Software obfuscation by CFI-hiding scheme and self-modifying scheme

Jun Ge
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Recommended Citation
https://lib.dr.iastate.edu/rtd/20426

This Thesis is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Software obfuscation by CFI-hiding scheme and self-modifying scheme

by

Jun Ge

A thesis submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Soma Chaudhuri, Co-major Professor
Akhilesh Tyagi, Co-major Professor
Thomas E. Daniels

Iowa State University
Ames, Iowa
2004
Copyright © Jun Ge, 2004. All rights reserved.
Graduate College
Iowa State University

This is to certify that the Master's thesis of

Jun Ge

has met the thesis requirements of Iowa State University

Signatures have been redacted for privacy
# TABLE OF CONTENTS

ABSTRACT ........................................................................................................... v

CHAPTER 1. INTRODUCTION ............................................................................. 1

CHAPTER 2. BACKGROUND ............................................................................. 5

2.1 Malicious Hosts Attack on Intellectual Property ..................................... 5

2.2 Malicious Reverse Engineering and Software Obfuscation ................... 6

2.3 Software Obfuscation Classification .......................................................... 7

2.4 Software Obfuscation Measures ................................................................. 8

2.5 Related Work ............................................................................................... 11

CHAPTER 3. HIDING THE CFI INTO THE MONITOR-PROCESS ............... 14

3.1 Basic Structure .............................................................................................. 15

3.2 Implementation Steps .................................................................................. 17

3.2.1 Get Profiling Information for Basic Blocks ........................................... 18

3.2.2 Choose Hot Blocks and Decide Hot Nodes ........................................... 19

3.2.3 Shuffling the Hot Nodes ........................................................................ 20

3.2.4 Generate the M-process ......................................................................... 22

3.3 Correctness of the CFI-hiding Scheme ....................................................... 23

3.4 Resilience of the CFI-hiding Scheme .......................................................... 24

3.4.1 Acceptance Problem in Linear Bounded Turing Machine .................... 25

3.4.2 Reachability Problem for the M-process ............................................ 26

3.4.3 MPROCESS-REACH is in PSPACE-complete .................................. 27

3.4.4 DTOM and SCOM of the Self-modifying Scheme ............................... 29
CHAPTER 4. PROTECT THE MONITOR-PROCESS BY A SELF-MODIFYING SCHEME

4.1 Basic Structure .................................................. 32
4.2 Protection of Jump Table ....................................... 34
4.3 Encryption Key Assignment ..................................... 35
4.4 Implementation Steps ........................................... 39
  4.4.1 Decide the Cells ............................................ 40
  4.4.2 Key Generation ............................................. 42
  4.4.3 Function xorAll ............................................ 42
  4.4.4 Modification for Each Basic Block ....................... 44
  4.4.5 Copy Binary Code to the Code Array ................... 48
4.5 Correctness of the Self-modifying Scheme ..................... 50
4.6 Resilience of the Self-modifying Scheme ...................... 51

CHAPTER 5. EFFICIENCY OF THE OBFUSCATOR ....................... 52

CHAPTER 6. SUMMARY AND FUTURE WORK .......................... 56
  6.1 Summary ..................................................... 56
  6.2 Future Work .................................................. 57

BIBLIOGRAPHY ....................................................... 59

ACKNOWLEDGMENTS ................................................ 61
ABSTRACT

A software obfuscator is a program \( O \) to transform a source program \( P \) which we want to protect from malicious reverse engineering. \( O \) should be correct (\( O(P) \) has same functionality as \( P \)), resilient (\( O(P) \) withstands attacks), and effective (\( O(P) \) is not too much slower than \( P \)).

In this thesis we describe the design of an obfuscator which consists of two parts. The first part extracts the control flow information from the program and saves it in another process named Monitor-process. The second part protects Monitor-process by enabling it to modify itself continuously during its execution. By proving the correctness and showing that both the resilience and efficiency are high, we claim that our approach is practical.
CHAPTER 1. INTRODUCTION

In 2003, the Business Software Alliance (BSA)'s annual study on software piracy showed that, although the world piracy rate decreased slightly to 39%, $13.08 billion was lost due to piracy globally. To protect software intellectual property, computer scientists and software companies have invested a lot of money and energy.

As described by Collberg et al. [7], software protection has three main technologies: water­marking — to provide intellectual property rights, obfuscation — to make it hard to reverse engineer the high level program from the machine level code and/or partial observation of its black-box behavior, and tamper-resistance — to detect any tampering of the software and to activate some counter-tamper measures.

Informally speaking, Software Obfuscation means protecting a program from being observed while maintaining its functionality. However, a widely accepted definition of the obfuscator does not exist and different results are achieved with different definitions of it. Barak et al. [2] proved that an obfuscator does not exist if the obfuscated program is viewed as a “virtual black box” which means that the obfuscated program reveals nothing more than its functionality. Although this impossibility result may imply the nonexistence of an obfuscator, the definition of the obfuscator as a “virtual black box” is very restrictive. With less restricted definitions, obfuscators do exist and several researchers have achieved positive results.

In this thesis, we define an obfuscator as a program $\mathcal{O}$ with certain properties. $\mathcal{O}$ takes the source program (the program that is to be protected) $P$ as its input and generates a protected program $\mathcal{O}(P)$. $\mathcal{O}$ should has three properties: correctness, efficiency and resilience. Informally, correctness means that $\mathcal{O}(P)$ has the same functionality as $P$, efficiency means that $\mathcal{O}(P)$ is
not much slower than $P$, and \textit{resilience} means that $O(P)$ tolerates reverse engineering.

Considering the information being obfuscated, software obfuscation technology can be divided into three categories. \textit{Lexical Obfuscation} typically tries to scramble the identifiers and is used in software like KlassMaster [15] and JZipper [13] for Java program protection. \textit{Data Obfuscation} modifies data structures to gain security [6]. The last category, \textit{Control Flow Obfuscation}, hides the Control Flow Information (CFI) of the program and is of interest of most researchers. ClockwareTM developed [3] a dispatcher which takes charge of the CFI of the protected program. If an adversary wants to statically analyze the protected program, he or she needs to be able to solve a reachability problem. They proved that the reachability problem for the dispatcher is PSPACE-complete [24], so the dispatcher prevents static attacks. Collberg \textit{et al.} [5] proposed another way to obfuscate the control flow of an application by inserting irrelevant conditionals and loops.

In our work, we design a novel software obfuscator consisting of two parts. The first part focuses on hiding the CFI of the source program $P$. $P$ is compiled into two co-processes: a main program (P-)process and a monitor (M-)process. The code to accomplish the functionality of the $P$ resides with the P-process and the key components obfuscation resides with the M-process. Part of the CFI in the P-process is extracted and moved to the M-process. During its execution, the P-process sends an address request to the M-process whenever it meets a point where the CFI is missing. M-process’s only task is to return the appropriate address anytime it gets a request from the P-process. After the P-process receives the address, it continues to execute to fulfill $P$’s functionality.

We claim that the CFI-hiding scheme above protects $P$ from static attacks, in which the adversary analyzes the program without executing it. Two methods are used to evaluate its resilience. The first way is similar to the one used in Zakharov’s [24] work: the reachability problem in $O(P)$ is proved to be PSPACE-Complete. The second method is to calculate metrics for it. Up to now there exist no widely accepted metrics for evaluating an obfuscator, so we defined two measures DTOM and SCOM for static obfuscation and evaluate both of them for
the CFI-hiding scheme.

