jp/.

[60] P. Selinger. ccrypt (utility for encrypting and decrypting files and streams). http://ccrypt.sourceforge.net/.

[61] The Mainichi Newspapers. Firms struggling to plug customer information leaks, 2004. The Mainichi Newspapers, March 2.

[62] R. K. Thomas and R. S. Sandhu. Task-based authorization controls (TBAC): A family of models for active and enterprise-oriented autorization management. In *Proc. the IFIP Workshop on Database Security*, pages 166–181, August 1997.

[63] P. M. Tyma. Method for renaming identifiers of a computer program. Assignee: PreEmptive Solutions, Inc. 6,102,966, United States Patent, August 2000.

[64] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obfuscating static analysis of programs. Technical report sc-2000-12, Department of Computer Science, University of Virginia, December 2000.

[65] H. Yamada and J. Kawahara. Current status of digital content protection and related issues. *Toshiba Review*, 58(6):2–7, June 2003. (in Japanese).

Appendix

A. Example of Camouflaged Program

A simple program containing a conditional branch and the camouflaged program are presented below:

A.1 Original Program (C language)

```
#include <stdio.h>
#define PASSNUM 13

int main() {
    int n;
    scanf("%d", &n);
    if(n!=PASSNUM) {
  printf("INVALID\n");
  return -1;
    }
    printf("OK\n");
    return 0;
}
```

A.2 Original Program (assembly)

```
LC0:
  .ascii "%d\0"
LC1:
  .ascii "INVALID\12\0"
```

```
LC2:
  .ascii "OK\12\0"
  .align 2
.globl _main
_main:
  pushl  %ebp
  movl  %esp, %ebp
  subl  $24, %esp
  andl  $-16, %esp
  movl  $0, %eax
  movl  %eax, -12(%ebp)
  movl  -12(%ebp), %eax
  call  __alloca
  call  ___main
  movl  $LC0, (%esp)
  leal  -4(%ebp), %eax
  movl  %eax, 4(%esp)
  call  _scanf
  cmpl  $13, -4(%ebp)
  je  L10
  movl  $LC1, (%esp)
  call  _printf
  movl  $-1, -8(%ebp)
  jmp  L9
L10:
  movl  $LC2, (%esp)
  call  _printf
  movl  $0, -8(%ebp)
L9:
  movl  -8(%ebp), %eax
  leave
  ret
```

A.3 Camouflaged Program

```
LC0:
  .ascii "%d\0"
LC1:
  .ascii "INVALID\12\0"
LC2:
  .ascii "OK\12\0"
  .align 2
.globl _main
_main:
  movl  $T2 + 0x824, %eax     # RR2
  subl  $0x824, %eax          # RR2
  movb  $0xeb, (%eax)         # RR2
  pushl  %ebp
  subb  $0x3d, T3 + 2         # RR3
  movl  %esp, %ebp
  subl  $24, %esp
  andl  $-16, %esp
  movl  $T1 - 20 + 3, %eax    # RR1
  addl  $20, %eax             # RR1
```

```
T3:
  movb  $0x4a, (%eax)          # RR1 target3
  movl  $0, %eax
  movl  %eax, -12(%ebp)
  movl  -12(%ebp), %eax
  call  __alloca
  call  ___main
  movl  $LC0, (%esp)
  leal  -4(%ebp), %eax
  movl  %eax, 4(%esp)
  movl  $T3 - 0x08 + 2, %eax    # HR3
  addl  $0x08, %eax             # HR3
  movb  $0x4a, (%eax)           # HR3
  call  _scanf
T1:
  cmpl  $7, -4(%ebp)            # target1
  je  L10
  movl  $LC1, (%esp)
  call  _printf
  movl  $-1, -8(%ebp)
T2:
  je  L9                        # target2
L10:
  movl  $LC2, (%esp)
  call  _printf
  movl  $0, -8(%ebp)
  movb  $0x74, T2               # HR2
```

```
L9:
  movl  -8(%ebp), %eax
  movl  $T1 + 0x120 + 3, %eax  # HR1
  subl  $0x120, %eax           # HR1
  movb  $0x07, (%eax)          # HR1
  leave
  ret
```

Index