軟體控制流程之模糊化分析與評估

學生：蔡欣宜　　　　　　　　　　指導教授：黃育綸 博士

國立交通大學 電機學院 電機與控制工程所

摘　　要

模糊軟體程式碼的執行/控制流程有助於阻擋反組譯程式及防止攻擊者惡意竄改程式碼，進而達到保護軟體程式碼完整性與控制存取的保護目的。這些應用於程式碼控制流程的模糊化作業通常可經由一連串的程式碼轉換來達成，在模糊控制邏輯的同時，仍能保有相同的執行結果。然而，目前相關研究僅止於提供可用於軟體程式碼控制流程轉換的模糊化技巧，例如應用內嵌法或模組化、加入等值碼或偽碼等技巧，並未針對模糊化處理後軟體程式碼抵擋反組譯的能力、下降的執行效能與增加的程式碼成本等加以分析。因此，在本篇論文中，我們提出一套剖析原始碼及評估模糊化技巧有效度的方法，藉由搭配軟體程式控制流程的抽象概念，剖析原始碼架構，並透過正規方法來分析模糊化技巧提供的保護能力。在本論文所提出的分析方法中，我們使用數種不同類型的基本轉換元素表示現有的控制流程模糊化作業，用以轉換程式碼的控制流程，模糊化其執行邏輯，進而達到保護的目的。透過模糊轉換的正規化，我們能容易地根據提出的評量準則對模糊化後的程式碼架構進行分析。在我們提出的方法中，不但針對轉換後程式碼的複雜度和抵抗反組譯攻擊的強度加以評估，更討論各種轉換元素所造成的程式碼大小的增加，使程式碼所有者能在複雜度、抗反組譯能力與成本之間，取得最佳平衡點。

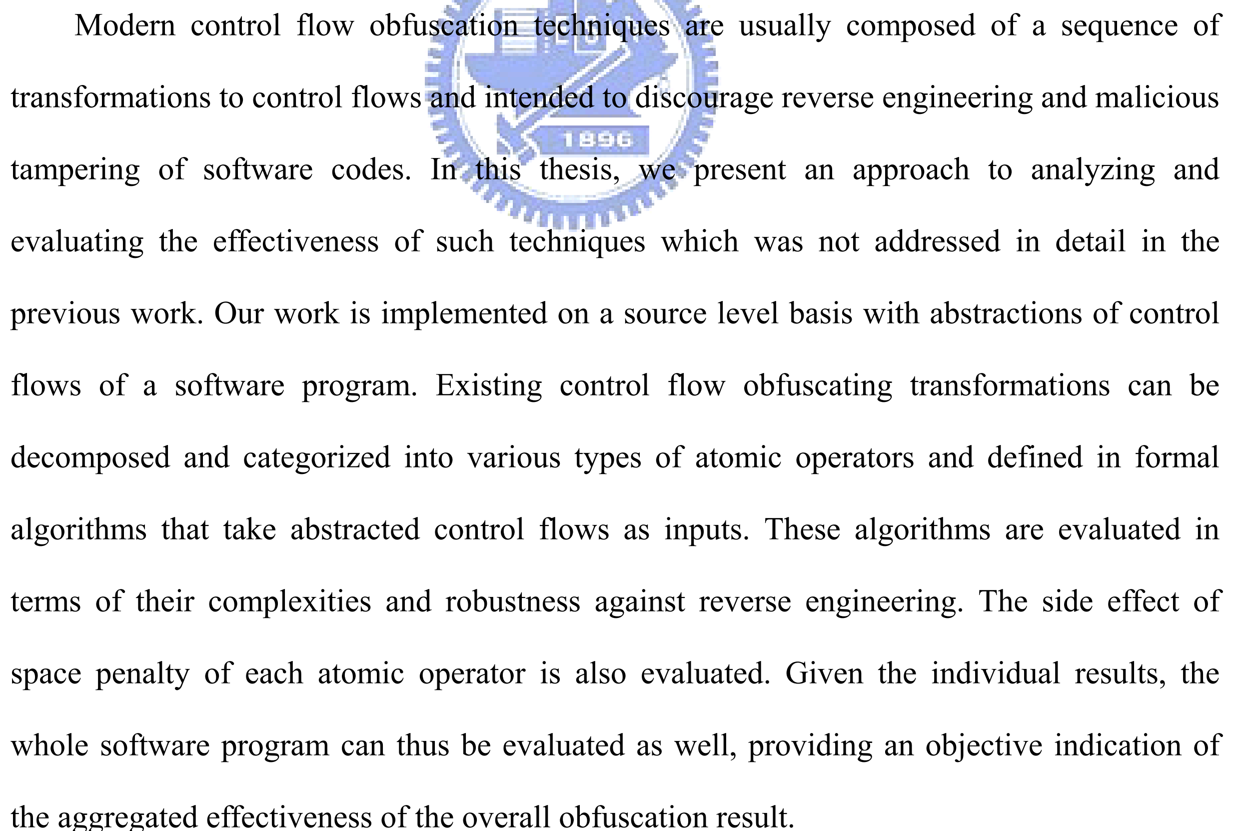
Analysis and Evaluation of Control Flow Obfuscations of Software Programs

Student: Hsin-Yi Tsai　　　　　　　　　　Advisors：Dr. Yu-Lun Huang

Department of Electrical and Control Engineering

National Chiao Tung University

Abstract



Modern control flow obfuscation techniques are usually composed of a sequence of transformations to control flows and intended to discourage reverse engineering and malicious tampering of software codes. In this thesis, we present an approach to analyzing and evaluating the effectiveness of such techniques which was not addressed in detail in the previous work. Our work is implemented on a source level basis with abstractions of control flows of a software program. Existing control flow obfuscating transformations can be decomposed and categorized into various types of atomic operators and defined in formal algorithms that take abstracted control flows as inputs. These algorithms are evaluated in terms of their complexities and robustness against reverse engineering. The side effect of space penalty of each atomic operator is also evaluated. Given the individual results, the whole software program can thus be evaluated as well, providing an objective indication of the aggregated effectiveness of the overall obfuscation result.

誌　　謝

不得不再次承認時間流逝的無情，我還記得當初和同學們一一拜訪實驗室、找指導教授的情景，如今已完成碩士學位，將往下一里程邁進。碩士論文能順利完成，首先要感謝指導教授黃育綸老師的悉心指導並且不厭其煩地為我解惑、與我討論。在 RTES 實驗室的二年中，從老師身上學習到了何謂研究、如何呈現，雖不顯著但我也察覺了自己細微的成長，很榮幸能成為老師的學生，謝謝老師。同時，我也要感謝口試委員們提供諸多寶貴的意見與建議，使我的論文內容能更趨於完善。

我要向我的家人獻上最深的謝意，感謝爸爸、媽媽和弟弟總是這麼相信我並支持我做的任何決定，因為有你們，我才能在疲累時休息再出發。當然，還要感謝我的男友，謝謝你包容我的任性，在自身課業壓力大到自顧不暇之餘還得分心照顧我、聽我抱怨、逗我開心，謝謝你。

RTES 實驗室是間溫馨的實驗室，我們一起下棋、一起討論、一起打棒球、一起關起門來說悄悄話，就是這樣和樂融融的氣氛，讓我覺得能成為這間實驗室的一份子真的是太棒了。另外，也感謝從大學陪伴至研究所的同學們，我想我會很想念與你們一起苦中作樂的日子，那些搞不清楚有幾屆的實驗室太鼓達人大賽、圍在一起溫故知新、交換情報的聚餐。我們各選擇了不同的路，希望你們在軍中時、在熬夜加班時、在洗手作羹湯時、在申報所得稅時還能不忘替我加油，我會很感激很感激的。

最後，我想再說一次，謝謝幫助過我、陪伴我、鼓勵我的人，謝謝你們。

Table of Contents

摘 要 i
Abstract ii
誌 謝 iii
Table of Contents iv
List of Tables vi
List of Figures vii
Chapter 1 Introduction 1
1.1 Background......1
1.2 Contribution......3
1.3 Synopsis......3
Chapter 2 Related Work 5
2.1 Control Flow Obfuscation 5
2.2 Evaluation of Obfuscations 9
2.3 Graph Distance 10
Chapter 3 Program Parser 13
3.1 Definition......13
3.2 Format of Parsed Programs 15
Chapter 4 Atomic Operators......17
4.1 Insert Opaque Predicates......17
4.2 Split Code Elements 22
4.3 Reorder Code Elements......26
4.4 Insert Dummy Codes......27
4.5 Replace with Equivalent Codes......29
Chapter 5 Formalization of Obfuscating Transformations......33
5.1 Computation Transformations......33
5.1.1 Branch Insertion Transformation, $\mathcal{T}^B$ 34
5.1.2 Loop Condition Extension Transformation, $\mathcal{T}^L$ 34
5.1.3 Language-Breaking Transformation, $\mathcal{T}^G$......35
5.1.4 Parallelize Code, $\mathcal{T}^P$ 35
5.1.5 Add Redundant Operands, $\mathcal{T}^R$ 36
5.2 Aggregation Transformations, $\mathcal{T}^A$ 36
5.3 Ordering Transformations, $\mathcal{T}^O$ 36
Chapter 6 Evaluation 38
6.1 DP Value 38
6.2 Distance Using Graph Edge 39
6.3 Example of DP Value upon Formalization......40
Chapter 7 Space Penalty 45

Chapter 8 Example: Prime Number List **48**
8.1 Source Program and Parsed Result........ 48
8.2 Obfuscation Formalization 49
8.3 Evaluation and Space Penalty 53
Chapter 9 Conclusion **56**
References **57**

List of Tables

Table 1. Classification of obfuscating transformations 6
Table 2. Space penalty of each atomic operator 46

List of Figures

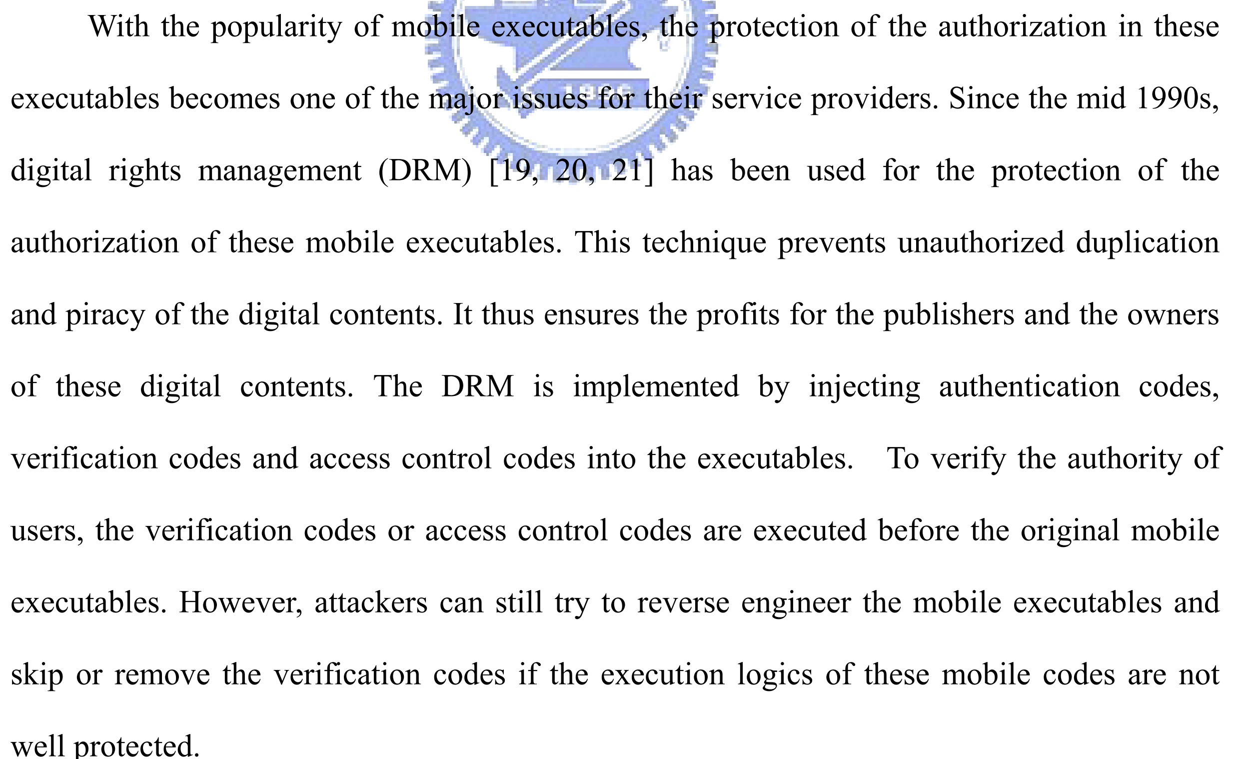
Figure 2.1. Three types of opaque predicates where solid lines indicate paths that may sometimes be taken, and dashed lines represent never-taken paths (C. Collberg [8]) 7
Figure 2.2. Branch insertion transformations (C. Collberg [8]) 7
Figure 2.3. The loop condition insertion transformation (D. Low [7]) 8
Figure 2.4. Example of graph union method 11
Figure 3.1. Example of the formal representation of a parsed program 15
Figure 4.1. Example of applying the operator, $O^f_{Op}(\psi, C_t)$ 19
Figure 4.2. Example of applying the operator, $O^q_{Op}(\psi, C_t)$ 21
Figure 4.3. Split simple blocks 23
Figure 4.4. The fragmented pieces are demarcated by Op_i. To ensure the priorities of any conditions, C_t should be segmented in reverse order. 25
Figure 4.5. Example of inserting dummy simple blocks 28
Figure 4.6. Example of inserting dummy loops 29
Figure 4.7. Example of replacing the target element with its equivalent codes 31
Figure 6.1. Subgraphs in two circles are the common subgraphs of G1 and G2. 40
Figure 6.2. Example of obfuscation formalization 42
Figure 8.1. The parsed representation of Program I 49
Figure 8.2. Program II: the obfuscated version of Program I after applying the specified basic block fission obfuscation 51
Figure 8.3. Program III: the obfuscated version of Program I after applying the specified branch insertion transformation 53

Chapter 1

Introduction

Recently, programs incur more and more risks of being maliciously tampered due to the popularity of mobile executables. To protect the executables from being sinfully pirated, professionals and researchers have proposed several techniques. However, the protections provided by the techniques usually accompany side effects. Hence, it is important to estimate the side effects in advance to get an optimal solution between the effectiveness and overheads of the techniques.

1.1 Background



With the popularity of mobile executables, the protection of the authorization in these executables becomes one of the major issues for their service providers. Since the mid 1990s, digital rights management (DRM) [19, 20, 21] has been used for the protection of the authorization of these mobile executables. This technique prevents unauthorized duplication and piracy of the digital contents. It thus ensures the profits for the publishers and the owners of these digital contents. The DRM is implemented by injecting authentication codes, verification codes and access control codes into the executables. To verify the authority of users, the verification codes or access control codes are executed before the original mobile executables. However, attackers can still try to reverse engineer the mobile executables and skip or remove the verification codes if the execution logics of these mobile codes are not well protected.

It has been reported that Windows Media DRM10 was stripped in 2006 [17, 18]. Marius Oiaga, a technology news editor, reported that "*the application FairUse4WM is designed to*

complement the DRM removal program "drmdbg" that manages files containing digital rights copyright protection code." It was also mentioned that "*FairUse4WM actually permits the stripping of DRM from subscription copyright protected content allowing for unlimited usage of the files independent of the subscription renewal process* [17]." It is clear that Windows Media DRM10 protects the authorization of the digital contents, but fails to protect the authorization codes used to verify the access rights of the contents. It implies that the verification codes can be easily skipped or removed, if the critical sections are compromised.

In recent years, advanced techniques, such as trusted computing platforms (TCP) [24, 25, 26], software encryption [22, 23, 27], software obfuscation [1, 2, 7, 8, 9], have been proposed to protect the execution logics of the authorization codes or access control codes in the mobile executables. When applying the trusted computing techniques, tamper-resistant hardware devices [14] are needed to protect the sensitive credentials. However, software execution performance, deployment flexibility and total cost can be sacrificed if the tamper-resistant devices are used.