The second part of our work protects M-process with a self-modifying scheme. The idea is to hide most code of the M-process by enabling it to modify itself continuously at run-time. The M-process is compiled in such a way that it is divided into several cells (group of basic blocks) of the same size. Each cell is encrypted with one or several keys. When M-process is initially loaded into the memory, only the entry cell (the first cell being executed) is opened in plain text and all the others are encrypted. During the execution, every time the control flow goes from a cell $C_a$ to another cell $C_b$, the M-process (the text section of its image in the memory) modifies itself, encrypts $C_a$, decrypts $C_b$, and continues executing the instructions in $C_b$. During the self modification, cells other than $C_a$ and $C_b$ are modified but still kept encrypted. With this scheme, the M-process executes correctly with most of its code hidden.

The main purpose of the self-modifying scheme is to protect software from dynamic reverse engineering. Dynamic reverse engineering is a widely used attack in which the adversary runs the obfuscated program manually. Without assuming any secure components provided by hardware, dynamic attacks are hard to prevent. With the self-modifying scheme, because it guarantees that only a small part (one cell) of the M-process is observable at any time during the execution, we claim that this scheme deters dynamic attacks. Intuitively, the adversary needs to run the M-process many times to expose the binary code for all the cells. On the other hand, for an unprotected program, they only need to run it once to observe all of its binary code. Formally, a metric name BOOM is defined and evaluated for this scheme in this thesis.

The implementation of our obfuscator has two kinds of compiler passes and several Perl scripts for data analysis. The first kind of passes are Machine-SUIF [17] passes whose main purpose is to get the Control Flow Graph Intermediate Representation of the program and add annotations into the assembly code marking out basic blocks. The second set of passes are written in Perl and they take care of all the code profiling, code modification and data procession.

The rest of this thesis is organized as follows. Chapter 2 provides a background of the
subject of Intellectual Property protection and Software Obfuscation. It also defines several important terms such as obfuscator, resilience and efficiency. Metrics for obfuscators are also defined. Chapter 3 describes the CFI-hiding scheme. After the basic structure is introduced, we give a detailed explanation of the implementation. The correctness of the scheme is proved and the resilience is evaluated in the last two sections of this chapter, respectively. Chapter 4 discusses the self-modifying scheme which is used to protect M-process. First we prove the existence of the keys used in the encryption and present the detailed implementation. The correctness of the scheme is proved in Section 4.5, and the resilience of it is evaluated in Section 4.6. Chapter 5 evaluates the efficiency of the whole obfuscator. Both the space cost and the time cost metrics are evaluated with empirical methods. Finally, Chapter 6 summarizes the main points of the thesis.
CHAPTER 2. BACKGROUND

In this chapter we present the background of attacks on intellectual property and the efforts against the attacks. Section 2.1 introduces several kinds of malicious hosts attacks. Section 2.2 defines software obfuscator which is used to protect software from malicious reverse engineering. The classes of software obfuscators are listed in Section 2.3. Several metrics for evaluating software obfuscators are defined in Section 2.4. Finally, a summary of the related work is presented in Section 2.5.

2.1 Malicious Hosts Attack on Intellectual Property

Software development is time- and money-consuming. Infringement of intellectual property hurts software vendors and is very harmful to software industry. Taking software piracy as example, around forty percent of business application software is installed without a license every year (see BSA Global Software Piracy Study 1994-2002).

As the Internet becomes more prevalent, more and more software is downloaded and has become more accessible. Nowadays most attacks on intellectual property are carried out by malicious hosts. That is, the attackers run the application on their own machine over which they have full control. For example, Java applets are downloaded to run on local machines. If the user is attacking, he can take his time to analyze the applets, with help of debuggers, specialized software and hardware analysis tools.

There are three main kinds of malicious host attacks.

- Software Piracy: Unauthorized duplication, use and/or resale of computer software.
• **Malicious Reverse Engineering**: The process of analyzing the executable of computer software to figure out the design of one or more modules of it and how the modules operate.

• **Software Tampering**: The process of tempering with important information, such as an encrypted key and the control flow information, in an application.

Collberg [7] performed an overview of all the three kinds of attacks. In this thesis, we focus on the software protection against the second kind of attack, malicious reverse engineering.

### 2.2 Malicious Reverse Engineering and Software Obfuscation

In order to protect their product from malicious reverse engineering, software vendors only release binary executables in most cases. From the binary code, the attackers try to recover parts of the original application so that they can reuse them in their own products. This process of module recovery is malicious reverse engineering. One efficient way to protect software from malicious reverse engineering is **Software Obfuscation**, a kind of code transformation [18]. Software Obfuscation transforms an executable into a new one which has the same functionality but is hard to be reverse engineered.

An obfuscator should have three important properties. The first is that it should keep the functionality of the program being protected; the second is that the obfuscated program should not be too much slower than the original one; the third is that it should take the attackers more time to reverse-engineer the application than developing a new one by themselves. Here we give the definition of Software Obfuscator. It is modified from [4] and [2].

**Definition 1 (Software Obfuscator)** An algorithm $\mathcal{O}$ is a **Software Obfuscator** if the following three conditions hold:

• *(Functionality)* For any given source program $P$, $\mathcal{O}$ transforms $P$ into another protected program $P'$ such that
If $P$ fails to terminate with an error condition, then $P'$ may or may not terminate.

Otherwise, $P'$ must terminate and produce the same output as $P$.

If the functionality is maintained, we say that $O$ is correct.

- (Efficiency) The size and the running time of $P'$ should be at most polynomially larger than the size and the running time of $P$, respectively.

- (Resilience) The time taken for the attackers to recover $P$ from $P'$ should be at least larger than developing $P$ from scratch.

2.3 Software Obfuscation Classification

Based on what elements are obfuscated, software obfuscation could be divided into three categories [7], as listed in Table 2.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Example</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical Obfuscation</td>
<td>Simply scramble identifiers</td>
<td>Some Java obfuscation tools</td>
<td>Annoying to a reverse engineer but that’s all</td>
</tr>
<tr>
<td>Data Obfuscation</td>
<td>Obfuscate data structure</td>
<td>(1)Change encoding of variables. (2)Split Variables</td>
<td>Make an expression quite different with the original one</td>
</tr>
<tr>
<td>Control Flow Obfuscation (our choice)</td>
<td>Hide the control flow information</td>
<td>(1)Add redundant code into a never referred branch. (2)Randomize the order of the computation. (3)Break up computation that logically belong together or merge those are not</td>
<td>Really frustrating if successfully implemented. The most powerful of the three transformation types</td>
</tr>
</tbody>
</table>

Although the Lexical Obfuscation and Data Obfuscation frustrate attackers to some extent, Control Flow Obfuscation is by far the most resilient obfuscation method, because most attack efforts are spent on the analysis of control flow. Control Flow Obfuscation falls into two main categories. The first category introduces spurious code blocks into the source program to
obscure the real CFI. Collberg et al. [5] implemented such an obfuscator. In their work opaque constructs are manufactured to embed intractable static analysis problems into the obfuscated program. The second category extracts the CFI from the code and hides it somewhere else. In Stanley Chow etc.’s work [3], the obfuscator divides the source program into several pieces, adds new dummy chunks into it and generates a dispatcher taking charge of the control flow between all the code pieces. They have proved that the CFI-hiding dispatcher is resilient static analysis.

2.4 Software Obfuscation Measures

To evaluate an obfuscator, we focus on its functionality, efficiency and resilience. First of all, it is a requirement that an obfuscator maintain the functionality of its input, the source program. That is, we need to show its correctness. In a CFI obfuscating scheme, we could prove that the obfuscation scheme is correct by showing that the control flow is preserved during the process of obfuscating.

For efficiency, we use the simple definitions of cost (adapted from [4]) to weigh separately the extra execution time and space of $O(P)$ compared to $P$.