In addition to the TCP techniques, many crypto methods are presented to prevent malicious tampering and discourage reverse engineering based on cryptography theories, such as software encryption [13, 22, 23]. In the methods, the encrypted software codes are decrypted right upon execution. However, the time consumed in the cryptography operations is proportional to the code sizes of the software modules, and this kind of methods is impractical for some real-time applications. Upon applying the software control flow obfuscation methods, the execution paths and logics are obfuscated. The authorization codes are hidden in the obfuscated execution paths and thus can be prevented from being skipped or removed. Code obfuscation techniques [1, 3, 7, 8] require no extra hardware and are platform-independence, and thus provides higher flexibility in deploying the mobile executables.

The basic idea of code obfuscation is to transform an application such that the

transformed outcome is functionally identical to the original but is much more difficult to be reverse engineered. Therefore, applications can run on an untrusted platform without the risk of reverse engineering, tampering or intellectual property thefts. Control flow obfuscation aims at disguising the real control flow in a program. Collberg *et al*. [6, 7, 8] proposed a classification of obfuscating transformations and several obfuscation methods. However, they did not clearly explain the improvement of the robustness against reverse engineering after obfuscation. The evaluation of robustness of a program before and after obfuscation is important in getting an optimal solution between security and obfuscation overheads.

1.2 Contribution

In this thesis, we present an approach to analyzing and evaluating the effectiveness of control flow obfuscating transformations which was not addressed in detail in the previous work. Analyzing existing control flow obfuscating transformations, the transformations can be performed with composition of various types of atomic operators which are defined in formal algorithms that take abstracted control flows as inputs. The transformations are evaluated in terms of their robustness against reverse engineering based on the difference between an original and its transformed programs. The side effect of space penalty of each atomic operator is also estimated that given the individual results, the whole overhead on code size can be evaluated as well.

1.3 Synopsis

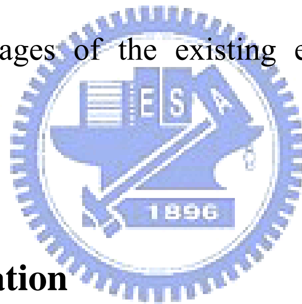
The remainder of this thesis is organized as follows. In the next chapter, we brief the related work. Chapter 3 defines the program parser with the entry and the directed graph of a source program. Chapter 4 depicts the proposed atomic operators which are basic elements used to perform more complicated obfuscating transformations. The formalization of control

flow obfuscating transformations is introduced in Chapter 5. We discuss robustness against reverse engineering after different transformations in Chapter 6, followed by the analysis of overheads as a result of obfuscation in Chapter 7. Chapter 8 presents an example to explain how to apply the approach to a source program. Finally the conclusion is given in the last chapter.

Chapter 2

Related Work

As software security becomes more and more important for their providers, several techniques in control flow obfuscations [2, 7, 8, 9] have been proposed to prevent software programs from being reversely engineered. Besides obfuscating transformations, evaluation measures of obfuscation were defined [6, 7, 8]. In this chapter, we survey some famous obfuscation techniques and evaluation metrics. We point out the weakness or infeasibility of the evaluation measures when they are used to estimate the robustness of control flow obfuscations. After the surveys, we propose an improved approach to analyzing control flow obfuscations upon the advantages of the existing evaluation measures in the following chapters.



2.1 Control Flow Obfuscation

To discourage reverse engineering, several obfuscating transformations [1, 2, 6, 7, 8] are proposed. The transformations can be classified as layout obfuscation, data obfuscation, control flow obfuscation and preventive transformation as Table 1 shows. Layout obfuscations affect the information in a program that is unnecessary to its execution. Examples include scrambling identifier names and removing comments and debugging information. Data obfuscations operate on the data structures used in a program. Extended techniques involve data storage, data encryption, data aggregation and data ordering. Preventive transformations are not intended to obscure a human reader, but intended to stop decompilers and deobfuscators from functioning correctly. Control flow obfuscations target on disturbing the logic of the execution path of the original program to make reverse engineering difficult.

Table 1. Classification of obfuscating transformations

Classification of Obfuscating Transformations	Description
Layout obfuscation	Affect information unnecessary to the execution of a program
Data obfuscation	Change data structures
Preventive transformations	Stop decompilers and deobfuscators from functioning correctly
Control flow obfuscation	Disturb execution flows

Since control flows of a program dominate and reveal the execution logic of the program that implies control flows are the key to understanding programs, we focus on evaluating the effectiveness of control flow obfuscation in this thesis. With the control flow obfuscation methods, the execution logic of the original program can be hidden behind the disturbed flows and thus makes the difficulty of reverse engineering relatively high. Techniques used for control flow obfuscation include branch insertion, ordering transformation, loop condition insertion transformation, etc. Followings depict the brief introduction to several techniques for control flow obfuscation.

Branch insertion transformation [8] is designed by inserting opaque predicates into a program to disturb and conceal the real control flow. An opaque predicate is a Boolean valued expression whose value is known a priori to an obfuscator but difficult for a deobfuscator to deduce. According to the outcome, these opaque predicates can be categorized into three types, as shown in Figure 2.1. For a type I or type II opaque predicate P, it is always evaluates to false or true, which can be denoted by P^F or P^T, respectively. $P^?$ is the representative of a type III opaque predicate P. Its outcome can be sometimes evaluated true and sometimes false.

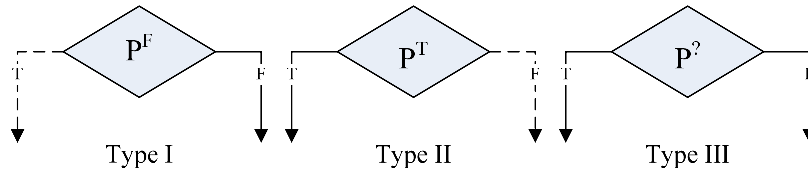


Figure 2.1. Three types of opaque predicates where solid lines indicate paths that may sometimes be taken, and dashed lines represent never-taken paths (C. Collberg [8])

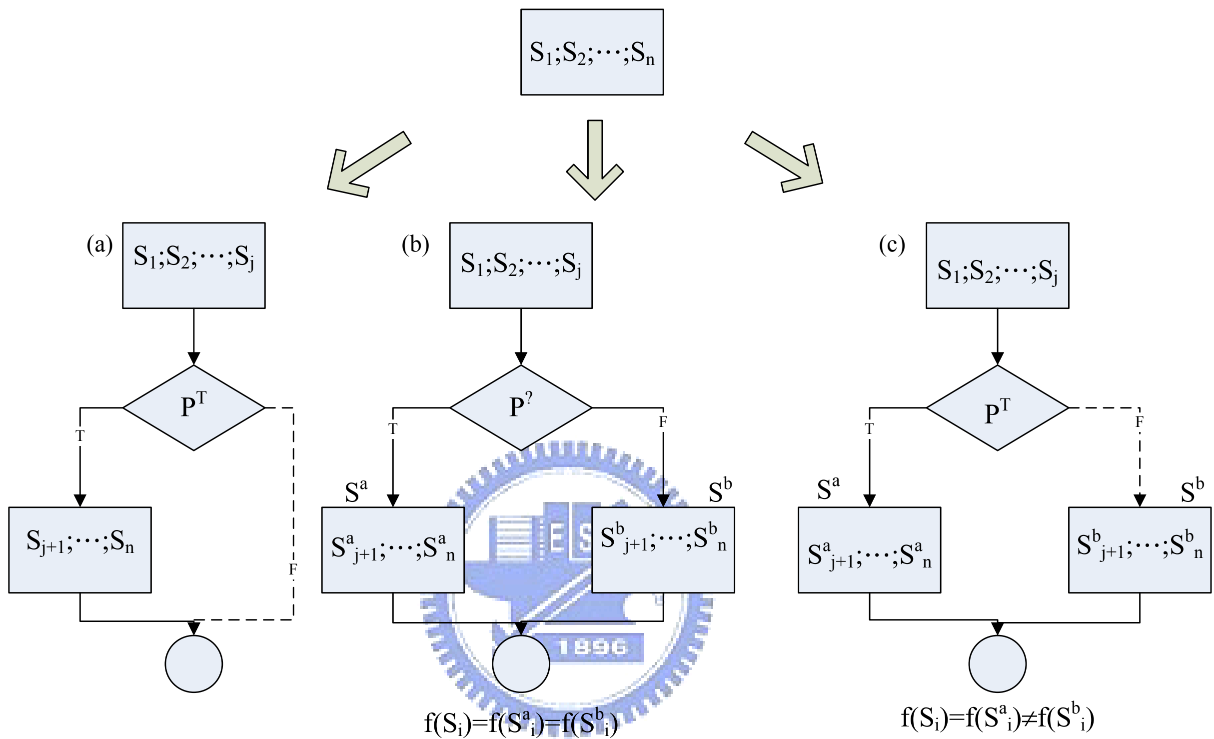


Figure 2.2. Branch insertion transformations (C. Collberg [8])

With a type I or type II predicate, the original codes should be moved to the false or the true target of the predicate to preserve the original functionality. Since type I or type II predicates always achieve the same result, dummy codes can be used for the never-achieved target of the predicates. With a type III predicate, the equivalent codes may be placed on one target while the original codes are placed on the other. Figure 2.2 illustrates the implementation of branch insertion transformations with different types of the opaque predicates.

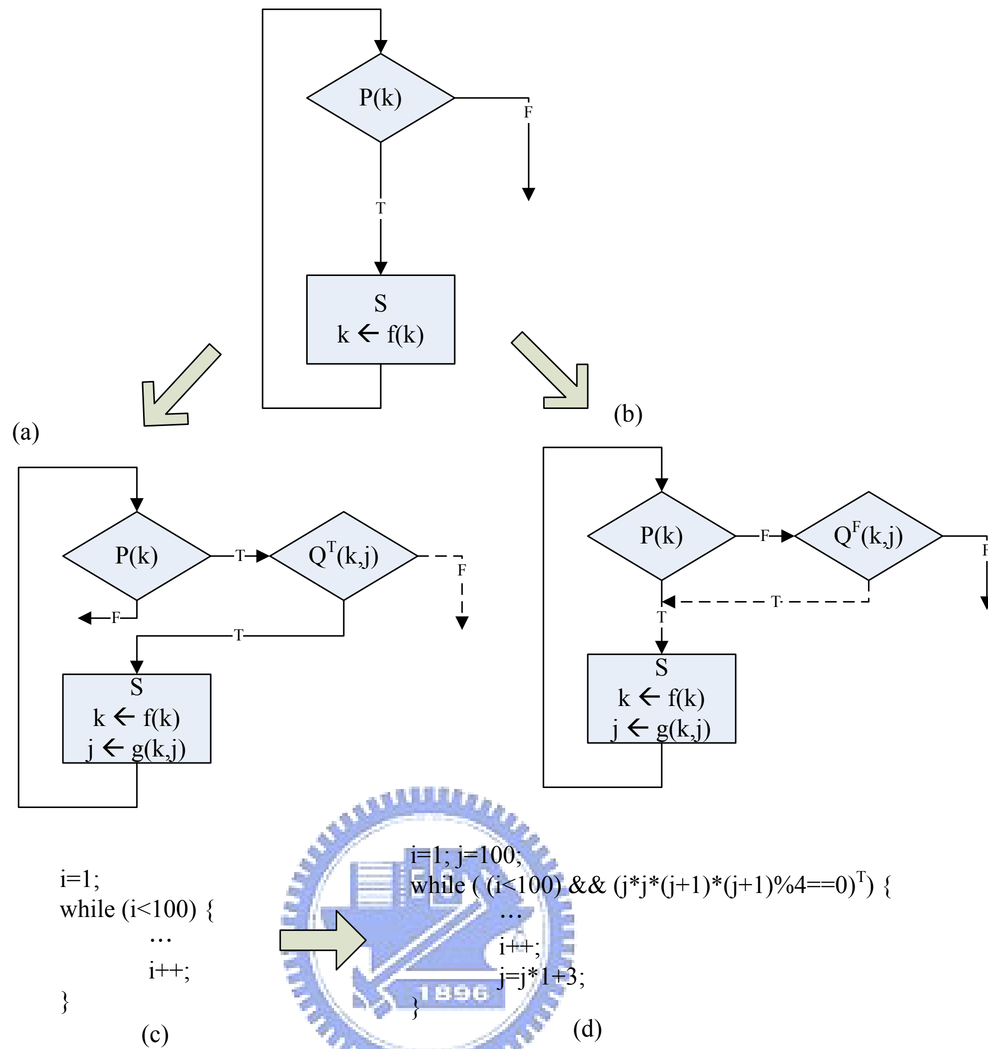


Figure 2.3. The loop condition insertion transformation (D. Low [7])

In addition to branch insertion transformation, the ordering obfuscation [7] randomizes the independent instructions so that the spatial locality of instructions cannot reveal the logical relations among the instructions, nor provide useful clues of the execution logic of the program. On the other hands, ordering obfuscation focus on jumbling the placement of any code section in a source program. Furthermore, to make branch conditions more complex and further increase the difficulty of reverse engineering, type I or II opaque predicates are introduced in loop condition insertion transformations [7]. In Figure 2.3(d), $j^2 * (j + 1)^2$ is used as a type I opaque predicate inserted in the control flow as shown in Figure 2.3(a). The insertion extends the condition of the loop, but the execution result is still preserved.

2.2 Evaluation of Obfuscations

To evaluate the complexity and overhead of obfuscated programs, D. Low *et al*. defined some metrics to evaluate an obfuscating transformation, including resilience, potency and cost. Resilience states how well an obfuscating transformation holds up under attack from an automatic deobfuscator. Cost indicates the additional run-time resources required to execute an obfuscated program. Potency shows the degree to which an obfuscating transformation confuses a human who is trying to understand the obfuscated program.

It is acknowledged that obfuscation can be used to discourage reverse engineering. The time spent on reverse engineering a program normally depends on the ability or the experience of a reverse engineer. Among the three metrics proposed in the previous work [7], only potency implies the difficulty for a reverse engineer to compromise and deduce an obfuscated program. Thus, in this thesis, we evaluate the robustness of an obfuscating transformation upon potency. The following is the definition of potency.

$$pot(P,P') = \frac{comp(P')}{comp(P)} - 1 \qquad \text{Eq (1)}$$

In Eq (1), *comp*(*P*) states the complexity of an original program *P* ,while *comp*(*P'*) refers to that of an obfuscated program *P'*. Despite the fact that the clear definition of potency was given, it was not clearly explained how to determine the explicit values of complexities, *comp*().

To determine the complexities of software programs, many methods are proposed in the past few years, such as Measure Relative Logical Complexity (RLC), Absolute Logical Complexity (ALC) or N-Scope [10]. In these methods, directed graphs are used to represent the software programs, and the complexities of the programs are measured by edges, branches and nodes in the graphs.

Measure RLC uses the ratio of the numbers of branches and nodes to represent the complexity while ALC counts branches only. In N-Scope, the complexity is determined by the

nesting levels of all branches in a program. In control flow obfuscation, the insertion of opaque predicates changes the number and depth of loops in a program, and thus changes its complexity. Nevertheless, according to the rules defined in RLC and ALC, it may result in the same complexities even different control flow obfuscating transformations are applied. Hence, in this thesis, N-Scope is used to evaluate the complexities of the program and then derive the potency for control flow obfuscation.

Potency with N-Scope is sensitive when the numbers or depths of loops are changed. Unfortunately, for some transformations, instructions are reordered without changing the number and depths of loops. In these cases, it results in an unchanged potency value and fails to reflect the complexities introduced by the obfuscating transformations.

2.3 Graph Distance

To address the drawback of potency with N-Scope and further make a precise evaluation, in our work, graph distance, giving a variation of the execution logics between two programs, should be evaluated to indicate how much confusion is to a human reader after obfuscation.

In 1998, Bunke [11] proposed the MCS method with a distance metric based on the maximal common subgraph. The method depicts the distance between two graphs with the number of vertices of their maximal common subgraph and of the larger graph between the two, as shown in Eq (2).

$$d(G_1,G_2)=1-\frac{|mcs(G_1,G_2)|}{max(|G_1|,|G_2|)} \qquad \text{Eq (2)}$$

where |G| means the number of nodes of the graph G, and $mcs(G_1,G_2)$ is the maximal common subgraph of G_1 and G_2.