**Definition 2 (Obfuscator Cost)** Given an obfuscator $O$ and a program $P$, the **Time Cost** $C_t(O, P)$ and the **Space Cost** $C_s(O, P)$ are defined as

$$C_t(O, P) = \frac{T(O(P))}{T(P)} - 1$$

and

$$C_s(O, P) = \frac{S(O(P))}{S(P)} - 1$$

where $T(M)$ is the execution time of program $M$ and $S(M)$ is the space taken by $M$.

Finally let us consider resilience, the most important property of an obfuscator. In the rest of this section, we propose several resilience metrics. Our discussion is based on the attack scheme deployed by the adversary. The reverse-engineering analysis methods used by the attackers can
be divided into two categories: static and dynamic. In a static attack, the adversary analyzes the obfuscated program without running it, while in a dynamic one, they run it manually, trace the control flow and data flow of the program, and try to recover the source program.

The purpose of obfuscation against static attack is to make it difficult for the adversary to reconstruct the static layout of the program. The overall goal, of course, is to determine the control behavior of the program (which can lead to the high level algorithmic intellectual property and private data). The larger control flow behavior is however constructed bottom-up through piecewise reconstruction of the control flow within a program segment (an intermediate program region with a few well defined entry points, and a few well defined exit points such as entry into a specific procedure and exit points from a procedure).

We propose two metrics that attempt to capture the difficulty of inferring instantiated control flow paths in a program from the static code layout. Our methodology is control-flow-centric in as much as it is assumed that the adversary’s primary objective is to determine the control flow of the program. In some sense, all of the obfuscation techniques such as static code permutation or data transformations are obfuscating the true control flow of the program.

The first metric is decision theoretic in nature. It measures the number of decisions required before an adversary can reconstruct the true control flow of the program. It is reminiscent of the classical decision theoretic lower bounds on the sorting algorithms’ execution time. Note that each decision is binary (whether a given basic block is a successor of a given basic block or not). The basic intuition is that the number of dynamic binary decisions an adversary needs to make is log of the number of possible instantiated paths as observable from the static code layout.

**Definition 3 (Decision Theoretic Obfuscation Measure)** Let $N$ be the number of possible instantiated control paths as observable from the static program layout. The decision theoretic obfuscation measure (DTOM) is defined as

$$k = \log(N)$$
DTOM captures the number of decisions (the complexity of the reverse engineering procedure deployed by the adversary).

The second measure, on the other hand, takes more of a combinatorial viewpoint. In the second metric we try to capture the size of the search space that the adversary would have to explore for reverse engineering. Specifically, the adversarial attack is modeled as determining the successor of each basic block one at a time. Such an attack model is not too far from reality wherein an attacker tries to build the Control Flow Graph (CFG) within a small CFG aperture (within a procedure or closer to the program where private data is held). In such a situation, a simple measure would be the size of the search space that the adversary needs to explore. The problem is even more focused if we only consider one basic block (and its branch instruction) at a time. In a traditional static program layout, each branch instruction has two successors. However, one of these successors is already fixed (the fall-through path where, when the branch is not taken, the next basic block in the static sequence is executed). The other successor for a taken branch could be any basic block in the program. In practice again there would be tricks to limit the number of potential taken successors (for instance, in a typical program over 90% of branches jump to a location within ±2^{15} of the current location. Hence, an adversary could consider focusing on basic blocks within this neighborhood.

**Definition 4 (Successor Count Obfuscation Measure)** Let $k$ basic blocks occur along a path to be exposed. Let the number of taken successors for each of these basic blocks be $T$. The successor count obfuscation measure (SCOM) is defined as the sum of all successors along this path: $k * T$.

Another way to evaluate an obfuscator's resilience against static attacks is to show that some problems involved in the static analysis is in some hard problem set. For example, Zakharov [24] reduced a PSPACE-Complete problem to the reachability problem of a cloak program dispatcher, which hides the control flow information of the source program. To fully reverse-engineer an obfuscated program, the attacker should be able to decide whether $s'$ is reachable from $s$, where $s$ and $s'$ is an arbitrary pair of states in the obfuscated program. Since the
reachability problem is proved hard, it follows that the obfuscator protects the program from static attack. Similar method is deployed in [20]. In her Ph.D dissertation, Wang developed an one-way transformed compiler which hides the static control-flow graph of the source program and showed that analyzing the the transformed program is NP-hard. We also use this method to show the power of our obfuscator in Section 3.4.3.

For the obfuscation against dynamic attacks, resilience is difficult to evaluate because adversary's interactions in reverse-engineering are hard to predict and model. There exist no metric that is widely accepted. One empirical measure is proposed in Wroblewski’s work [21]: several groups of people tried to understand the obfuscated program and the time taken is looked as the resilience of the obfuscator.

We propose a new metric Binary Observation Obfuscator Metric which describes the difficulty of revealing all the binary code for the obfuscated program. In a dynamic attack, the attacker runs the program with tracing tools like debuggers and software analysis tools such as SoftIce. When the program is loaded into the memory, the binary code is observed and disassembled. If the binary code is not protected at all, all the code is ready to be analyzed once it is loaded into memory. However, if the binary code is protected in some way so that not all of the code is open during the execution, the dynamic analysis is impeded to some extent.

**Definition 5 (Binary Observation Obfuscator Metric)** Given an obfuscated program $P' = O(P)$, let $n$ be the number of basic blocks in $P$ and $r$ be the smallest integer such that, by running $P'$ for $r$ times, the adversary can observe all the binary code of $P'$. The binary observation obfuscator metric (BOOM) is defined as $r/n$.

### 2.5 Related Work

The related work in software obfuscation has been discussed in the previous sections in this chapter. Here is a summary of it. Comparisons between our work and the existing approaches are presented whenever possible.
Collberg et al. [7] [4] made a survey on software protection and software obfuscation. They classified the software obfuscation into three categories (as described in section 2.3), presented several obfuscation measures, and listed a number of technologies used in software obfuscation. In our work, besides the efficiency metrics Time Cost and Space Cost which are adapted from their work, DTOM, SCOM and BOOM metrics are developed for evaluating both static and dynamic obfuscation.

Most research on software obfuscation has been focused on control flow obfuscation. Collberg et al. [5] proposed to obfuscate the control flow of the source program by constructing opaque predicates. Opaque predicates are program predicates with certain properties which are hard for the attackers to analyze. Cloakware™ [3] [24] developed a dispatcher to take charge of the control flow information at run-time. To argue the resilience of their obfuscator, they reduced a PSPACE-Complete problem to the reachability problem in the process of analyzing the obfuscated program (see Section 2.4). Wang [20] presented a suite of code transformation at intra- and inter-procedural level to deter static analysis. She provided an NP-complete proof to argue that in a general case, determining indirect branch targets statically for a transformed program is an NP-complete problem. Similar approach to prove the resilience of the obfuscator is deployed in this thesis (see Section 3.4).

All the work listed above focused on static obfuscation. To protect a software from dynamic attacks, Aucsmith [1] designed an Integrity Verification Kernel (IVK) which is a critical code segment consisting of a number of cells. At run-time, all the cells are encrypted except for the one that is currently executing. Because the attackers can only observe part of the executable, IVK deters dynamic attacks to some extent. Although the author presented the basic idea of IVK, he did not provide a detailed method to manipulate the IVK. In our work, a self-modifying scheme is developed to protect the M-process. We formalize an open-close property (see Section 4.3) which is vital for the keys used in this scheme. For any given program, the existence of a set of keys with the open-close property is proved by construction.

Other work of interest is Barak et al.’s research on the impossibility of obfuscating programs.
They proved that an obfuscator does not exist if the obfuscated program is viewed as a "virtual black box" (see Section 2.4).
CHAPTER 3. HIDING THE CFI INTO THE MONITOR-PROCESS

Our approach to software obfuscation focuses on Control Flow Obfuscation and consists of two parts. The first part hides the CFI of the source program by compiling it into two co-processes: a main program (P-)process and a monitor (M-)process. The two processes together are the obfuscated program. The P-process implements the main functionality and it is similar to the source program. Part of P-process' CFI is extracted and stored in a table named the Jump Table, which is not accessible to P-process directly during the execution. M-process takes charge of the Jump Table and the P-process communicates with M-process whenever it reaches a point where the CFI is obfuscated.

Chapter 4 will discuss the second part of our obfuscator, where a self-modifying scheme is deployed to makes sure that only a small part of the code is observable by the adversary. In the rest of this section, we discuss the CFI-hiding scheme. The basic structure is introduced in Section 3.1. It includes how the source program is divided into groups of blocks, how the CFI is extracted by shuffling the blocks, and how the P- and M-processes collaborate. With this structure, we can claim that the obfuscated program is correct. Section 3.2 describes in detail the implementation, which comprises how the source program is profiled, how the hot nodes (defined later) are selected and shuffled, and how P-process generates M-process. Finally, we evaluate the CFI-hiding scheme in Section 3.3, claiming its correctness and showing its resilience. The evaluation of the whole obfuscator, which combines the CFI-hiding scheme and the self-modifying scheme, is described in Chapter 5.
3.1 Basic Structure

The basic structure of our CFI-hiding scheme is illustrated in Figure 3.1. Several definitions are given first.

**Definition 6 (Hot Node)** A Hot Node is one or more consecutive basic blocks that are chosen to be shuffled as a sequential static unit in the obfuscation. The size of a Hot Node is the number of the basic blocks in the hot node.

**Definition 7 (Hot Block)** A basic block $B$ in $P$-process is called a Hot Block if $B$ is executed with high frequency during the execution of $P$-process and is chosen as the start block of a Hot Node.

At shown in Figure 3.1, there are four steps in the implementation. At first, we profile the source program because such information helps us to hide important CFI. For example, the information on frequently instantiated control flow is found in this step. With the help of Machine-SUIF ([17], [10], [23]), an infrastructure for constructing compiler back ends, we developed passes to get the profiling information. We also make sure that the assembly code generated contains only label addresses instead of relative or absolute addresses. The reason will be explained later.

The second step is to choose the hot blocks and hot nodes. Based on the profiling information, we select a set of hot blocks. The selection of hot nodes is based on the hot blocks and other information such as requests from the owner of the source program. The owner chooses the number of blocks s/he wants to obfuscate the software. Note that the number of hot blocks affects the efficiency of the obfuscator. This will be discussed in detail in Section 3.2.2.

The third step is to shuffle the hot nodes so as to hide the CFI of the source program. The shuffling is performed at assembly code level. Hot nodes are shuffled randomly among themselves. The new addresses of the hot nodes are the CFI to be hidden and they are recorded in the Jump Table. Certain codes are inserted at the boundaries of the hot nodes to request CFI from M-process. Except for these modifications on the hot nodes, all other parts of the
Figure 3.1 Hide the CFI into Monitor-process
source program are not changed. After extracting the CFI from the original process, we get the code for P-process.

The last step is the development of the M-process. M-process is initialized right after P-process is started, then it enters a finite loop waiting for P-process' address (CFI) requests. M-process has exclusive access to the Jump Table. Just before the P-process terminates, the M-process also exits. In this chapter, we only discuss the functionality of M-process. In Chapter 4, the self-modifying scheme is discussed in detail and is shown to be correct. In the rest of this chapter, we assume that M-process acts properly and always returns the correct CFI.

3.2 Implementation Steps

Now let us describe the detailed implementation of the CFI-hiding scheme. In our implementation, we developed several compiler passes to fulfill the task of program obfuscation. Most passes are generated in the following way. At first, the CFG Intermediate Representation (IR) of the source program is generated with the help of Machine-SUIF. Then annotations are added at proper locations of the CFG IR such as the beginning and the end of each basic block. The last task performed in Machine-SUIF was to generate the assembly code with the annotation. Generally each annotation is identified with the unique ID of the basic block to which it is attached. One thing to note is that the assembly code is generated in such a way that all the addresses of Control Transfer Instructions (CTI, such as jmp, jc) are label addresses. This is a requirement because the shuffling of the nodes will modify the absolute addresses of many blocks and consequently, the relative addresses between blocks. Thus absolute or relative addresses won't work with our code shuffling scheme. However, although a label may be moved, its address will be decided later by the linker and loader, so label addresses still work after the code shuffling.

Several other passes were developed in Perl to manipulate the assembly code generated by Machine-SUIF passes. All the code insertion, modification and shuffling are completed in Perl except for the SUIF inserted annotations as mentioned above. Perl was chosen because of its
powerful text processing and regular expression support.

We also developed several other Perl scripts to process the data generated during the process of obfuscation. For example, the profiled program generates profiling information but that information is not ready for use, so a script was written to refine the data into a more manageable format.

3.2.1 Get Profiling Information for Basic Blocks

The idea is as follows: Function calls are added at the beginning of each basic block. Every time a block \( b \) is accessed during the execution, the function at the beginning of \( b \) is called once so that the profiling information could be recorded.

The first stage of the profiling process is to annotate the source program. We get the CFG Intermediate Representation (IR) of the source program in Machine-SUIF at first. Then another Machine-SUIF pass attaches annotations to the beginning of each basic block. Each annotation is with an integer which is the unique ID of the block to which it is attached. A special block is the first block of the main function and it is marked by a special annotation. This point is marked specially because it is the place to fulfill the initialization task such as opening files and forking a new process.

After the annotation is added, the IR is processed to generate assembly code. Then another Perl pass profile.pl scans through the annotated assembly code to insert function calls. The annotations are used as markers of the basic blocks and two kinds of function call instructions are inserted. At the beginning of the main function, an instruction call _prfStart is added. What _prfStart does is to open a file for profiling information recording. At the beginning of all other blocks except for the first one in main, two instructions pushl $id and call _excOnce are inserted. $id is the block ID in the annotation and it is the argument of function _excOnce, which writes an entry in the profiling file opened by _prfStart.

Before and after _prfStart is called, a certain set of instructions are added to save and restore registers respectively. There are also codes for cleaning the arguments from the stack.
after the function calls. The purpose of these instructions is to recover possible register modification done by \texttt{prfStart} and \texttt{excOnce} and guarantee a proper execution of the instructions after the function calls.

After the code insertion, the binary is generated from the assembly code and is executed for a certain number of times. Then we get a file which contains the raw profiling information, which will be reformatted by another script. Now the profiling information is ready for use. The number of times that the binary is executed can be assigned by the user of the obfuscator.

### 3.2.2 Choose Hot Blocks and Decide Hot Nodes

The first $n$ most frequently accessed blocks are chosen as hot blocks. They are the first blocks in the hot nodes.

The decision of how to pick a hot block is important for both obfuscation and performance. Our goal is to maximize both resilience and efficiency. First, we need to choose a proper number of hot nodes. As this number increases, higher level of obfuscation is achieved, but high overhead is incurred as well. The best way is to let the user be the decision maker. Let \textit{shuffling rate} denote the ratio of the number of hot nodes to the number of total blocks. The user is able to decide the shuffling rate.

Secondly, a node should have some limits on its position and size. First, a node should not trespass the border of functions. That is, all blocks of a node should all relate to the same function. Second, a node should not contain more than one hot block. Otherwise the relation between the two hot blocks is not hidden, which is not consistent with our philosophy.