Nonetheless, if the size of the maximal common subgraph is unchanged, the distance between graphs remains the same even if the smaller graph changes its size. In other words, changes in the smaller graph cannot be reflected when using the MCS method. To improve the

drawback of the MCS method, in 2001, Wallis *et al*. [12] proposed a measure based on graph union, which is referred to below as the graph union method, as shown in Eq (3).

$$d(G_1,G_2)=1-\frac{|mcs(G_1,G_2)|}{|G_1|+|G_2|-|mcs(G_1,G_2|)} \qquad \text{Eq (3)}$$

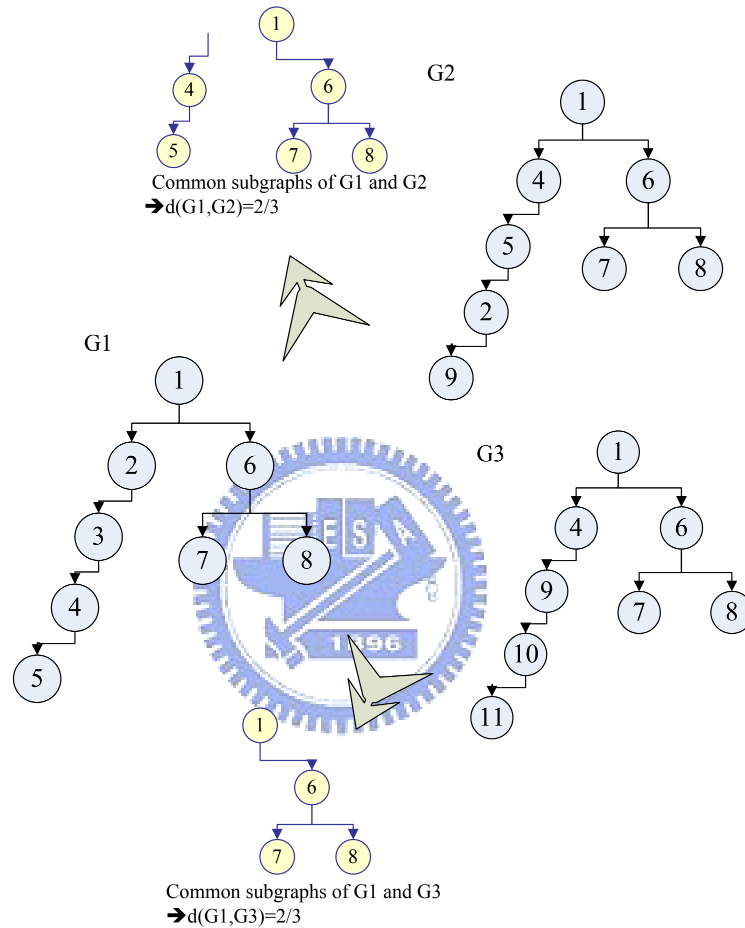


Figure 2.4. Example of graph union method

Using the union rather than the larger of two graphs, changes in the smaller graph can be distinguished. Considering the example in Figure 2.4, *d(G1,G2)* equals to *d(G1,G3)* in terms of the graph union method even though G1 and G2 have more common subgraphs than G1 and G3. The distance changes only when size of the maximal common subgraph is changed. It is not able to reflect the changes in other common subgraphs. To emphasize more on the disparity of the execution logics of two programs, we propose a distance measure in terms of

the number of edges. In the proposed measure, the summation of all edges in all common subgraphs is counted in so that even minor changes can be reflected in the derived distance. The proposed measure is detailed in Chapter 6.

Chapter 3

Program Parser

To formalize the obfuscating transformations, a program must be parsed, fragmented and converted into its graph representation. In this chapter, we describe the definition of basic components of a program graph and the rules for the program parser defined in the thesis.

3.1 Definition

As a high-level abstraction, a software program is composed of a sequence of code blocks. Upon decomposing the software program, it can be converted into a directed graph, whose vertices are the code blocks of the program and edges are the execution orders for these code blocks. In this section, we define the code blocks and edges used to explicitly represent software programs based on directed graphs.

According to the diversion of execution, the code blocks in a program can be classified as branches or simple blocks, defined as follows.

- *Branch* (*B*): A branch in this thesis refers to one of the branch statements used in ***for*** loop, ***while*** loop, ***do-while*** loop, ***if-else*** statements, and ***go-to*** unconditional jumps in high-level programming languages.
- *Simple Block* (*S*): A simple block is defined as a set of sequential statements with no branch instructions inside this code block.
- *Code Element* (*C*): A code element refers to a branch or a simple block. In other words, a set of code elements is the union of sets of branches and simple blocks. A code element can be classified according to their specialties. The following gives a notation of the classification of code elements.

- C_0 is the entry point of a source program.
- *D*, the *don't-care* codes, is the element which is never executed.
- *E(C)* is an equivalence of *C*. It preserves the same functionality as *C*.
- An empty statement (φ) represents the termination of an execution path and can also be considered as a code element.

The edges of a directed graph of a software program are the execution sequences of the program. Normally, instructions in a program are either sequentially executed or branched. To specify the different characteristics in execution, two types of edges, *sequential edge* and *branch edge*, are defined respectively to represent the execution sequences of code elements.

- *Sequential edge* (***e***): A *sequential edge*, $\boldsymbol{e} = (C_i, C_j)$, $i \neq j$, is defined for two code elements, C_i and C_j, whose execution order is sequential. It stands that C_j, the immediate successor, is executed right after C_i, the immediate predecessor.
- *Branch edge* (***b***): Since a branch may jump to its true or false target, there are two code elements which are possible to be executed right after the branch, *B*. To state the branching relationship between the branch and its targets, a *branch edge*, $\boldsymbol{b} = B(C^{True}, C^{False})$, is defined. In this representation, *B* stands for the branch, where C^{True} and C^{False} represent the true and false targets of *B*, respectively.

With these notations and definitions, the directed graph, $\mathcal{G}$, can be represented by a combined tuple $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ is the vertex set and $\mathcal{E}$ is the edge set. The vertex set, $\mathcal{V}$, contains all the code elements, including simple blocks and branches, in the parsed program. The edge set, $\mathcal{E}$, is composed of *sequential edges* and *branch edges*. The following formal representation details the relationship of the above notations.

$\mathcal{G} = (\mathcal{V}, \mathcal{E})$.

$\mathcal{V} = \mathcal{C} = \mathcal{S} \cup \mathcal{B}$, where

$\mathcal{S} = \{S_i \mid \forall i, S_i \text{ is a simple block}\}$,

$\mathcal{B} = \{B_i \mid \forall i, B_i \text{ is a branch}\}$,

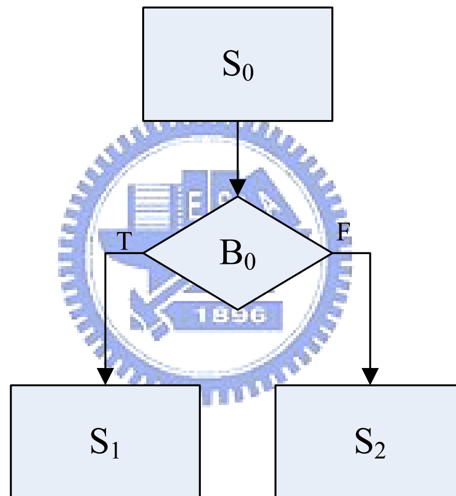
$\mathcal{C} = \{C_i \mid \forall i, C_i$ is a code element, either a simple block or a branch$\}$.

$\mathcal{E} = \{E_i \mid \forall i, E_i$ is either a *sequential edge* or a *branch edge*$\}$.

3.2 Format of Parsed Programs

With the definitions described in Section 3.1, a software program can be decomposed and formatted into a parsed program, Ψ, with its entry point C_0 and a directed graph $\mathcal{G}$, represented in the form of

$$\Psi = (C_0, \mathcal{G}).$$



$\Psi = (S_0, \{S_0, B_0, S_1, S_2\}, \{(S_0, B_0), B_0(S_1,S_2), (S_1,\varphi), (S_2,\varphi)\})$

Figure 3.1. Example of the formal representation of a parsed program

Figure 3.1 shows an example to decompose a program into three simple blocks and one branch, where

$C_0 = S_0$,

$\mathcal{V} = \{S_0, B_0, S_1, S_2\}$,

$\mathcal{E} = \{(S_0, B_0), B_0(S_1, S_2), (S_1, \varphi), (S_2, \varphi)\}$,

$\mathcal{G} = (\mathcal{V}, \mathcal{E})$

$= (\{S_0, B_0, S_1, S_2\}, \{(S_0, B_0), B_0(S_1, S_2), (S_1, \varphi), (S_2, \varphi)\}).$

Thus, the parsed program can be presented as

$\Psi = (C_0, \mathcal{G})$

$= (S_0, \{S_0, B_0, S_1, S_2\}, \{(S_0, B_0), B_0(S_1,S_2), (S_1, \varphi), (S_2, \varphi)\}).$

Chapter 4

Atomic Operators

A program graph is derived by executing the program parser mentioned in the pervious chapter. Therefore, obfuscating control flows of a program can be treated as modifying the program graph into another graph. To maintain the same functionalities of a program, any modification to this can be achieved by changing edges and/or vertices in its program graph. It is possible to split or merge vertices and edges without affecting the functionalities. Yet, it is difficult to remain the same execution results by merely applying deletion to edges or vertices.

The addition without affecting functionalities can be realized by inserting branches, equivalent vertices or dummy vertices. The modification can be realized by splitting, merging vertices or extracting new vertices from existing ones. Since that each vertex in the program graph represents a code element in the corresponding program, from these actions, we conclude five atomic operators as basic building blocks for obfuscating programs. These operators are inserting opaque predicates, splitting code elements, reordering code elements, inserting equivalent codes and dummy codes. In this thesis, the atomic operator, $\boldsymbol{O}$, applying to the target code element C_t in the program Ψ, is defined in the form of

$$\boldsymbol{O}(\Psi, C_t).$$

The following sections describe the proposed atomic operators and their transformation algorithms.

4.1 Insert Opaque Predicates

Inserting opaque predicates hides the real control flow of the source program. Since there are three types of opaque predicates introduced in Chapter 2, the insertions also fall into three

categories, accordingly. O_{op}^{f}, O_{op}^{t} and O_{op}^{q} represent the three types of insertion: type I (false), II (true) and III, respectively. The following sub sections depict the formalization algorithms for the three opaque predicate insertion transformations.

I) Insert Type I Opaque Predicates, $O_{Op}^{f}(\psi, C_t)$

As a type I opaque predicate, P^F, is inserted in front of the target code element C_t, C_t should be moved to the false target of P^F to remain the same functionality. Owing to that the execution result of P^F is always false, any existing element, C_{any}, may be specified as the true target of P^F that C_{any} does not contribute to various results. In case that the code size is not a factor in obfuscating the program, a *don't-care* element can also be placed on the true target by applying another atomic operator, inserting dummy codes. The following is the algorithm of applying $O_{op}^{f}(\psi, C_t)$ to a parsed program, Ψ.

Algorithm 4.1.1 Type I Opaque Predicate Insertion, $O_{Op}^{f}(\psi, C_t)$

Insert P^F into $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} \cup \{P^F\}$;
IF $C_t = C_0$ **THEN**
  Replace the entry point with P^F;
END IF;
FOR $\boldsymbol{e_{ij}}$:= a *sequential edge*(C_{ei}, C_{ej}) **OR** $\boldsymbol{b_{ij}}$:= a *branch edge* $B_{ij}(C_{bi}, C_{bj})$ in $\mathcal{E}$ **DO**
  SWITCH C_t
    CASE C_{ei}:
      $b_{new} \leftarrow P^F(C_{any}, C_t)$;
    CASE C_{ej}:
      Replace (C_{ei}, C_{ej}) with (C_{ei}, P^F);
    CASE B_{ij}:
      $b_{new} \leftarrow P^F(C_{any}, C_t)$;
    CASE C_{bi}:
      Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(P^F, C_{bj})$;
    CASE C_{bj}:
      Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(C_{bi}, P^F)$;
END FOR;

Insert b_{new} to $\mathcal{E}$; ■

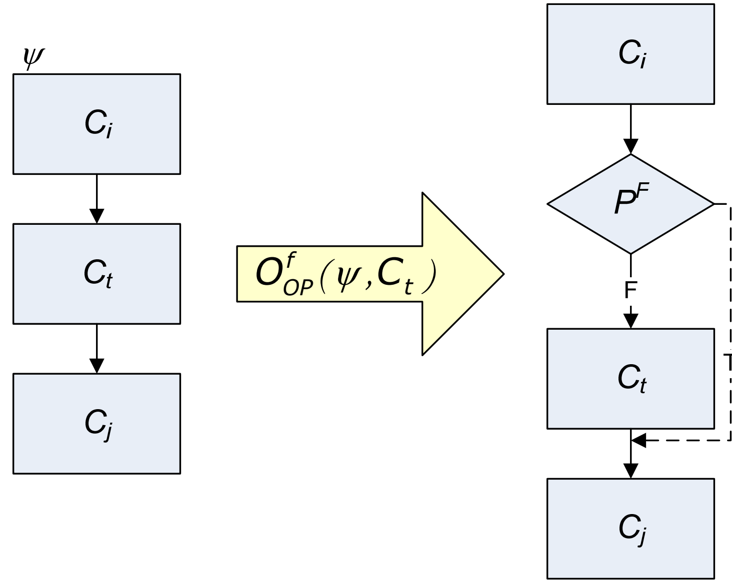


Figure 4.1. Example of applying the operator, $O^f_{Op}(\psi, C_t)$

The insertion varies according to the characteristics of the target code element, C_t. In the case that if C_t is also the entry point C_0, then, in the graph representation of the program, C_0 is replaced by P^F. If C_t is the code element C_{ei} in the *sequential edge* $\boldsymbol{e_{ij}}$ = (C_{ei}, C_{ej}), a *branch edge*, $P^F(C_{any}, C_t)$ is inserted to the program graph $\mathcal{P}$. On the other hand, if C_t is the code element C_{ej} in $\boldsymbol{e_{ij}}$, the existing edge (C_{ei}, C_{ej}) is replaced by (C_{ei}, P^F). In the third case, if C_t is the branch code elements (B_i) in a *branch edge* $\boldsymbol{b_{ij}}$, then a new edge $P^F(C_{any}, C_t)$ should be inserted to the program graph. If C_t is one of the target code elements of the *branch edge* $\boldsymbol{b_{ij}}$, then $B_i(C_{bi}, C_{bj})$ will be replaced by either $B_i(P^F, C_{bj})$ or $B_i(C_{bi}, P^F)$, respectively. In the example as shown in Figure 4.1, C_j is designated as C_{any}. That is, C_j is assigned to the true target of P^F in this example.

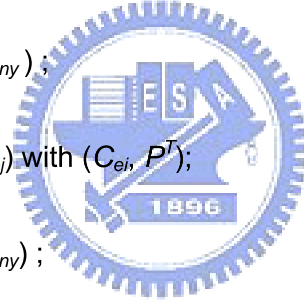
II) Insert Type II Opaque Predicates, $O^t_{Op}(\psi, C_t)$

The procedure for inserting type II opaque predicates is similar to that for type I

predicates. Since a type II opaque predicate (P^T) always evaluates true, the target code element C_t should be moved to the true target of P^T as it is inserted such that the maintenance of the execution result is assured. Similarly, any code element can be its false target that contributes nothing to the execution result. The formalization algorithm for type II opaque predicate insertion, $O^t_{op}(\psi, C_t)$, is described in Algorithm 4.1.2.

Algorithm 4.1.2 Type II Opaque Predicate Insertion, $O^t_{Op}(\psi, C_t)$

Insert P^T into $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} \cup \{P^T\}$;
IF $C_t = C_0$ **THEN**
    Replace the entry point with P^T;
END IF;
FOR $e_{ij} := (C_{ei}, C_{ej})$ **OR** $b_{ij} := B_{ij}(C_{bi}, C_{bj})$ **DO**
    SWITCH C_t
        CASE C_{ei}:
            $b_{new} \leftarrow P^T(C_t, C_{any})$;
        CASE C_{ej}:
            Replace (C_{ei}, C_{ej}) with (C_{ei}, P^T);
        CASE B_{ij}:
            $b_{new} \leftarrow P^T(C_t, C_{any})$;
        CASE C_{bi}:
            Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(P^T, C_{bj})$;
        CASE C_{bj}:
            Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(C_{bi}, P^T)$;
END FOR;
Insert b_{new} to $\mathcal{E}$; ■



III) Insert Type III Opaque Predicates, $O^q_{Op}(\psi, C_t)$

As described in Chapter 2, a type III opaque predicate ($P^?$) may sometimes evaluates true and sometimes false, thus the target code element C_t should be placed on both targets of $P^?$, as shown in Figure 4.2. However, this placement is meaningless in performing obfuscation. It is strongly recommended applying another atomic operator, inserting equivalent codes, after type III opaque predicate insertion. The insertion of equivalent

codes is described in Algorithm 4.4. Algorithm 4.1.3 explains the procedure upon inserting type III opaque predicates for obfuscation.

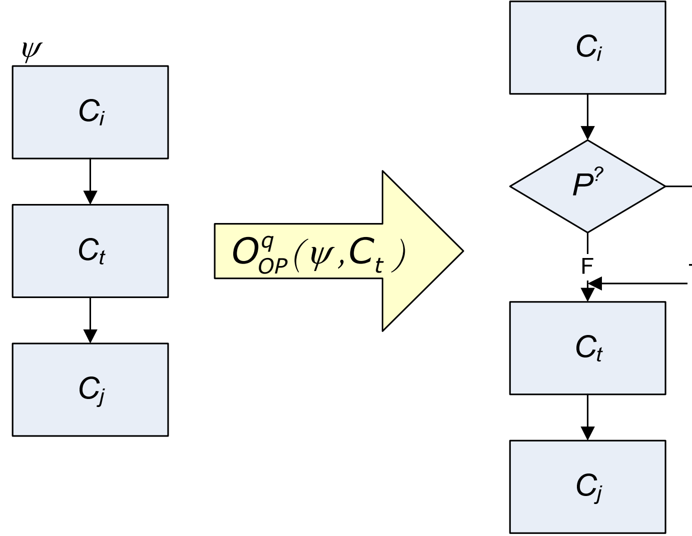


Figure 4.2. Example of applying the operator, $O^q_{Op}(\psi, C_t)$

1896

Algorithm 4.1.3 Type III Opaque Predicate Insertion, $O^q_{Op}(\psi, C_t)$

Insert $P^?$ into $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} \cup \{P^?\}$;

IF $C_t = C_0$ **THEN**

    Replace the entry point with $P^?$;

END IF;

FOR $\boldsymbol{e_{ij}} := (C_{ei}, C_{ej})$ **OR** $\boldsymbol{b_{ij}} := B_{ij}(C_{bi}, C_{bj})$ **DO**

    SWITCH C_t

        CASE C_{ei}:

            $b_{new} \leftarrow P^?(C_t, C_t)$;

        CASE C_{ej}:

            Replace (C_{ei}, C_{ej}) with $(C_{ei}, P^?)$;

        CASE B_{ij}:

            $b_{new} \leftarrow P^?(C_t, C_t)$;

        CASE C_{bi}:

            Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(P^?, C_{bj})$;

        CASE C_{bj}:

            Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(C_{bi}, P^?)$;

END FOR;

Insert b_{new} to $\mathcal{E}$; ■

4.2 Split Code Elements

Splitting a code element into pieces can increase the number of vertices in the program graph and also increase its complexity. Since code elements are distinguished into simple blocks and branches, two types of splitting operators are addressed in this section.

I) Split Simple Blocks, $O_{Ss}^{n}(\psi, C_t)$

Type I splitting operator, denoted by $O_{Ss}^{n}(\psi, C_t)$, tries to obfuscate the simple block, C_t, in further by dividing C_t into n pieces, C_{ti}, $\forall\ 0 \le i < n$. Figure 4.3 shows the operation of the type I splitting operator. Due to the limitation of instruction counts in the source code level, the parameter n is limited to the instruction counts (N) in the target code element C_t. The algorithm for formalizing this operator, $O_{Ss}^{n}(\psi, C_t)$, is described as follows.

Algorithm 4.2.1 Split Simple Blocks, $O_{Ss}^{n}(\psi, C_t)$

IF $C_t \in \mathcal{B}$ **THEN**
    BREAK;
END IF;
IF n > N **OR** n < 2 **THEN**
    BREAK;
END IF;
IF $C_t = C_0$ **THEN**
    Replace the entry point C_0 with C_{t0};
END IF;
Remove C_t from $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} - \{C_t\}$;
Insert new vertices to $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} \cup \{C_{ti} \mid \forall i, 0 \le i < n\}$;
FOR $\boldsymbol{e_{ij}} := (C_{ei}, C_{ej})$ **OR** $\boldsymbol{b_{ij}} := B_{ij}(C_{bi}, C_{bj})$ **DO**
    IF $C_t = C_{ei}$ **THEN**
        Replace (C_{ei}, C_{ej}) with $(C_{t(n-1)}, C_{ej})$;
    END IF;
    ELSE IF $C_t = C_{ej}$ **THEN**

Replace (C_{ei}, C_{ej}) with (C_{ei}, C_{t0});

END IF;

IF $C_t = C_{bi}$ **THEN**

Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(C_{t0}, C_{bj})$;

END IF;

ELSE IF $C_t = C_{bj}$ **THEN**

Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_i(C_{bi}, C_{t0})$;

END IF;

END FOR;

$j \leftarrow 0$;

$m \leftarrow |e|$, where $|e|$ is the number of edges in $\mathcal{E}$;

FOR all j such that $0 < j < n - 1$ **DO**

Insert $\mathbf{e}_{m+j} = (C_{tj}, C_{t(j+1)})$ to $\mathcal{P}$

$j \leftarrow j + 1$;

END FOR; ■

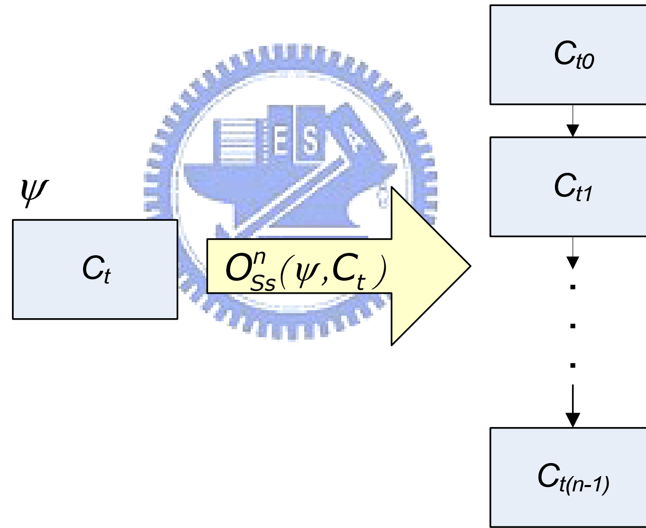


Figure 4.3. Split simple blocks

II) Split Branches, $O^n_{Sb}(\psi, C_t)$

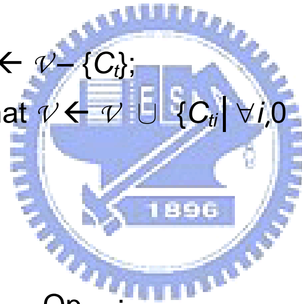
Type II splitting operator, denoted by $O^n_{Sb}(\psi, C_t)$, aims at splitting a target branch, C_t, into n smaller pieces. Before splitting, the expression of the branch should be converted as in postfix. In this case, the expression is reformatted as

$$Cond_0\ Cond_1\ Op_0\ Cond_2\ Op_1 \ldots\ldots\ Cond_{N-2}\ Op_{N-3}\ Cond_{N-1}\ Op_{N-2}$$

where $Cond_i$ is a condition in C_t, and Op_j refers to "AND" or "OR." The format contains N conditions and (N-1) operators. Note that when applying type II splitting operator, the parameter n cannot be larger than N, condition counts of C_t. Algorithm 4.2.2 details the steps of splitting branches.

Algorithm 4.2.2 Split Branches, $O^n_{Sb}(\psi, C_t)$

IF $C_t \in \mathcal{S}$ **THEN**
    BREAK;
END IF;
IF n > N **OR** n < 2 **THEN**
    BREAK;
END IF;
IF $C_t = C_0$ **THEN**
    Replace the entry point C_0 with C_{t0};
END IF;
Remove C_t from $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} - \{C_t\}$;
Insert new vertices to $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} \cup \{C_{ti} \mid \forall i, 0 \leqq i < n\}$;



$C_{t0} \leftarrow C_t$;
$m \leftarrow 2$;
FOR $m \leq n$ **DO**
    $C_{t(m-1)} \leftarrow C_{t(m-2)} - \mathrm{Cond}_{N-(m-1)} - \mathrm{Op}_{N-m}$;
    $C_{t(m-2)} \leftarrow \mathrm{Cond}_{N-(m-1)}$;
    $m \leftarrow m + 1$;
END FOR;
FOR $\boldsymbol{e_{ij}} := (C_{ei}, C_{ej})$ **OR** $\boldsymbol{b_{ij}} := B_i(C_{bi}, C_{bj})$ **DO**
    IF $C_t = C_{ej}$ **THEN**
        Replace (C_{ei}, C_{ej}) with (C_{ei}, C_{t0});
    END IF;
    ELSE IF $C_t = B_i$ **THEN**
        Replace $C_t(C_{bi}, C_{bj})$ with $C_{t(n-1)}(C_{bi}, C_{bj})$;
        FOR $n \geq 2$ **DO**
            SWITCH Op_{N-n}
                CASE "OR"
                    Insert $C_{t(n-2)}(C_{bi}, C_{t(n-1)})$ to $\mathcal{E}$;
                CASE "AND"
                    Insert $C_{t(n-2)}(C_{t(n-1)}, C_{bj})$ to $\mathcal{E}$;

$n \leftarrow n-1$;

END FOR ;

END IF ;

END FOR ; ■

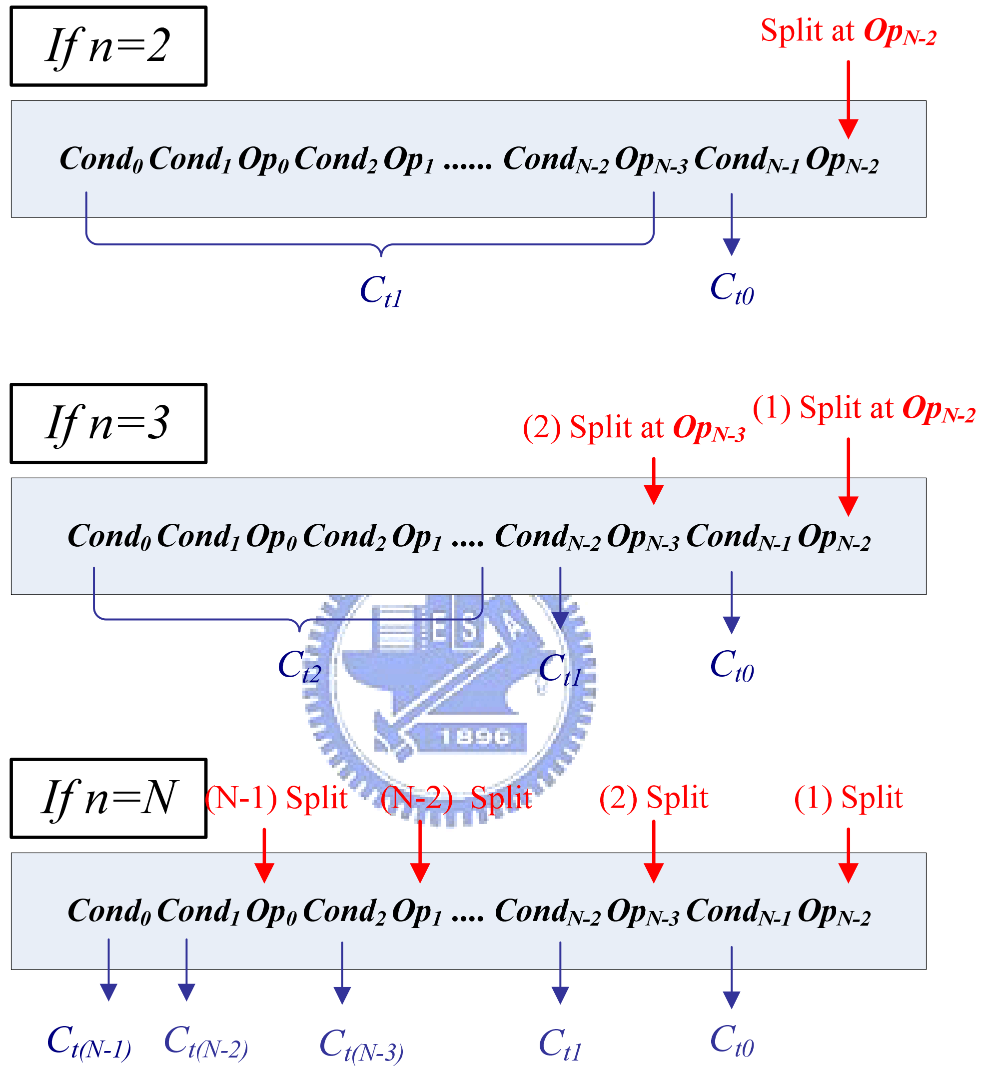


Figure 4.4. The fragmented pieces are demarcated by Op_i. To ensure the priorities of any conditions, C_t should be segmented in reverse order.

The fragmented pieces are demarcated by Op_i, and to ensure the priorities of any conditions, C_t should be segmented in reverse order. That is, the splitting starts from Op_{N-2} sequentially back to Op_0 as Figure 4.4 shows. First, the original expression is segmented into two parts, C_{t0} and C_{t1} where C_{t0} contains only $Cond_{N-1}$, and C_{t1} contains almost the total conditions and operators of the original expression except for Op_{N-2} and $Cond_{N-1}$. After the

first round, C_{t1} is going to be split into a new C_{t1} and C_{t2}. After the second round, C_{t2} is the outcome of removing Op_{N-3} and $Cond_{N-2}$ from C_{t1}, and C_{t1} is reset as $Cond_{N-2}$. The same steps are repeated until n pieces are achieved.

After splitting C_t, new connections between the segmented pieces, $C_{t(n-i)}$ and $C_{t(n-i+1)}$, are developed based on the type of the operator $Op_{N-(n-i+2)}$, i $\in$ [n + 2 – N, n]. In other words, $C_{t(n-i+1)}$ would be the true or false target of $C_{t(n-i+1)}$ due to the type of $Op_{N-(n-i+2)}$. In consideration of the case that the type of Op_{N-2} is "AND," it implies if C_{t0} evaluates false, no matter what the evaluation of C_{t1} is, it fails to reach the original true target of C_t. Under the circumstances, C_{t1} is set as the true target of C_{t0}, and the false target of C_{t0} would be that of the original C_t. For the other case that the type of Op_{N-2} is "OR," although C_{t0} evaluates false, it may reach the original true target of C_t as long as one of the fragments of C_t evaluates true. Thus, C_{t1} should be placed on the false target of C_{t0}. On the basis of the above analysis, if $Op_{N-(n-i+2)}$ is "AND," a new branch edge $C_{t(n-i)}(C_{t(n-i+1)}, C_{bj})$ is inserted where C_{bj} is the false target of C_t. Otherwise, if $Op_{N-(n-i+2)}$ is "OR," a new branch edge $C_{t(n-i)}(C_{bi}, C_{t(n-i+1)})$ is inserted where C_{bi} is the true target of C_t.

4.3 Reorder Code Elements

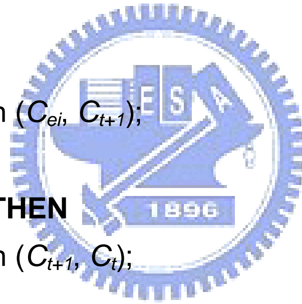
Upon performing reverse engineering, instruction localities usually reveal the execution logics of a program. Randomizing the placement of instructions in a source program helps to hide the execution logics of the source program from being reversely engineered.

The reordering operator, denoted by $O_R(\psi, C_t)$, then becomes one of the atomic operators in obfuscating source programs. However, to remain the execution result unchanged, it is necessary to check the execution dependency for the target code element C_t and its immediate successor C_{t+1}, before applying the reordering operator. If dependency exists, then reordering operator may result in an incorrect execution result. The following is the algorithm

formalizing the reordering operator $O_R(\psi, C_t)$.