Finally comes the decision of the node size. The size of a node should be a function of the node’s properties. In our current implementation, a node’s size is decided by three factors. The first factor is $r$, the node’s rank in the profiling information table. The second factor is $d$, the max possible size of the node. That is, from the start hot block to the next block right before the next hot block or to the last block in the function. The third factor is $n$, the number of hot nodes chosen. Let $s(b)$ denote the size of the hot node whose first block is $b$. $s(b)$ is decided by
function

\[
s(b) = \min(d, \max(1, d(b) \ast r(b)/n))
\]

(3.1)

Let us discuss the properties of this function. Firstly \(s\) is proportional to \(r\), because the smaller the rank, the more frequently the node is accessed. Considering the locality of hot blocks, it is very possible that the blocks at the neighborhood of \(b\) are also accessed frequently. To obfuscate a program, we should hide the information between these hot blocks.

Secondly, \(s\) is proportional to \(d\). This favors a larger node. If a hot node is small and it is shuffled to a new location, it has very few relationships with its new neighbors and is easily distinguished. On the other hand, if we choose big nodes, a shuffled node contains several blocks and there is relationship in between these blocks. So it is hard to distinguish the hot nodes.

In our implementation, we first analyze the assembly code with annotations to learn the relation between functions and hot blocks and also the relation between hot blocks. Based on this information, the largest possible size for each hot node is calculated. Finally, the hot nodes’ sizes are calculated by Equation (3.1) so that the start and end blocks of each hot node can be determined.

3.2.3 Shuffling the Hot Nodes

The hot nodes are shuffled randomly. The hot nodes list derived from Section 3.2.2 is shuffled by the fisher-yates shuffling algorithm [16]. According to the original list and the shuffled one, we move around the hot nodes in the annotated source program at assembly code level.

To help the shuffling of the nodes, we generate another Machine-SUIF pass for annotation. This time each block is marked by two annotations, one at its beginning and one at its end. Each node \(N_i\) is marked with two annotation, \(stt_i\) and \(end_i\). After the annotation, the assembly code is processed by a script named \texttt{shfl.pl}. Table 3.1 is the pseudo-code \texttt{shfl.pl} generated at the location where \(N_i\) is originally located. In the shuffling, \(N_i\) is substituted by \(N_j\). Function \texttt{getAddr} is implemented in another source file whose task is to contact M-process for address queries.
Table 3.1 Pseudo-code After The Shuffling Where $N_i$ Is Substitute by $N_j$

1. push flags and registers onto stack
2. push $stti$ onto the stack, call function $\text{getAddr}$ and clean argument $stti$
3. jmp %eax
4. label $g_1.stt_j$:
5. restore flags and registers
6. code of $N_j$
7. push $end_j$ onto the stack, call function $\text{getAddr}$ and clean argument $end_j$
8. jmp %eax
9. label $g_1.end_i$:
10. restore flags and registers
11. original code after $N_i$

Before calling $\text{getAddr}$, flags and registers used by $\text{getAddr}$ are pushed onto the stack (Step 1). Then $\text{getAddr}$ is called with $stti$ as the argument (Step 2). After returning from $\text{getAddr}$, %eax holds the start address of $N_i$, so the jmp instruction in Step 3 directs the control flow to the new address of $N_i$.

After Step 3, codes similar to Step 4 - 8 (with $j$ substituted by $i$) are executed at $N_i$'s new location. Step 5 restores the flags and registers before executing the codes of $N_i$. After $N_i$'s code is finished (Step 6), the control flow should return to the code that originally follows $N_i$. This is done in Step 9 where $\text{getAddr}$ is called again with argument $end_i$. Then the flags and registers are restored at Step 10. At last the original code after $N_i$ is executed at Step 11. It is obvious that the control flows are the same before and after the shuffling. The only difference is that the addresses of the hot nodes are hidden after the shuffling.

Another issue is how to get the absolute addresses of the hot notes. As described above, labels are added to all the targets of instructions jmp %eax, which are generated in the shuffling.
After the shuffling, assembler generates an executable. It is from the executable that we get the absolute addresses of the labels. We could parse the executable directly, but it would take us too much effort because the binary format is complicated. Our workaround is **nm** in GNU’s **binutils**. **nm** prints all the information we need. We just parse its output with a script **getJmpTable.pl**. One possible vulnerability in this scheme is that the labels are still left in the executable so that they may give traces to the attackers and diminish our effort. There are easy ways to hide these labels though. One possible method is to rename the labels and keep the mapping of the old names and the new names in a file. Although the new labels are meaningless to the attackers, **getJmpTable** can still find the correct addresses with the help of the file containing the mapping information.

### 3.2.4 Generate the M-process

M-process’s only job is to handle address requests from P-process. Because it controls the crucial CFI in the Jump Table, it is protected by a self-modifying scheme. Chapter 4 will discuss the protection of M-process in detail. Here we only show M-process’s interface, i.e., its creation and activation in P-process and its functionality.

Instructions are added into the P-process to create the M-process and send requests. The code modification is done by the same script that shuffled the code, **shfl.pl**. M-process is initialized (forked) at the entry point of P-process’ **main**(). Recall that there is a special annotation there so that **shfl.pl** could insert an instruction call **_startMp** at the correct point. Function **_startMp** is implemented in a C program which creates a M-process.

Every time the M-process gets a block ID, it returns the address of the hot node which starts with the block from the Jump Table. At the P-process side, the boundaries of the hot nodes are the places where the CFI is hidden and instructions call **_getAddr** are inserted by **shfl.pl**, as described in Step 2 and Step 7 of Table 3.1. Like **_startMp**, **_getAddr** is a C function. It sends address requests to the M-process, receives the addresses from M-process, and returns these addresses. After **_getAddr**’s return, the address is stored in register %eax so
that instruction `jmp %eax` can redirect the control flow to the new addresses of the hot nodes. (See Step 3 and Step 8 in Table 3.1). The insertion of `call .getAddr` is described in Step 2 and Step 7 in Table 3.1.

### 3.3 Correctness of the CFI-hiding Scheme

In this section we prove the correctness of the CFI-hiding scheme. Section 3.4 will show its resilience. To show that the CFI-hiding scheme preserves the functionality of the source program, we claim that the node shuffling process doesn’t change the original control flow.

First let us assume that the M-process will always return the addresses stored in the Jump Table. We separate our whole obfuscator into two parts and will prove the self-modifying scheme’s correctness later in Section 4.5. The correct self-modifying scheme guarantees the correct execution of M-process. We consider all the effect of nodes shuffling. If we can show that all the modification such as cell movements and instructions insertion won’t change the CFI of the source program, we can claim that the CFI-hiding scheme is correct.

The first kind of things changed in the shuffling are the absolute addresses of many instructions, because we move chunks of instructions around. The shuffling also modifies the relative addresses of instructions for the same reason. As we mentioned several times before, our compiler passes always generate assembly code with label addresses for all CTIs. The absolute addresses or relative addresses are only generated in the assembling stage which is after the code shuffling. With the labels as their targets, all the CTIs will get the correct target addresses in the binary generated by the assembler.

The second kind of things changed are the hot nodes addresses. Although they have been moved to new places, it does not pose a problem. We already discussed it in Section 3.2.3. If a CTI targets the first block in a hot node $N_i$, which is moved to another place, the control still goes to the place (by jumping to a label or taking the fall through branch) where $N_i$ is located before the shuffling, i.e. the beginning of the code listed in Table 3.1. The first three steps are the instructions which will redirect the control flow to the new address of $N_i$, since we assume
that M-process always returns correct addresses.

Another case of entering a hot node is jumping to some block other than the first one in the node. In this case the control flow will go to the address directly because we didn’t change the label inside the node.