Algorithm 4.3 Reorder Code Elements, $O_R(\psi, C_t)$

IF either $C_t \in \mathcal{B}$ **OR** $C_{t+1} \in \mathcal{B}$,
    BREAK;
END IF;
IF dependency exists between C_t and C_{t+1},
    BREAK;
END IF;
IF $C_t = C_0$ **THEN**
    Replace the entry point with C_{t+1}.;
END IF;
FOR $e_{ij} := (C_{ei}, C_{ej})$ **DO**
    IF $C_{t+1} = C_{ei}$ **THEN**
        Replace (C_{ei}, C_{ej}) with (C_t, C_{ej});
    END IF;
    IF $C_t = C_{ej}$ **THEN**
        Replace (C_{ei}, C_{ej}) with (C_{ei}, C_{t+1});
    END IF;
    IF $C_t = C_{ei}$ **AND** $C_{t+1} = C_{ej}$ **THEN**
        Replace (C_{ei}, C_{ej}) with (C_{t+1}, C_t);
    END IF;
END FOR; ■



4.4 Insert Dummy Codes

As insertion of dummy codes changes the original execution logics of a source program, the program becomes more obscure. According to the level of obscurity, the dummy code element which is going to be inserted can be a simple block or a branch. Even a new loop can be created and inserted by combining a branch with a simple block. Insertion of opaque predicates can be regarded as insertion of dummy branches. Hence, in this sub section, we only introduce the algorithms of inserting a dummy loop and simple block. Algorithm 4.4.1 describes the formalization algorithm of the operator on inserting dummy simple blocks D_s,

denoted by $O_D^s(\psi, C_t)$. Algorithm 4.4.2 represents how to insert a dummy loop, which is composed of a dummy branch D_b and simple block D_s, in front of the target C_t.

Algorithm 4.4.1 Insert Dummy Simple Blocks, $O_D^s(\psi, C_t)$

Insert D_s into $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} \cup \{ D_s \}$;
IF $C_t = C_0$ **THEN**
    Replace the entry point with D;
END IF;
FOR $\boldsymbol{e_{ij}} := (C_{ei}, C_{ej})$ **OR** $\boldsymbol{b_{ij}} := B_{ij}(C_{bi}, C_{bj})$ **DO**
    SWITCH C_t
        CASE C_{ej}:
            Replace (C_{ei}, C_{ej}) with (C_{ei}, D_s);
        CASE C_{bi}:
            Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(D_s, C_{bj})$;
        CASE C_{bj}:
            Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(C_{bi}, D_s)$;
END FOR;
Insert a new *sequential edge* (D_s, C_t) to $\mathcal{E}$; ■

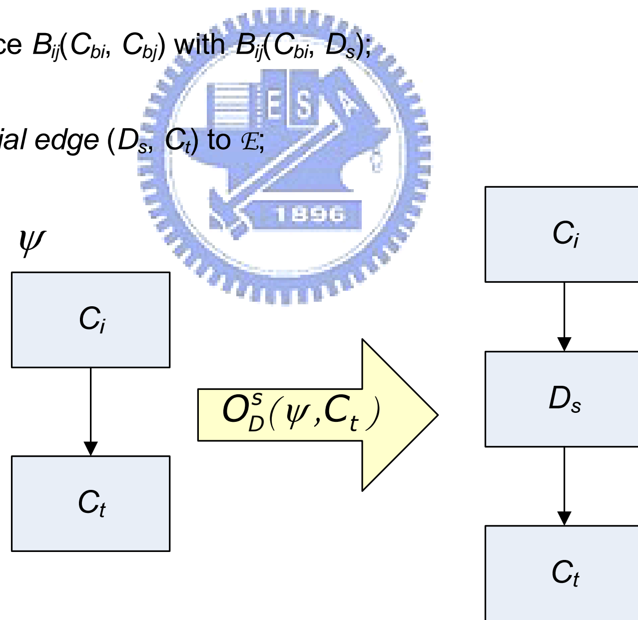


Figure 4.5. Example of inserting dummy simple blocks

Algorithm 4.4.2 Insert Dummy Loops, $O_D^l(\psi, C_t)$

Insert D_s and D_b into $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} \cup \{ D_s, D_b \}$;
IF $C_t = C_0$ **THEN**
    Replace the entry point with D_b;
END IF;
FOR $\boldsymbol{e_{ij}} := (C_{ei}, C_{ej})$ **OR** $\boldsymbol{b_{ij}} := B_{ij}(C_{bi}, C_{bj})$ **DO**
    SWITCH C_t

CASE C_{ej}:

Replace (C_{ei}, C_{ej}) with (C_{ei}, D_b);

CASE C_{bi}:

Replace B_{ij}(C_{bi}, C_{bj}) with B_{ij}(D_b, C_{bj});

CASE C_{bj}:

Replace B_{ij}(C_{bi}, C_{bj}) with B_{ij}(C_{bi} ,D_b);

END FOR;

Insert a new *sequential edge* (D_s, D_b) to $\mathcal{E}$;

Insert a new *branch edge* D_b(D_s, C_t) to $\mathcal{E}$; ■

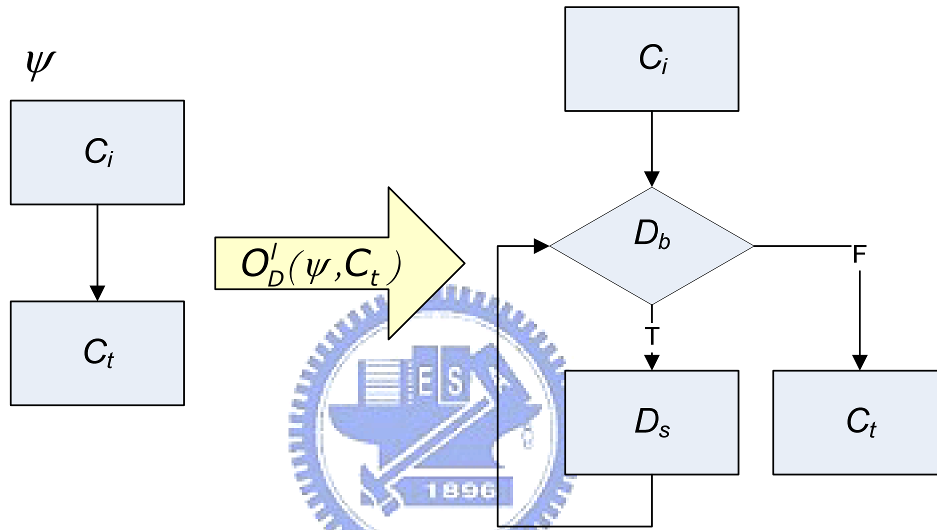


Figure 4.6. Example of inserting dummy loops

4.5 Replace with Equivalent Codes

Equivalent codes are the codes with the same execution result as the origin while implementations of the equivalent codes and the origin are different. In other words, the equivalent codes can confuse reverse engineers by providing the codes with different execution logics while preserving the same functionality.

This operator, denoted by $O_E(\psi, C_t)$, replaces the target code element C_t with its equivalence, $E(C_t)$. For a branch, if both of its true and false targets are C_t, only the false target is replaced with $E(C_t)$, Figure 4.7 illustrating this case. The formalization algorithm of $O_E(\psi, C_t)$ is described as follows.

Algorithm 4.5 Replace with Equivalent Codes, $O_E(\psi, C_t)$

Insert $E(C_t)$ into $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} \cup \{E(C_t)\}$;
Remove C_t from $\mathcal{V}$ such that $\mathcal{V} \leftarrow \mathcal{V} - \{C_t\}$;
IF $C_t = C_0$ **THEN**
    Replace the entry point with $E(C_t)$;
END IF;
FOR $\boldsymbol{e_{ij}} := (C_{ei}, C_{ej})$ **OR** $\boldsymbol{b_{ij}} := B_{ij}(C_{bi}, C_{bj})$ **DO**
    IF $C_t = C_{bi}$ **AND** $C_t = C_{bj}$ **THEN**
        Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(C_{bi}, E(C_t))$;
        Insert C_t into $\mathcal{V}$ such that $\mathcal{V} = \mathcal{V} \cup \{C_t\}$;
    ELSE
        SWITCH C_t
            CASE C_{ei}:
                Replace (C_{ei}, C_{ej}) with $(E(C_t), C_{ej})$;
            CASE C_{ej}:
                Replace (C_{ei}, C_{ej}) with $(C_{ei}, E(C_t))$;
            CASE B_{ij}:
                Replace $B_{ij}(C_{bi}, C_{bj})$ with $E(C_t)(C_{bi}, C_{bj})$;
            CASE C_{bi}:
                Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(E(C_t), C_{bj})$;
            CASE C_{bj}:
                Replace $B_{ij}(C_{bi}, C_{bj})$ with $B_{ij}(C_{bi}, E(C_t))$;
    END IF;
END FOR;
IF $C_t \in \mathcal{V}$ **THEN**
    Find $(E(C_t), y)$;
    Insert (C_t, y);
END IF; ■

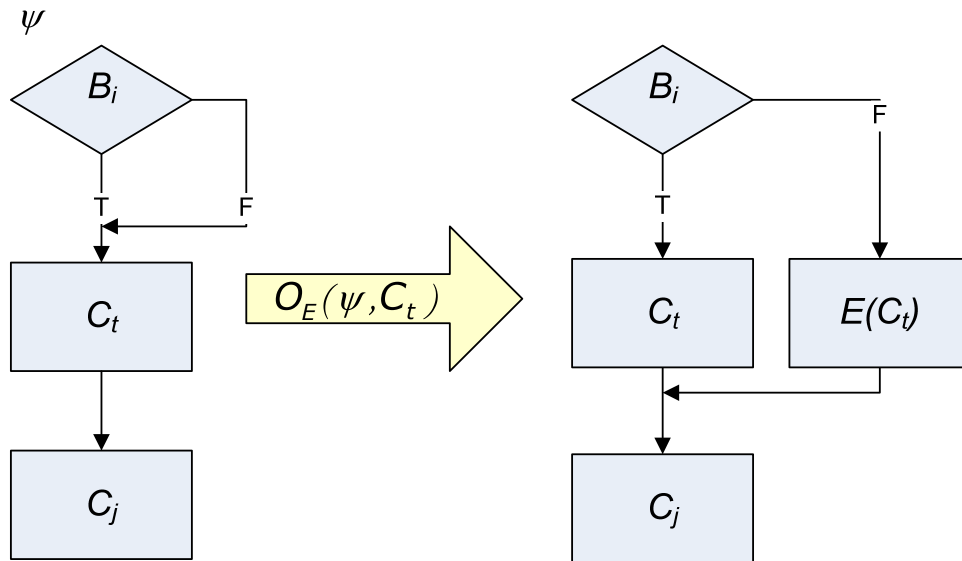


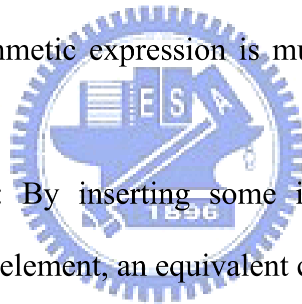
Figure 4.7. Example of replacing the target element with its equivalent codes

Algorithm 4.5 shows the algorithm of inserting equivalent codes based on directed graphs. Techniques which can be used for creating equivalent codes are briefly introduced as follows.

- Inline method: Inline is an important technique in compiler optimization. It is also useful in creating equivalent codes. In inline methods, a procedure call is replaced with the body of called procedure, and the procedure itself is removed.
- Outline method: With outline method, a sequence of instructions is turned into a subroutine and the instructions are replaced with a new procedure call. Outlining is a useful companion transformation to inline methods. They can be combined to create more obscure equivalent codes.
- Interleave method: The idea on interleaving two methods declared in the same class is to merge the bodies and parameter lists of the methods. To distinguish between calls to the two methods, an extra parameter is added to judge which instructions should be executed for the same execution result.
- Clone method: In clone methods, several different versions of a method are created by

applying different sets of techniques in equivalence creation, and a method dispatch is used to select between the different versions at runtime. Clone methods make it appear that different routines are being called, but in fact, the routines derive the same execution.

- Parallelize method: For a target code element, C_t, if the contained instructions can be split into two independent pieces, some technique, like using threads, can be used to execute the pieces in parallel. Otherwise, if no independently smaller pieces are contained in C_t, C_t and a *don't-care* code element, D, are executed in parallel after the creation of D. Note that since D will be executed, it cannot be equivalent codes of C_t or other code elements that will change the execution result.
- Add redundant operands: Once opaque variables have been constructed, we can use algebraic laws to add redundant operands to arithmetic expressions. For example, an original variable of an arithmetic expression is multiplied by an opaque variable whose real value is one.
- Insert dummy instructions: By inserting some instructions that will not change the execution result of the code element, an equivalent code element can be achieved.



Chapter 5

Formalization of Obfuscating Transformations

Since any transformation, $\mathcal{T}$, can be decomposed into a series of atomic operators described in the previous chapters. In this thesis, the obfuscating transformation, $\mathcal{T}$, applying to the program Ψ, is represented in the form of

$$\mathcal{T}(\Psi).$$

Different transformations are derived while atomic operators are applied to different target elements. Therefore, $\mathcal{T}$ can be represented as a subset of the set of the atomic operators defined in this thesis. That is,

$$\mathcal{T}(\Psi) \subseteq \{O^{f}_{Op}(\Psi, C_{t_a}),\ O^{t}_{Op}(\Psi, C_{t_b}),\ O^{q}_{Op}(\Psi, C_{t_c}),\ O^{n}_{Sb}(\Psi, C_{t_d}),\ O^{n}_{Ss}(\Psi, C_{t_e}),\ O_{R}(\Psi, C_{t_f}),\ O_{E}(\Psi, C_{t_g}),\ O^{s}_{D}(\Psi, C_{t_h}),\ O^{l}_{D}(\Psi, C_{t_i})\}^{+},$$

where C_{t_x} is a code element of Ψ, $\forall$ x $\in$ {a, b, c, d, e, f, g, h, i}. In this chapter, we detail a formal representation of these obfuscating transformations with a combination of the above atomic operators.

In 1998, Collberg *et al.* proposed several control transformations of obfuscating the control flow of source programs attempting to further discourage the reverse engineering attacks. These transformations are classified as affecting the computations, aggregation or ordering of the control flow, as described in the following sections.

5.1 Computation Transformations

Computation transformations are targeted to make algorithmic changes by inserting

opaque predicates together with redundant or dummy codes to source programs. Branch insertion, loop condition extension, irrelevant code insertion and non-reducible flow conversion are falling into this category.

5.1.1 Branch Insertion Transformation, $\mathcal{T}^B$

Branch insertion transformations, denoted by $\mathcal{T}^B$, are designed based on one of the three opaque predicate insertion operators, O_{Op}^{f}, O_{Op}^{t} and O_{Op}^{q}. The formalization of the branch insertion transformation can be defined by an ordered four-operator tuple, shown as follows

$$\mathcal{T}^B = (O_{Ss}^{2}, O_{Op}, [O_E], [O_D]).$$

In this transformation, $\mathcal{T}^B$, the target code element is first split into two pieces, which is indicated by O_{Ss}^{2}. Then, the second step is to apply one of the three opaque predicates to the split pieces. The second operator is represented by O_{Op}, where $O_{Op} \in \{ O_{Op}^{f}, O_{Op}^{t}, O_{Op}^{q} \}$. Finally, the insertion of equivalent codes and dummy codes, O_E and O_D, are optional in this transformation.

5.1.2 Loop Condition Extension Transformation, $\mathcal{T}^L$

A loop can be obfuscated by making the loop condition more complex. The idea is to extend the loop condition with a type I (P^T) or type II (P^F) opaque predicate that will not affect the times the loop will execute. For this purpose, the target code element can first be split into two pieces and then the opaque predicates, P^T or P^F, can be inserted into the program. A dummy code can also be used to replace the never reached target of the predicate, optionally. The formal representation of this transformation can be defined in the form of:

$$\mathcal{T}^L = (O_{Ss}^{2}, O_{Op}, [O_D]),$$

where O_{Ss}^{2} splits a simple block into two halves and O_D is an optional to $\mathcal{T}^L$. Note that, to remain the loop execution times unchanged, only type I or II opaque predicates can be inserted into the split code elements. Therefore, O_{Op} is one of O_{Op}^{f} and O_{Op}^{t}, and thus can be represented as

$$O_{Op} \in \{ O_{Op}^{f},\ O_{Op}^{t} \}.$$

5.1.3 Language-Breaking Transformation, $\mathcal{T}^G$

A language-breaking transformation, denoted by $\mathcal{T}^G$, introduces instruction sequences which have no direct correspondence with any source language construct. After the transformation, when faced with such instruction sequences, a deobfuscator will either have to try to synthesize an equivalent but convoluted source language program or give up altogether.

The language-breaking transformation converts a reducible flow graph to a non-reducible one by turning a structured loop into a loop with multiple headers. The formalization of $\mathcal{T}^G$ is defined with the atomic operators as follows.

$$\mathcal{T}^G = (O_{Ss}^{2}, O_{Op}^{f}, [\, O_D \,]),$$

where O_{Ss}^{2} is the operator on splitting a simple block into two halves, and O_D is an option used in $\mathcal{T}^G$.