Similarly, there are also two cases to capture the control flow exits from a block, (1) exits from the last block or (2) exits from a block other than the last block in a cell. Let $B_{end}$ be the last block in $N_i$, and let $B_{follow}$ be the block after $B_{end}$. If the control flow leaves from $B_{end}$ by taking the fall through path, Step 7 and 8 will redirect the control flow to $B_{follow}$ after executing several instructions (Step 10 and 11), which restore the registers and clean the stack. This control flow is correct. If the control flow leaves $B_{end}$ but not via the fall through branch, it will jump to the correct address because it is jumping to a label.

The second case is that of the control flow exit from $N_i$ from a block other than $B_{end}$. The control flow in this case is also correct because, again, the label addresses are generated later.

Since we have exhausted all the cases, we showed that our CFI-hiding scheme is correct.

### 3.4 Resilience of the CFI-hiding Scheme

This section discusses the resilience the CFI-hiding scheme. Several metrics and measuring methods described in Section 2.4 are used.

We first prove that the reachability problem of our obfuscator is a PSPACE-Complete problem. Section 3.4.1 introduces the Linear Bounded Deterministic Truing Machine and the definition of PSPACE-complete problems. Section 3.4.2 formulates the reachability problem in our obfuscator and Section 3.4.3 proves that the reachability problem of our obfuscator is in PSPACE-Complete by reducing the acceptance problem in a Linear Bounded Turing Machine to it. Finally, DTOM and SCOM of our obfuscator is calculated in Section 3.4.4.
3.4.1 Acceptance Problem in Linear Bounded Turing Machine

A Linear Bounded Deterministic Turing Machine (LBDTM) (see [9]) $M$ is a conventional Deterministic Turing Machine (DTM) (see [11]) with the restriction that no computation uses more than $n + 1$ tape cells. There are a left marker $\leftarrow$ and right marker $\rightarrow$ at the two sides of the input string. The head of the machine is not allowed to move to the left of $\leftarrow$ or right of $\rightarrow$. LBDTM's formal definition is given below:

**Definition 8 (Linear Bounded Deterministic Turing Machine)** A Linear Bounded Deterministic Turing Machine is a system

$$M = \langle Q, \Sigma, \Gamma, \leftarrow, \rightarrow, \delta, B, q_0, q_{accept}, q_{reject} \rangle.$$ 

where

- $Q$ is the finite set of states,
- $\Gamma$ is the finite tape alphabet,
- $B$ is the blank marker,
- $\leftarrow$ and $\rightarrow$ are the end markers,
- $\Sigma$ is the input alphabet, $\Sigma \subseteq \Gamma - \{B, \leftarrow, \rightarrow\},$
- $q_0$ is the initial state,
- $q_{accept}$ is the accepting state,
- $q_{reject}$ is the rejecting state and
- $\delta$ is the transition function

$$\delta : (Q - \{q_{accept}, q_{reject}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

where no transition of form $(q, \leftarrow) \rightarrow (q', d, L)$ or $(q, \rightarrow) \rightarrow (q', d, R)$ exists, where $d \in \Gamma$ and $q, q' \in Q - \{q_{accept}, q_{reject}\}.$
Now we give the definition of PSPACE and PSPACE-complete.

A language $L$ is said to have deterministic (nondeterministic) polynomial space complexity if there exists a deterministic (nondeterministic) Turing machine $M$ and a polynomial $p$ such that $M$ accepts $L$ and $M$ scans no more than $p(|w|)$ tape cells for any string $w$ in $L$.

PSPACE (NPSPACE) is the set of languages having deterministic (nondeterministic) polynomial space complexity. As a corollary of Savitch Theorem we know that PSPACE=NPSPACE.

A language $L_1$ is polynomial-time reducible [8], written $L_1 \leq_p L_2$, to a language $L_2$ if there exists a polynomial-time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for all $x \in \{0,1\}^*$, $x \in L_1 \iff f(x) \in L_2$.

**Definition 9 (PSPACE-complete)** A language $L$ is PSPACE-complete if

- $L$ is in PSPACE and
- for every language $L' \in PSPACE$, $L' \leq_p L$.

The PSPACE-complete language which is reduced in the proof of the resilience of our obfuscator is Linear Bounded Automaton Acceptance (LBTM-ACC) [9]:

$LBTM$-$ACC = \{< x, M > : x \text{ is accepted by } M, \text{ where } M \text{ is a } LBTM \text{ and } x \text{ is an input to } M \}$.

In [14], Karp proved that LBTM-ACC is in PSPACE-complete.

### 3.4.2 Reachability Problem for the M-process

The essential functionality of M-process is to receive the address request and return the (deterministic) address to the P-process, so it is a deterministic automaton with an output function. The input of this automaton is the ID of a hot node and the output is the address of the hot node. Because the total number of the blocks in the software we want to obfuscate is limited, the M-process is a Deterministic Finite Automaton defined formally as the follows.

**Definition 10 (M-process Deterministic Finite Automaton)** $M$-process is a DFA $A$ which is a 6-tuple: $< S, \Lambda, \Psi, \varsigma, s_0, F, \tau >$, where
• $S$ is a finite set of states,

• $\Lambda$ is the input alphabet the DFA operates on,

• $\Psi$ is the output alphabet of the DFA,

• $\varsigma$ is the transition function, $\varsigma : S \times \Lambda \rightarrow S$,

• $s_0$ is the initial state,

• $F$ is a set of final states and

• $\tau$ is the output function, $\tau : S \rightarrow \Psi$.

Given two states $s$ and $s'$, we say that $s'$ is reachable from $s$ if there exists a sequence of transition of M-process DFA such that, before the transition, the state of the DFA is $s$ and, after it, the state is $s'$. Following is the formal definition of it.

**Definition 11 (MPROCESS-REACH)** A state $s'$ is reachable from another state $s$ iff there exists a sequences of input $\mu = w_0, w_1, \ldots$ such that

$$\varsigma^*(s, \mu) = s'$$

where

$$\varsigma^*(s, w\sigma) = \varsigma(\varsigma^*(s, w), \sigma)$$
and $\varsigma(s, \epsilon) = s'$. $s$ is call the start state and $s'$ is called the end start.

### 3.4.3 MPROCESS-REACH is in PSPACE-complete

First we prove that M-process is in PSPACE (NPSPACE), then we reduce LBDTM-ACC to MPROCESS-REACH.

**Theorem 3.4.1** $\text{MPROCESS-REACH} \in \text{NPSPACE}$

**Proof:** Suppose that the internal state $S$ of M-process is encoded into binary format with length of $n$. The total number of internal states is $2^n$. Given any pair of states $s_i$ and $s_j$, $s_j$
Figure 3.2 Non-Deterministic Dichotomic Search

is reachable from $s_i$ iff there exists a sequence of input $\mu$ of length less than or equal to $2^n$ such that $\zeta^*(s_i, \mu) = s_j$. Consider the standard non-deterministic dichotomic search algorithm shown in Figure 3.2. We choose an arbitrary state $s_k$ and verify that whether $s_k$ is reachable from $s_i$ with an input sequence of $\mu_{ik}, |\mu_{ik}| \leq 2^{n-1}$, and $s_j$ is reachable from $s_k$ with an input sequence of $\mu_{kj}, |\mu_{kj}| \leq 2^{n-1}$. We continue to guess the middle point between each pair of states, say $<s_i, s_k>$ and $<s_k, s_j>$, recursively. For example, we guess another arbitrary state $s_l$ and verify that whether $s_l$ is reachable from $s_i$ with an input sequence of $\mu_{il}, |\mu_{il}| \leq 2^{n-2}$, and $s_k$ is reachable from $s_l$ with an input sequence of $\mu_{lk}, |\mu_{lk}| \leq 2^{n-2}$. In the memory, there are always at most $n$ states, each of size $n$. Hence the maximum memory cell usage is $n \times n = n^2$. Because $n$ is less than the size of M-process, $\text{MPROCESS-REACH} \in \text{NPSPACE}$. 