5.1.4 Parallelize Code, $\mathcal{T}^P$

A reverse engineer will find a parallel program much more difficult to understand than a sequential one. Thus, parallelization yields high levels of potency. The transformation, $\mathcal{T}^P$, can be formalized with the atomic operator O_E such that the formal representation is shown as follows.

$$\mathcal{T}^P = (O_E),$$

where the technique of creation of equivalent codes is limited to the parallelize method.

5.1.5 Add Redundant Operands, $\mathcal{T}^R$

Algebraic laws can be used to add redundant operands to arithmetic expressions. In this way, the logic of the original expression is modified and the operation becomes more complex. The formalization of $\mathcal{T}^R$ is defined in the form of

$$\mathcal{T}^R = (O_E),$$

where only the method "add redundant codes" can be used as the technique for creating equivalent codes for the atomic operator O_E.

5.2 Aggregation Transformations, $\mathcal{T}^A$

The basic idea of aggregation transformations falls into two categories. The one is to break up codes which programmers aggregated them into a method and scatter the codes over the program. The other is to aggregate the codes which seems not to belong together into one method. The transformation $\mathcal{T}^A$ can be implemented by the operator, O_E, with specific techniques of creation of equivalent codes. The formalization of $\mathcal{T}^A$ can be defined in the form of

$$\mathcal{T}^A \subseteq \{ O_{Ei}, O_{Eo}, O_{Ev}, O_{Ec} \}^+,$$

where O_{Ei} is the operator of inserting equivalent codes which created by inline methods. Similarly, outline methods, interleave methods and clone methods are used for the creation of equivalent codes in the operator O_{Eo}, O_{Ev} and O_{Ec}, respectively.

5.3 Ordering Transformations, $\mathcal{T}^O$

To eliminate useful spatial clues for understanding the execution logics of a program,

ordering obfuscation was proposed to randomize the placement of any code element in the source program. The reordering operator, O_R, is introduced in ordering transformations, $\mathcal{T}^O$. The definition of $\mathcal{T}^O$ is represented in the form of

$$\mathcal{T}^O = (O_R),$$

where O_R exchanges the two target code elements if no dependency exists between them.

Chapter 6

Evaluation

Normally, reverse engineering a software application starts at using disassemblers or decompilers so that executable codes can be decompiled to the corresponding high-level representations. After decompiling or disassembling, reverse-engineers try to gather the desired information by analyzing the control flow and the data structures on the basis of the high-level representations. Hence, the difficulty of reverse engineering an obfuscated program should consist of two categories, decompiling time and the difficulty to reverse-engineers. In this chapter, we try to evaluate the difficulty in reverse engineering after applying different obfuscation methods.

6.1 DP Value

Since decompiling time is normally proportional to code size, the measure referred to in this thesis focuses on how to measure the difficulty for a reverse-engineer in reverse engineering obfuscated programs derived from different obfuscating methods. The difficulty for reverse-engineers depends on their own senses and abilities, i.e. different engineers may spend different time reverse engineering the same program. For systematical and numerical analysis, we propose a measure that tries to eliminate personal factors in reverse engineering. This measure does not express the difficulties of reverse engineering the same program between different reverse-engineers, but this measure distinguishes the difficulties for an identical reverse-engineer while different obfuscating transformations are applied to the same program.

$\mathcal{DP}_{\mathcal{T}}(P_{ori})$ is defined as the measure of the difficulty in reverse engineering an obfuscated

program that is the result after applying a transformation $\mathcal{T}$ to a source program P_{ori}. In other words, $\mathcal{DP}_{\mathcal{T}}(P_{ori})$ indicates the robustness to resist reverse-engineers after an obfuscating transformation $\mathcal{T}$. $\mathcal{DP}_{\mathcal{T}}(P_{ori})$ consists of two metrics, graph distance and potency, and it is defined as follows:

$$\mathcal{DP}_{\mathcal{T}}(P_{ori}) = (dis(P_{ori}, P_{obf}), pot(P_{ori}, P_{obf})). \qquad \text{Eq (4)}$$

In Eq (4), $dis(P_{ori}, P_{obf})$ means the distance between the original program P_{ori} and the obfuscated program P_{obf} that results from applying $\mathcal{T}$ to P_{ori}.

Potency with N-Scope can evaluate the impact made by some control flow obfuscation methods on the difficulty in reverse engineering. However, potency cannot detect the change of the execution paths which provide useful information for reverse-engineers. $dis(P_{ori}, P_{obf})$ indicates how much difference exists between the execution logics of P_{ori} and P_{obf}, and thus makes up for the deficiency of potency. Even if distance measure supplement the drawback of potency, it cannot determine the change resulting from some control flow obfuscation methods, such as branch insertion transformation, as efficacy as potency. Hence, both potency and distance measure are introduced in the proposed method to address evaluation.

An increment of the distance between P_{ori} and P_{obf} means that their correlation is reduced, and thus implies that it is more difficult to understand the logic of P_{ori} by tracing P_{obf}. The more complex the obfuscated program, the more time will be spent on reverse engineering. Therefore, the larger the $\mathcal{DP}$ is, the strong the robustness to resist reverse-engineers after obfuscation.

6.2 Distance Using Graph Edge

To present more explicit disparities between the original and obfuscated programs after different obfuscation methods, all common subgraphs are taken account to measure the distance, not merely the maximal common subgraph. Furthermore, to focus more on

execution logics and execution paths of programs, the number of edges contained in common subgraphs rather than the number of nodes is used in the proposed distance measure.

The proposed distance measure between two graphs G1 and G2 is as follows.

$$dis(G1,G2)=1-\sum_i \frac{2|edge(CS_i(G1,G2))|}{|edge(G1)|+|edge(G2)|} \quad \text{Eq (5)}$$

where CS_i(G1, G2) refers to the ith common subgraph of G1 and G2, *edge*(*G*) means the set of edges within graph G, and |*edge*(*G*)| is the number of edges within G. The minimum value of *dis*(G1, G2) is zero while the two graphs are exactly the same. The maximum value of *dis*(G1, G2) is one while no common subgraph exists between them.

In Figure 6.1, both G1 and G2 are composed of eight nodes and seven edges. Subgraphs in two circles are the common subgraphs of G1 and G2, where two and four nodes are included individually. The smaller common subgraph contains one edge while the larger one has three. According to Eq (5), *dis*(G1, G2) can be derived as

$$dis(G1, G2) = 1-\frac{2\cdot 1+2\cdot 3}{7+7}=\frac{3}{7}.$$

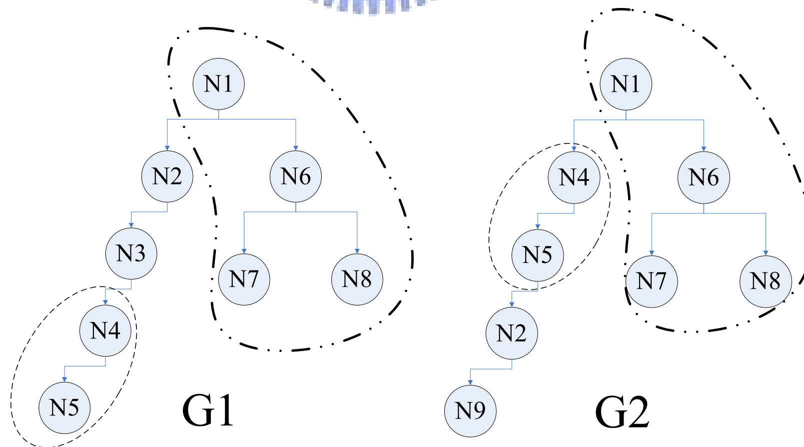


Figure 6.1. Subgraphs in two circles are the common subgraphs of G1 and G2.

6.3 Example of DP Value upon Formalization

To measure distance, we need to compare edge sets of two program graphs, record

conjunct edges and count the number of the conjunct edges. Before measuring potency, complexity of programs should be calculated. After tracing an edge set, we record nodes which are in a loop or on forked paths directed by a branch until the paths encounter. Then we use the number of recorded nodes to derive the potency with N-Scope value. In the following, an example is presented to illustrate how to calculate $\mathcal{DP}$ value.

Taking Figure 6.2 as an example, the original program is parsed to

$$\Psi = (S_0, \{S_0, B_1, S_1, S_2\}, \{(S_0, B_1), B_1(S_1, S_2), (S_1, B_1), (S_2, \varphi)\}),$$

and Ψ becomes

$$\Psi_1 = (S_0, \{S_0, B_1, S_2, S_{10}, S_{11}, S_{12}\}, \{(S_0, B_1), B_1(S_{10}, S_2), (S_{10}, S_{12}), (S_2, \varphi), (S_{12}, S_{11}), (S_{11}, B_1)\}).$$

after applying the transformation $\mathcal{T}_1(\Psi)$ where

$$\mathcal{T}_1(\Psi) = (O_{Ss}^3(\Psi, S_1), O_R(\Psi, S_{11})).$$

Moreover, after applying $\mathcal{T}_2(\Psi_1)$ where

$$\mathcal{T}_2(\Psi_1)$$

$$= \mathcal{T}_3(\Psi)$$

$$= (O_{Ss}^3(\Psi, S_1), O_R(\Psi, S_{11}), O_{Op}^q(\Psi, S_{12}), O_E(\Psi, S_{12})),$$

Ψ_1 is converted to

$$\Psi_2 = (S_0, \{S_0, B_1, S_2, S_{10}, S_{11}, S_{12}, P^?, E(S_{12}), E(S_{11})\}, \{(S_0, B_1), B_1(S_{10}, S_2), (S_{10}, P^?), (S_2, \varphi), (S_{12}, S_{11}), (S_{11}, B_1), P^?(S_{12}, E(S_{12})), (E(S_{12}), S_{11})\}).$$

Comparing edge sets of Ψ and Ψ_1,

$$\sum_i |edge(CS_i(\Psi, \Psi_1))| = 2,$$

$$\forall i, edge(CS_i(\Psi, \Psi_1)) = \{(S_0, B_1), B_1(\times, S_2)\},$$

where $B_1(\times, S_2)$ means that only the relation between B_1 and its false target S_2 is counted in this edge set.

The comparison between Ψ and Ψ_2 is similar to the above.

$$\sum_i |edge(CS_i(\Psi, \Psi_2))| = 2,$$

$$\forall i, edge(CS_i(\Psi, \Psi_2)) = \{(S_0, B_1), B_1(\times, S_2)\}.$$

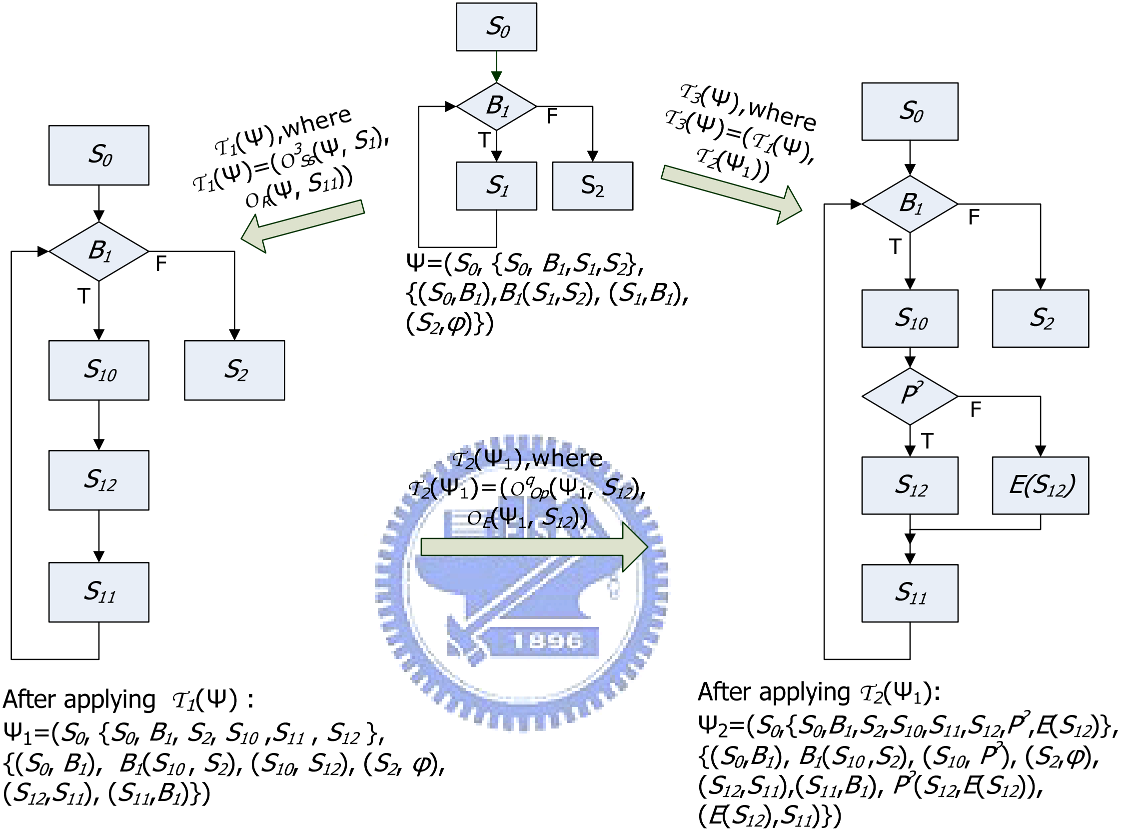


Figure 6.2. Example of obfuscation formalization

According to Eq (5), $dis(\Psi, \Psi_1)$ and $dis(\Psi, \Psi_2)$ can be derived as follows.

$|edge(\Psi)| = 4,$

$|edge(\Psi_1)| = 6,$

$|edge(\Psi_2)| = 9,$

$dis(\Psi,\Psi_1) = 1 - 2 * 2 / (4 + 6) = 3 / 5,$

$dis(\Psi,\Psi_2) = 1 - 2 * 2 / (4 + 9) = 9 / 13.$

Note that (S_2, φ) is not counted in the edge set of the program graphs since φ stands for the

termination of an edge.

The formula of N-Scope is in the form of

$$N - Scope(\Psi) = \frac{\sum_{x_i \in branch\ in\ \Psi} |range(\Psi, B_i)|}{\sum_{x_i \in branch\ in\ \Psi} |range(\Psi, B_i)| + |N_\Psi|} \quad [10], \qquad \text{Eq (6)}$$

where $|N_\Psi|$ is the number of vertices in Ψ, and $|range(\Psi, B_i)|$ means the number of nodes that are contained in the loop leaded by the branch, B_i, or are on the forked paths branching out at B_i until the paths converge.

In Figure 6.2, since Ψ contains a branch, B_1, either a loop or forks exist in Ψ. Observing the edge set of Ψ, according to Eq (6),

$range(\Psi, B_1) = \{B_1, S_1\}$,

$|range(\Psi, B_1)| = 2$.

N-Scope(Ψ) = 1 / 3.

For Ψ_1, nodes contained in the loop leaded by B_1 or lying on any forked paths from B_1 to S_2 are B_1, S_{10}, S_{11} and S_{12}. That is,

$range(\Psi_1, B_1) = \{B_1, S_{10}, S_{11}, S_{12}\}$,

$|range(\Psi_1, B_1)| = 4$,

$N\text{-}Scope(\Psi_1) = 4 / (4 + 6) = 2 / 5$.

The final obfuscated program Ψ_2 has two branches, B_1 and $P^?$. For B_1, its range set is as follows:

$range(\Psi_2, B_1) = \{B_1, S_{10}, S_{11}, S_{12}, P^?, E(S_{12})\}$,

$|range(\Psi_2, B_1)| = 6$.

For the predicate $P^?$, its range set is shown as below.

$range(\Psi_2, P^?) = \{S_{12}, E(S_{12})\}$,

$|range(\Psi_2, P^?)| = 2$.

Based on $|range(\Psi_2, B_1)|$ and $|range(\Psi_2, P^?)|$, $N\text{-}Scope(\Psi_2)$ can be derived as follows.