Theorem 3.4.2 $\text{LBDTM-ACC} \leq_p \text{MPROCESS-REACH}$

The proof is very similar to the one used in [24] because the dispatcher's functionality of CFI-hiding is exactly the same as that of the M-process in our scheme. Here we only provide
a brief overview of the proof. Please refer to Section 6 of [24] for details.

The basic idea is to simulate the execution of a LBDTM on input $w$ with an autonomous M-process Deterministic Finite Automaton. A dispatcher is autonomous if its transition function $\zeta$ is independent of the inputs. That is, $\zeta(s, \mu_1) = \zeta(s, \mu_2)$, where $s$ is any state of the automaton, and $\mu_1$ and $\mu_2$ are a pair of inputs.

Given an input $w$, we specify a M-process Deterministic Finite Automaton $D_{w,M}$ simulating the execution of a LBDTM $M$ on $w$. Every configuration of $M$ is encoded into a state of $D$. $D$’s transition function is specified by three groups of formulae. The first group simulates the rewriting action of $M$, the second group simulates the transition action of $M$, and the third group simulates the moving of $M$’s head. Considering the start states $s$ and ending state $s'$ in Definition 11, $s$ is mapped from the initial configuration of $M$, and $s'$ from any configuration with an accepting state. Thus the LBDTM $M$ accepts $w$ iff some ending state $s'$ in $D$ is reachable from the start state $s$.

### 3.4.4 DTOM and SCOM of the Self-modifying Scheme

As in the previous section, we assume that M-process will always return correct addresses. We first calculate the DTOM of the obfuscator. Consider the binary of the source program with normal static layout. For the sake of simplicity, assume that each control flow path from the program root node (entry node) to a termination (leaf node) has $k$ branches. This would imply that there are $2^k$ possible instantiated paths. Hence DTOM of this program is $\log(2^k)$ which equals $k$. In other words, the targets of $k$ branches would have to be resolved by the adversary to reverse engineer this program. Now consider a program whose basic blocks have been permuted in the static layout. Each of the branches now can have any of the branch instructions (corresponding basic block) as a target. If there are $k' > k$ unique basic blocks in the program, from the adversary’s perspective, static observation only yields the information that each branch can have any of the $k'$ basic blocks as its target. Hence the number of statically observable instantiated control flow paths is $k'^k$ (each of the $k$ branches has $k'$ possible
outcomes). Hence the DTOM of the permuted static layout is \( \log (k'^k) \) which equals \( k \log k' \) (it goes up by the factor \( \log k' \)). For a complete tree like CFG topology, \( k' \) will be proportional to \( 2^k \), and hence DTOM goes up to \( k^2 \). On the other extreme, \( k' \approx k \), and hence DTOM goes up by a factor at least \( \log k \). Not all of the \( k \) branches along a path from the root node to a leaf node need to be obfuscated (permuted). In our implemented approach only a subset of these branches is permuted (say every 10th branch for instance). Let \( l \leq k \) branches be the permuted type. Then the number of statically observable paths is given by \( k'^l \times 2^{k-l} \) which leads to a DTOM of \( l \times \log k' + (k - l) = l \times (\log k' - 1) + k \).

These analyses assume uniformity in the CFG. The DTOM metric could be computed for each individual program CFG. The CFG can have probabilities associated with each control flow edge from program profiling. We generated such profiling driven probabilities for hot block selection in obfuscation. These probabilities over all leaf nodes add up to 1 (the program has to terminate at one of these leaf nodes). Hence this probability \( p \) can be viewed as a random process. Its entropy \( H(p) \) \((H(p) = \Sigma(p_i \times \log(1/p_i)))\) provides a lower bound on the number of binary decisions required to differentiate between these paths. This is a more program specific measure of DTOM.

As for SCOM, let there be \( k' \) unique basic blocks in the program once again. Then for each of the \( k \) branches (basic blocks) along the path to be exposed (deobfuscated), the number of potential taken successors to be searched \( T \) is \( k' \). Hence SCOM of the traditional static layout program is given as \( k \times k' \). For the statically permuted program (our obfuscation technique), both the taken and not taken successors of a basic block are not known. Let \( T \) be the number of taken successors and \( NT \) be the number of not taken successors (on average for each of the \( k \) branches). Without any other information source, both \( T \) and \( NT \) are \( k' \). Hence the number of successors (or size of the search space) for each basic block along the path is \( \binom{k}{2} \) which is approximately \( k'^2 \). Hence SCOM for the permuted static layout is \( k \times k'^2 \), which is higher by a factor \( k' \). For a program where \( l \leq k \) branches are permuted, SCOM is a hybrid value given by \( l \times k'^2 + (k - l) \times k' \).
CHAPTER 4. PROTECT THE MONITOR-PROCESS BY A SELF-MODIFYING SCHEME

In the first half of this thesis we described a way to extract the CFI out of P-process and move it to M-process. The next part of the task is to protect the M-process.

To obfuscate a program, many research projects assume that there is a secure component [25] which is free from observation. The secure component may reside in a smart card or in a secured server. Although a secure component makes the obfuscation quite strong, in the real world it is annoying and bothersome for the users to deploy extra hardware to run software. Furthermore, with network connection incur much overhead. In this chapter, we describe a self-modifying scheme, in which the image of the obfuscated program in memory keeps modifying itself. This protection approach requires no extra hardware support but still offers good obfuscation. Hence it is quite practical.

Although the M-process is the only program that is protected in our discussion, the protection scheme is not limited to the M-process. In fact, it works fine in theory for any program. In this thesis, our approach is adapted to get high performance from exploring the specific attributes of the M-process. Hence we only talk about M-process protection.

Section 4.1 introduces the basic structure of the self-modifying scheme. Then Section 4.2 discusses the protection of Jump Table, which is specific to the M-process. A group of keys with certain properties are used during the encryption. Section 4.3 at first proves that the keys exist by construction, and then discuss a little more about the key generation. Detailed implementation is in Section 4.4. Section 4.5 proves the correctness of the self-modifying scheme and Section 4.6 discusses the resilience of it.
4.1 Basic Structure

In order to protect the M-process, we try to expose the unencrypted code as little as possible. First let us define some terms used in our description.

The binary executable of the obfuscated program (obfuscated M-process in this case) is divided into several pieces, each of which contains one or more blocks. Each piece of code is called a Cell. During the execution of M-process, the control flow sometimes exits from one cell and enters another. We call this a Control Flow Switch. The block from where the control flow switch takes place is call a Source Block. The block executed after the control flow switch is called a Target Block. The cell containing the source block is called a Source Cell and the one containing the target block is called a Target Cell. The block from where M-process starts execution is called an Entry Block and it is in the Entry Cell. The block from where M-process may exit is called an Exit Block while the cell containing it is called an Exit Cell. Note that there is only one entry block and one entry cell, but there may be more than one exit block and exit cell.

There are keys embedded at the end of each source block. If there is only one branch at the end of a source block B, only one key is embedded in B. If there are two branches in B, B contains two keys. Each key corresponds to a control flow switch. All the keys have an important property that after XOR'ing a proper key with the whole M-process’ image, the source cell is encrypted, the target cell is decrypted and all the others are changed but still remain encrypted.

When M-process starts, one or more blocks in the entry cell C₀ are executed until the first control flow switch s occurs. Let B_{new} be the target block, C_{new} be the target cell of s and the key corresponds to s be k₀,new. Before B_{new} is executed (in fact B_{new} is not able to be executed at this moment because it is encrypted), k₀,new is used to XOR the whole image. The XOR operation will close C₀, open C_{new}, and change all the other cells. Then the control flow goes to B_{new} and executes one or more blocks until another control flow switch occurs. This process continues until an exit block is executed and the M-process terminates. In the whole
execution, there is always only one cell that is observable in plain text.