$N\text{-}Scope(\Psi_2) = (2 + 6) / (2 + 6 + 8) = 1 / 2.$

By the definition of potency, we can derive the following potency values.

$$pot(\Psi, \Psi_1) = \frac{2/5}{1/3} - 1 = \frac{1}{5},$$

$$pot(\Psi, \Psi_2) = \frac{1/2}{1/3} - 1 = \frac{1}{2}.$$

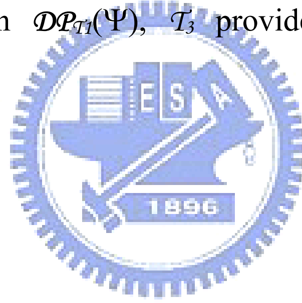
In the above example, the ability to resist reverse engineering by the transformation $\mathcal{T}_1$ with respect to the original program Ψ, is derived as

$$\mathcal{DP}_{T1}(\Psi) = (dis(\Psi, \Psi_1), pot(\Psi, \Psi_1)) = (3 / 5, 1 / 5),$$

where Ψ_1 results from applying $\mathcal{T}_1$ to Ψ. The robustness against reverse engineering after applying a transformation $\mathcal{T}_3$ to the original program Ψ is derived as

$$\mathcal{DP}_{T3}(\Psi) = (dis(\Psi, \Psi_3), pot(\Psi, \Psi_3)) = (9 / 13, 1 / 2).$$

Since $\mathcal{DP}_{T3}(\Psi)$ is larger than $\mathcal{DP}_{T1}(\Psi)$, $\mathcal{T}_3$ provides the stronger robustness to resist reverse-engineers than $\mathcal{T}_1$.



Chapter 7

Space Penalty

Control flow obfuscation uses techniques such as creating buggy loops and inserting dummy codes to disorder the real execution path. After obfuscating transformations, a source program can forbid malicious tampering and reverse engineering. However, it suffers from space penalty. The more transformations applied to the program, the more code size overheads are suffered. Thus, estimation of space penalty is important for assurance whether the increment of code sizes due to the designated transformations is tolerable. Through the proposed formal representation, estimation of space penalty can be efficiently determined in advance such that users can decide whether to apply more transformations or not. In this chapter we analyze overheads on code sizes resulting from each obfuscating transformation.

Assuming that an original parsed program Ψ has n code elements where the size of the ith element is denoted as z_i, $\forall$ i $\in$ [1, n], the total code size of Ψ is $\sum_{i=1}^{n} z_i$. After obfuscating transformations, x simple blocks and y branches are inserted into Ψ where the size of the ith simple block and the jth branch are respectively indicated as s_i and $б_j$, $\forall$i $\in$ [1, x] and $\forall$ j $\in$ [1, y]. Hence, the total code size of the obfuscated program becomes

$$\sum_{i=1}^{n} z_i + \sum_{i=1}^{x} s_i + \sum_{i=1}^{y} б_i ,$$

and the space penalty is

$$\sum_{i=1}^{x} s_i + \sum_{i=1}^{y} б_i .$$

For simplicity of analysis, the summation of the sizes of all inserted elements is replaced with the product of the average size and the number of elements. Since the gap between the

average size of simple blocks and that of braches is too large to be ignored, they should be individually denoted by $\overline{\mathcal{S}}$ and $\overline{\mathcal{B}}$. The mentioned space penalty becomes $x \cdot \overline{\mathcal{S}} + y \cdot \overline{\mathcal{B}}$. In the following, we describe the space penalty with respect to each proposed atomic operator, and Table 1 makes the arrangement.

Table 2. Space penalty of each atomic operator

Atomic Operators	Space Penalty
Insert opaque predicates, $O^f_{Op}(\psi, C_t)$ / $O^t_{Op}(\psi, C_t)$ / $O^q_{Op}(\psi, C_t)$	$\overline{\mathcal{B}}$
Split code elements, $O^n_{Ss}(\psi, C_t)$ / $O^n_{Sb}(\psi, C_t)$	0
Reorder code elements, $O_R(\psi, C_t)$	0
Insert equivalent codes, $O_E(\psi, C_t)$	0 or $\overline{\mathcal{S}}$ or $\overline{\mathcal{B}}$
Insert dummy simple blocks, $O^s_D(\psi, C_t)$	$\overline{\mathcal{S}}$
Insert dummy loops, $O^l_D(\psi, C_t)$	$\overline{\mathcal{S}} + \overline{\mathcal{B}}$

- Insert opaque predicates, $O^f_{Op}(\psi, C_t)$ / $O^t_{Op}(\psi, C_t)$ / $O^q_{Op}(\psi, C_t)$:

  According to the proposed algorithms, any type of opaque predicate insertion introduces a predicate. Thus the space penalty is $\overline{\mathcal{B}}$.

- Split code elements, $O^n_{Ss}(\psi, C_t)$ / $O^n_{Sb}(\psi, C_t)$:

  This operator splits a target code element into smaller pieces. Each smaller piece is a part of the original target element. Hence, the total code size is not raised, i.e. the space penalty is zero, even the number of code elements increases.

- Reorder code elements, $O_R(\psi, C_t)$:

  In this operator, no code element is added. Thus, the space penalty is zero.

- Insert equivalent codes, $O_E(\psi, C_t)$:

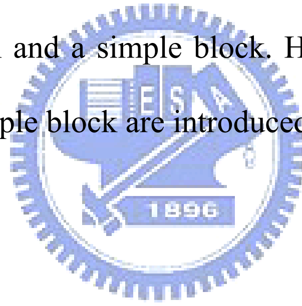
  This operator replaces C_t with its equivalence, E(C_t). If the program Ψ contains a branch edge whose true and false targets are both C_t, then the false target is replaced with E(C_t), and the true target remains the same. In this case, an additional code element is inserted that makes the space penalty become $\overline{\mathcal{S}}$ or $\overline{\mathcal{B}}$ depending on the type of C_t. Otherwise, if no such branch edge as mentioned above exists in Ψ, a new code element is added while an existing element is removed. Hence, the space penalty is zero.

- Insert dummy simple blocks, $O_D^s(\psi, C_t)$:

  An extra simple code element is inserted while applying the operator. Thus the space penalty is $\overline{\mathcal{S}}$.

- Insert dummy loops, $O_D^l(\psi, C_t)$:

  A loop consists of a branch and a simple block. Hence, inserting a dummy loop implies that an extra branch and simple block are introduced. The space penalty is thus $\overline{\mathcal{S}} + \overline{\mathcal{B}}$.



Chapter 8

Example: Prime Number List

In this chapter, we present how the proposed formalization method can be applied to a complete example program, and we will give the evaluation after obfuscation.

8.1 Source Program and Parsed Result

Program Ⅰ, a prime number printer that prints out prime numbers which are not larger than the input *a*, is accounted an example program used to present how the proposed method works.

```
#include <stdio.h>
#include <stdlib.h>
int k(int);
int main()
{   int a, b, sum;
    printf("insert a number:");
    scanf("%d",&a);
     for(sum=0,b=2;b<=a;b++)
     {
       if(k(b)) printf("%3d",b);
       sum+=b;
      }
     return 0;
}

int k(int b)
{   int i;
    for(i=2;i<=b/2;i++)
      if(b%i==0) return 0;
    return 1;
}
```

Program I. Prime number list

We first parse the source program and derive three sets, $\mathcal{S}$, $\mathcal{B}$ and $\mathcal{E}$, as defined in Section 3.1. Figure 8.1 shows the parsed result of Program I, including its control flow, simple blocks, $\forall S_i \in \mathcal{S}$, and branches, $\forall B_j \in \mathcal{B}$. Therefore, we now have

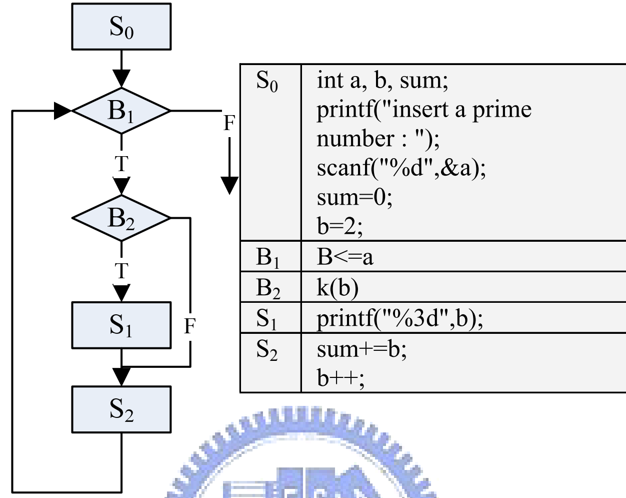
$\mathcal{S} = \{S_0, S_1, S_2\}$,

$\mathcal{B} = \{B_1, B_2\}$ and

$\mathcal{E} = \{(S_0, B_1), B_1(B_2, \varphi), B_2(S_1, S_2), (S_1, S_2), (S_2, B_1)\}$.

The parsed program Ψ is then denoted as

$$\Psi = (S_0, \{S_0, S_1, S_2, B_1, B_2\}, \{(S_0, B_1), B_1(B_2, \varphi), B_2(S_1, S_2), (S_1, S_2), (S_2, B_1)\}).$$



S_0	int a, b, sum;
	printf("insert a prime number : ");
	scanf("%d",&a);
	sum=0;
	b=2;

B_1	B<=a
B_2	k(b)
S_1	printf("%3d",b);
S_2	sum+=b; b++;

Figure 8.1. The parsed representation of Program I

8.2 Obfuscation Formalization

After parsing a source program, we can apply our formalization method to target elements of the program. In this section, we take two control flow obfuscation methods, the basic block fission obfuscation [2] and the branch insertion transformation [8], as the designated transformations applied in the example. The basic block fission obfuscation is to split the chosen basic block into several smaller blocks, where opaque predicates and goto instructions are inserted. With analyzing the basic block fission obfuscation, the action of inserting goto instructions can be corresponding with the atomic operator, O^t_{Op}. Thus, this obfuscation method can be denoted by

$\mathcal{T}_b = \{O^n_{Ss}, O^n_{Sb}, O_{Op}, O_D\}$,

where $O_{Op} \in \{ O^f_{op}, O^t_{op}, O^q_{op} \}$.

Assume that the specific basic block fission obfuscation and branch insertion transformation used in this example are specified respectively as follows.

$$\mathcal{T}_1(\Psi) = (O^2_{Ss}(\Psi, S_0),\ O_D(\Psi, B_1),\ O^t_{Op}(\Psi, B_1),\ O^f_{Op}(\Psi, S_{01})).$$

$$\mathcal{T}_2(\Psi) = (O^2_{Ss}(\Psi, S_0),\ O^f_{Op}(\Psi, S_{01}),\ O_E(\Psi, S_{01}),\ O_D(\Psi, B_1)).$$

In the following, we clarify the procedures of applying the transformations to the program in detail.

I) Apply the specified basic block fission obfuscation, $\mathcal{T}_1(\Psi)$

a) Running $O^2_{Ss}(\Psi, S_0)$:

$\Psi = (S_{00}, \{S_{00}, S_{01}, S_1, S_2, B_1, B_2\}, \{(S_{01}, B_1), B_1(B_2, \varphi), B_2(S_1, S_2), (S_1, S_2), (S_2, B_1), (S_{00}, S_{01})\})$.

In this example, S_{00} is "int a, b, sum; printf("insert a prime number \n");," and S_{01} is "scanf("%d",&a); sum=0; b=2;".

b) Running $O_D(\Psi, B_1)$:

$\Psi = (S_{00}, \{S_{00}, S_{01}, S_1, S_2, B_1, B_2, D_1\}, \{(S_{01}, D_1), B_1(B_2, \varphi), B_2(S_1, S_2), (S_1, S_2), (S_2, B_1), (S_{00}, S_{01}), (D_1, B_1)\})$.

In this case, we choose "$sum = a + b\%4$" as D_1.

c) Running $O^t_{Op}(\Psi, B_1)$:

$\Psi = (S_{00}, \{S_{00}, S_{01}, S_1, S_2, B_1, B_2, D_1, P_1^T\}, \{(S_{01}, D_1), B_1(B_2, \varphi), B_2(S_1, S_2), (S_1, S_2), (S_2, P_1^T), (S_{00}, S_{01}), (D_1, P_1^T), P_1^T(B_1, B_2)\})$.

In this example, "$(a^3 - a)\%3 == 0$" is chosen for the predicate P_1^T.

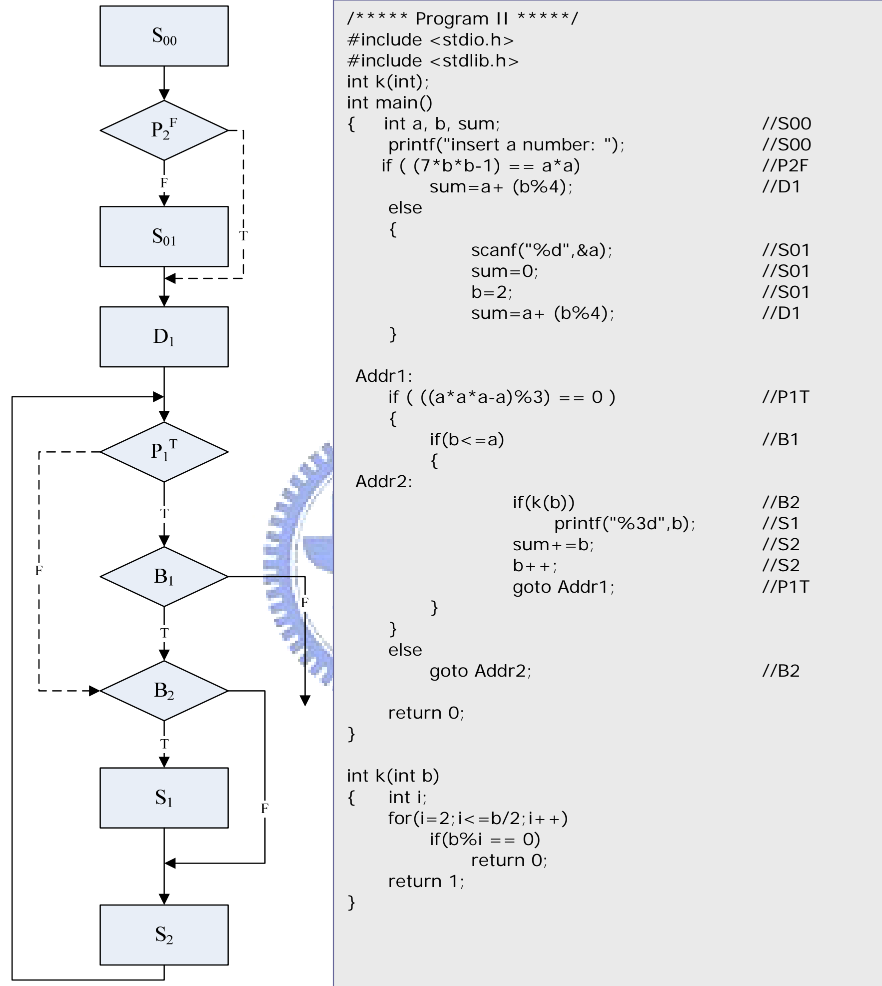
d) Running $O^f_{Op}(\Psi, S_{01})$:

$\Psi_1 = (S_{00}, \{S_{00}, S_{01}, S_1, S_2, B_1, B_2, D_1, P_1^T, P_2^F\}, \{(S_{01}, D_1), B_1(B_2, \varphi), B_2(S_1, S_2), (S_1,$

S_2), (S_2, B_1), (S_{00}, P_2^F), (D_1, P_1^T), P_1^T(B_1, B_2), P_2^F(D_1, S_{01})}).

In this case, we choose "$7b^2 - 1 == a^2$" as P_2^F.

After the basic block fission obfuscation, the obfuscated control flow of Program I is derived, and the obfuscated program, Program II, is then regenerated according to the final result Ψ_1, as shown in Figure 8.2.




```
/***** Program II *****/
#include <stdio.h>
#include <stdlib.h>
int k(int);
int main()
{    int a, b, sum;                                   //S00
     printf("insert a number: ");                     //S00
    if ( (7*b*b-1) == a*a)                            //P2F
          sum=a+ (b%4);                               //D1
     else
     {
               scanf("%d",&a);                        //S01
               sum=0;                                 //S01
               b=2;                                   //S01
               sum=a+ (b%4);                          //D1
     }

 Addr1:
     if ( ((a*a*a-a)%3) == 0 )                        //P1T
     {
          if(b<=a)                                    //B1
          {
 Addr2:
                    if(k(b))                          //B2
                         printf("%3d",b);             //S1
                    sum+=b;                           //S2
                    b++;                              //S2
                    goto Addr1;                       //P1T
          }
     }
     else
          goto Addr2;                                 //B2

     return 0;
}

int k(int b)
{    int i;
     for(i=2;i<=b/2;i++)
          if(b%i == 0)
               return 0;
     return 1;
}
```

Figure 8.2. Program II: the obfuscated version of Program I after applying the specified basic block fission obfuscation

II) Apply the specified branch insertion transformation, $\mathcal{T}_2(\Psi)$

a) Running $O^2_{Ss}(\Psi, S_0)$:

Ψ = (S_{00}, {S_{00}, S_{01}, S_1, S_2, B_1, B_2}, {(S_{01}, B_1), B_1(B_2, φ), B_2(S_1, S_2), (S_1, S_2), (S_2, B_1), (S_{00}, S_{01})}).

In this example, S_{00} is “int a, b, sum; printf("insert a prime number \n");,” and S_{01} is “scanf("%d",&a); sum=0; b=2;”.

b) Running $O_{Op}^{f}(\Psi, S_{01})$:

$\Psi = (S_{00}, \{S_{00}, S_{01}, S_1, S_2, B_1, B_2, P_1^F\}, \{(S_{01}, B_1), B_1(B_2, \varphi), B_2(S_1, S_2), (S_1, S_2), (S_2, B_1), (S_{00}, P_1^F), P_1^F(B_1, S_{01})\})$.

In this case, we choose “$7b^2 - 1 == a^2$” as P_1^F.

c) Running $O_E(\Psi, S_{01})$:

$\Psi = (S_{00}, \{S_{00}, E(S_{01}), S_1, S_2, B_1, B_2, P_1^F\}, \{(E(S_{01}), B_1), B_1(B_2, \varphi), B_2(S_1, S_2), (S_1, S_2), (S_2, B_1), (S_{00}, P_1^F), P_1^F(B_1, E(S_{01}))\})$.

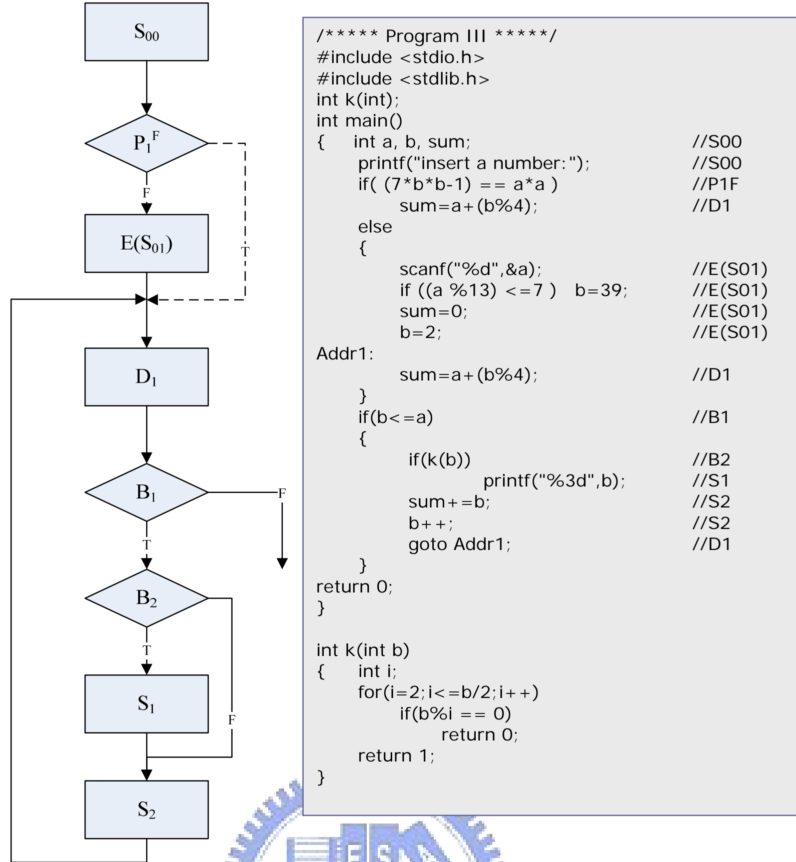
In this example, the equivalent code element $E(S_{01})$ is generated by inserting dummy instructions.

d) Running $O_D(\Psi, B_1)$:

$\Psi_2 = (S_{00}, \{S_{00}, E(S_{01}), S_1, S_2, B_1, B_2, P_1^F, D_1\}, \{(E(S_{01}), D_1), B_1(B_2, \varphi), B_2(S_1, S_2), (S_1, S_2), (S_2, D_1), (S_{00}, P_1^F), P_1^F(D_1, E(S_{01})), (D_1, B_1)\})$.

In this case, we choose “$sum = a + b\%4$” as D_1.

After applying the branch insertion transformation $\mathcal{T}_2$, the obfuscated control flow of Program I is shown in Figure 8.3. With deparsing the obfuscated program Ψ_2, the obfuscated program, Program III, resulting from $\mathcal{T}_2$ can be achieved.




```
/***** Program III *****/
#include <stdio.h>
#include <stdlib.h>
int k(int);
int main()
{    int a, b, sum;                          //S00
     printf("insert a number:");             //S00
     if( (7*b*b-1) == a*a )                  //P1F
          sum=a+(b%4);                       //D1
     else
     {
          scanf("%d",&a);                    //E(S01)
          if ((a %13) <=7 )   b=39;          //E(S01)
          sum=0;                             //E(S01)
          b=2;                               //E(S01)
Addr1:
          sum=a+(b%4);                       //D1
     }
     if(b<=a)                                //B1
     {
           if(k(b))                          //B2
                   printf("%3d",b);          //S1
           sum+=b;                           //S2
           b++;                              //S2
           goto Addr1;                       //D1
     }
return 0;
}

int k(int b)
{    int i;
     for(i=2;i<=b/2;i++)
          if(b%i == 0)
               return 0;
     return 1;
}
```

Figure 8.3. Program III: the obfuscated version of Program I after applying the specified branch insertion transformation

8.3 Evaluation and Space Penalty

According to the formal representations of the original and two obfuscated programs, we obtain the following analysis. Comparing Ψ and Ψ_1,

$$\sum_i |edge(CS_i(\Psi, \Psi_1))| = 4,$$

$$\forall i, edge(CS_i(\Psi, \Psi_1)) = \{B_1(S_2, \varphi), B_2(S_1, S_2), (S_1, S_2)\}.$$

$$\sum_i |edge(CS_i(\Psi, \Psi_2))| = 4,$$

$$\forall i, edge(CS_i(\Psi, \Psi_2)) = \{B_1(S_2, \varphi), B_2(S_1, S_2), (S_1, S_2)\}.$$

The number of edges in Ψ, Ψ_1 and Ψ_2 are 6, 12 and 10, respectively. Thus,

$$dis(\Psi, \Psi_1) = 1 - 2 \cdot 4 / (6 + 12) = 5 / 9,$$

$dis(\Psi, \Psi_2) = 1 - 2 \cdot 4 / (6 + 10) = 1 / 2.$

For the original program Ψ, there are two branches B_1 and B_2. Hence,

$range(\Psi, B_1) = \{S_1, S_2, B_1, B_2\},$

$range(\Psi, B_2) = \{S_1\},$

$N\text{-}Scope(\Psi) = (4 + 1) / (4 + 1 + 5) = 1 / 2.$

For the obfuscated program Ψ_1, four branches are contained. Corresponding values are as follows.

$range(\Psi_1, P_1^T) = \{B_1\},$

$range(\Psi_1, B_1) = \{B_2, S_1, P_1^T, S_2, B_1\},$

$range(\Psi_1, B_2) = \{S_1\},$

$range(\Psi_1, P_2^F) = \{S_{01}\},$

$$N\text{-}Scope(\Psi_1) = \frac{1+5+1+1}{1+5+1+1+9} = \frac{8}{17},$$

$$pot(\Psi, \Psi_1) = (\frac{8}{17} \Big/ \frac{1}{2}) - 1 = -\frac{1}{17}.$$

For the obfuscated program Ψ_2, three branches are contained. Thus,

$range(\Psi_2, P_1^F) = \{\mathrm{E}(S_{01})\},$

$range(\Psi_2, B_1) = \{B_2, S_1, D_1, S_2, B_1\},$

$range(\Psi_2, B_2) = \{S_1\},$

$$N\text{-}Scope(\Psi_2) = \frac{1+5+1}{1+5+1+8} = \frac{7}{15},$$

$$pot(\Psi, \Psi_2) = \frac{7/15}{1/2} - 1 = -\frac{1}{15}.$$

The abilities against reverse engineering provided by $\mathcal{T}_1$ and $\mathcal{T}_2$ are derived as follows.

$\mathcal{DP}_{T1}(\Psi) = (5 / 9, - 1 / 17),$

$\mathcal{DP}_{T2}(\Psi) = (1 / 2, - 1 / 15).$

The negative potency values indicate that after these two transformations, the structures of the obfuscated programs are not as complex as that of the original program. It implies that these

transformations may fail from the perspective on the complexity of program structure. Compared with $\mathcal{DP}_{T2}$, the value of $\mathcal{DP}_{T1}$ is a little larger. Hence, $\mathcal{T}_1$ provides the stronger robustness than $\mathcal{T}_2$.

The space penalty caused by $\mathcal{T}_1$ can be estimated as

$$0 + \overline{\mathcal{S}} + \overline{\mathcal{B}} + \overline{\mathcal{B}} = \overline{\mathcal{S}} + 2 \cdot \overline{\mathcal{B}},$$

where $O^2_{Ss}(\Psi, S_0)$ results in no overheads, $O_D(\Psi, B_1)$ leads to $\overline{\mathcal{S}}$, $O^t_{Op}(\Psi, B_1)$ leads to $\overline{\mathcal{B}}$, and $O^f_{Op}(\Psi, S_{01})$ leads to $\overline{\mathcal{B}}$. On the other hand, the space penalty caused by $\mathcal{T}_2$ is

$$0 + \overline{\mathcal{B}} + 0 + \overline{\mathcal{S}} = \overline{\mathcal{B}} + \overline{\mathcal{S}},$$

where $O^2_{Ss}(\Psi, S_0)$ and $O_E(\Psi, S_{01})$ results in no overheads, $O^f_{Op}(\Psi, S_{01})$ leads to $\overline{\mathcal{B}}$, and $O_D(\Psi, B_1)$ leads to $\overline{\mathcal{S}}$.

Chapter 9

Conclusion

We present an approach to evaluating and analyzing control flow obfuscating transformations. In our approach, we formalize the transformations and develop a new evaluation measure. In formalizing the transformations, we first analyze and parse a source program which is going to be obfuscated into sequences of code elements and its directed graph. With analysis of the transformations, we design the atomic operators which are the basic components of any obfuscating transformation. On the basis of the formal representation, we develop an applicable evaluation measure to estimate the robustness against reverse engineering after obfuscation. The measure provides sensitive results in terms of software complexities and differences.

Nevertheless, if we consider the side effects of obfuscating a software program that code size will be increased and execution performance will be slowed down, we need to carefully determine the obfuscation criteria for different applications and make a proper compromise between the security and overhead. In our method, space penalty is also evaluated such that a tradeoff between the robustness and the overheads can be well judged.

So far, we have quantized the degree of the robustness of obfuscating transformations. In the future, we will figure out the relationship between the $\mathcal{DP}$ values and the time spent on reversing engineering obfuscated programs. This helps understand the effectiveness of software obfuscations under the real attacks.

References

[1] M. D. Preda and R. Giacobazzi, "Control Code Obfuscation by Abstract Interpretation," In Proc. 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM'05), pp. 301-310, September 2005.

[2] T. W. Hou, H.Y. Chen and M. H. Tsai, "Three Control Flow Obfuscation Methods for Java Software," *IEE Proceedings Software*, volume 153, No. 2, pp.80-86, April 2006.

[3] B. Pfitzmann and M. Schunter, "Asymmetric Fingerprinting," EUROCRYPT 96, pp. 84-95, 1996.

[4] W. Cho, I. Lee, and S. Park, "Againt Intelligent Tampering: Software Tamper Resistance by Extended Control Flow Obfuscation," In Proc. World Multiconference on Systems, Cybernetics, and Informatics. International Institute of Informatics and Systematics, 2001.

[5] C. Wang, J. Hill, J. Knight, and J. Davidson, "Software Tamper Resistance: Obstructing Static Analysis of Programs," Technical Report CS-2000-12, December 2000.

[6] C. Collberg and C. Thomborson, "Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection," *IEEE Transactions on Software Engineering*, volume 28, No. 8, August 2002.

[7] D. Low, "Java Control Flow Obfuscation," Master Thesis, University of Auckland, 1998.

[8] C. Collberg, C. Thomborson and D. Low, "A Taxonomy of Obfuscating Transformations," Technical Report, 1997.

[9] J. M. Memon, Shams-ul-Arfeen, A. Mughal and F. Memon, "Preventing Reverse Engineering Threat in Java Using Byte Code Obfuscation Techniques," In Proc. 2nd International Conference on Emerging Technologies, pp. 689-694, November 2006.

[10] H. Zuse, "*Software Complexity: Measures and Methods*," Watler de Gruyter, New York,

1991.

[11] H. Bunke and K. Shearer, “A Graph Distance Metric Based on the Maximal Common Subgraph,” *Pattern Recognition Letters*, volume 19, issue 3-4, pp. 255-259, 1998.

[12] W. D. Wallis, P. Shoubridge, M. Kraetz and D. Ray, “Graph Distances Using Graph Union,” *Pattern Recognition Letters*, volume 22, issue 6-7, pp. 701-704, 2001.

[13] D. Aucsmith, “Tamper-Resistant Software: An Implementation,” In Proc. 1st International Workshop on Information Hiding (Lecture Notes in Computer Science), volume 1174, pp. 317–333, 1996.

[14] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, “Architectural Support for Copy and Tamper Resistant Software,” In Proc. 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), pp.168–177, November 2000.

[15] C. Collberg, C. Thomborson, and D. Low, “Breaking Abstractions and Unstructuring Data Structures,” In Proc. International Conference on Computer Languages, pp. 28–38, May 1998.

[16] C. Collberg, C. Thomborson and D. Low, “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs,” In Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL’98), PP. 184-196, January 1998.

[17] SOFTPEDIA.
http://news.softpedia.com/news/FairUse4WM-Kills-Microsoft-DRM-34206.shtml

[18] afterdawn.com. http://www.afterdawn.com/news/archive/7875.cfm

[19] S. R. Subramanya and B. K. Yi, “Digital Rights Management,” *IEEE Potentials Magazine*, volume 25, issue 2, pp. 31-34, 2006.

[20] Y. Nishimoto, A. baba, T. Kurioka and S. Namba, “A Digital Rights Management System for Digital Broadcasting Based on Home Servers,” *IEEE Transactions on Broadcasting*, volume 52, issue 2, pp. 167-172, June 2006.

[21] X. Wang, "MPEG-21 Rights Expression Language: Enabling Interoperable Digital Rights Management," *IEEE Multimedia*, volume 11, issue 4, pp. 84-87, 2004.

[22] T. Wikinson, D. Hearn and S. Wiseman, "Trustworthy Access Control with Untrustworthy Web Servers," In Proc. 15th Annual Computer Security Applications Conference (ACSAC'99), pp. 12-21, 6-10 December 1999.

[23] N. Komninos, B. Honary and M. Darnell, "Security Enhancements for A5/1 Without Loosing Hardware Efficiency in Future Mobile Systems," In Proc. 3rd International Conference on 3G Mobile Communication Technologies, pp. 324-328, 8-10 May 2002.

[24] J. Reid, J. M. G. Nieto, E. Dawson and E. Okamoto, "Privacy and Trusted Computing," In Proc. 14th International Workshop on Database and Expert Systems Applications, pp. 383-388, 1-5 September 2003.

[25] Z. D. Shen, F. Yan, W. Z. Qiang, X. P. Wu and H. G. Zhang, "Grid System Integrated with Trusted Computing Platform," In Proc. 1st International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06), pp. 619-625, 20-24 June 2006.

[26] B. Balacheff, L. Chen, S. Pearson, D. Plaquin, and G. Proudler, *Trusted Computing Platforms – TCPA Technology in Context*, Prentice Hall, 2003.

[27] N. Komninos, B. Honary and M. Dernel, "An Efficient Stream Cipher Alpha1 for Mobile and Wireless Devices," In Proc. 8th IMA International Conference on Cryptography and Coding, pp. 294-300, 2001.