![An Execution of the Protected Program, with the Control Flow of Start -> Cell 0 -> Cell j -> Cell k -> ...... -> Cell i -> End](image)

Figure 4.1 An Example Execution of the Protected M-process

Figure 4.1 illustrates an example of the execution of the protected M-process which contains n cells. At the beginning of the execution, only cell $C_0$ is exposed as plain text (machine code). After the execution of some blocks in $C_0$, a control flow switch $s_1$ whose source cell is $C_j$ occurs. Then the key corresponding to $s_1$ is used to XOR the whole image. It closes $C_0$, opens $C_j$, and changes all the other cells. After the encryption and decryption, the target block of $s_1$ is executed. At this moment, $C_j$ becomes the only cell that is observable. When the second control flow switch $s_2$ happens, $C_k$ is the target cell. The key corresponding to $s_2$ is used to close $C_i$, open $C_k$ and modify all the others. Then the target block in $C_k$ is executed. This
continues until an exit block in $C_i$ is reached.

4.2 Protection of Jump Table

As mentioned in Section 3, the CFI of the P-process is extracted and saved in a table named Jump Table which is only accessible to the M-process. Each entry in Jump Table is a $<\text{NodeID, Address}>$ tuple whose meaning is self-explaining.

To encrypt Jump Table, the first concern is where it should be located. Because the binary code of the M-process is protected by the self-modifying scheme, the best place for holding Jump Table is inside the code of the M-process. That is, all entries in Jump Table are hard-coded as local variables of the M-process. This approach, however, has drawbacks because Jump Table becomes part of the code for the M-process and it incurs much running time overhead. Because the size of Jump Table is proportional to the size of the source program $P$ (recall the definition of shuffling rate) and the size of the M-process without Jump Table is a constant, the size of Jump Table dominates the size of the M-process with Jump Table. When $P$ is huge, M-process becomes very large and the time taken to modify the whole image of M-process becomes very long. Another difficulty of hard-coding Jump Table into the M-process is that it is hard to distribute the entries of Jump Table into different cells of the M-process. For the reasons above, we leave Jump Table in an array which is separated from the code for the M-process. The array is encrypted and the keys\footnote{Note that the keys mentioned here are different from the keys used to open and close the cells.} are hard-coded in the code of the M-process. With this scheme, Jump Table is still protected and the size of the M-process does not becomes larger when $P$'s size increases.

The second concern is how many keys should be used for encrypting Jump Table. To get higher resilience, a large number of keys are preferred. These keys could be distributed in the cells so that the protected program need to be run for many times to uncover all of them. In an extreme situation where each cell contains at least one key, the adversary needs to open all the cells to get the whole Jump Table. The process to observe all the cells will be proved resilient...
in Section 4.6. However, efficiency suffers with this scheme. Because each entry in Jump Table is short, the size of a key could not be much smaller than that of an entry, which means that the size of the M-process will still become larger when $P$'s size increases. The best way to solve this problem is to use one key to encrypt a group of entries in Jump Table. Both good resilience and acceptable overhead can be achieved with this method. Because of lack of time, our current implementation only uses one key to encrypt Jump Table. The disadvantage of this simplified implementation is a weak protection of Jump Table, but it also has advantages. With a constant number of keys, the size of the M-process is a constant, thus the efficiency of our obfuscator increases as $P$ becomes larger. In our future work, we will increase the number of the keys used to protect Jump Table.

### 4.3 Encryption Key Assignment

The keys are crucial for our self-modifying scheme. With the set of keys embedded in the obfuscated M-program, the M-program is able to keep modifying itself during its execution and there is only one cell that is observable at any point of time. The important property the set of keys should satisfy is defined as the open-close property.

**Definition 12 (Open-close Property)** Given a program $P$ which is divided into a group of cells and an encryption method $E$, let $S$ be the set of all the control flow switches in $P$. A function $f$ on $S$ is said to have open-close property for $P$ with encryption method $E$ if the following requirements hold:

(i) There is a way to initialize $P$ so that only the entry cell is in plain machine code.

(ii) After the initialization of $P$, $P$ begins to execute. Every time a control flow switch $s$ occurs, apply $E$ on the whole image of $P$ with the key $k = f(s)$. The target cell of $s$ is decrypted, the source cell of $s$ is encrypted and all the other cell are modified but not decrypted.

Now let us prove the following theorem.
Theorem 4.3.1 Given any program $P$ and a method to portion it into a group of cells (let $S$ be the set of all the control flow switches in $P$ with the portion), there exists at least one function $f$ on $S$ that satisfies the open-close property with encryption method XOR.

Proof: We prove this theorem by construction. First let us define the following notations, where $s$ is a control flow switch. These notations are also used in the rest of the thesis.

- $C^0_k$: initial state of $C_k$
- $C^s_k$: state of $C_k$ after $s$ takes place.
- $sb(s)$: ID for the source block of $s$
- $sc(s)$: ID for the source cell of $s$
- $tb(s)$: ID for the target block of $s$
- $tc(s)$: ID for the target cell of $s$

First we build a Control Flow Graph whose nodes are the cells and the directions of the edges are ignored. We call this graph a Control Flow Graph in terms of Cell (CFGC). For each edge in the CFGC, a key is assigned. For any $s$, the value of $f(s)$ is decided by $C_{sc(s)}$ and $C_{tc(s)}$. That is, for any two control flow switches $s_1$ and $s_2$, if their source blocks are in the same cell and their target blocks are also in the same cell, the keys assigned to $s_1$ and $s_2$ are the same. To generate the key for the edges in the CFGC, we add edges to it first to get a complete graph and assign keys for all the edges in the graph. Let $k_{i,j}$ denotes the key assigned to edge between $C_i$ and $C_j$ in the CFGC.

Suppose there are $n$ cells $C_0, C_1, \ldots, C_{n-1}$ in the CFGC of M-process, we generate $n - 1$ keys randomly for $k_{0,1}, k_{0,2}, \ldots, k_{0,n-1}$. All the other keys $k_{ij}$ are generated by

$$k_{ij} = k_i \oplus k_j \quad (4.1)$$
Function $f$ is defined by
\[
 f(s) = k_{sc(s),tc(s)}
\]
(4.2)
, and the initialization of $P$ is defined as follows: entry cell $C_0$ is not changed; any cell $C_i$ other than $C_0$ is obfuscated by $C_i \oplus k_{0,i}$.

Now we are ready to show that $f$ satisfies the two requirements listed in the definition of open-close property. It is obvious that the initialization mentioned above satisfies requirement (i). The main task is to prove requirement (ii). Given any control flow switch $s$ which occurs during the execution of the M-process at time $t$ (right before $s$ occurs), let $p = C_{p_1} C_{p_2} \ldots C_{p_n}$ denote the sequence of cells on the control flow path by which the program reached $s$ from the beginning. Note that $C_{p_1} = C_0$, $C_{p_n} = C_{sc(s)}$ and there may be duplicates of some cells, i.e., $C_{p_i} = C_{p_j}$ for some $i, j$.

At time $t$, every cell has been XOR’ed with a series of keys which correspond to all the control flow switches on path $p$. The value of XOR’ing all these keys is
\[
 K_s = k_{p_1,p_2} \oplus k_{p_2,p_3} \oplus \ldots \oplus k_{p_{n-1},p_n}
 = k_{0,p_2} \oplus k_{p_2,p_3} \oplus \ldots \oplus k_{p_{n-1},sc(s)}
 = k_{0,p_2} \oplus (k_{0,p_2} \oplus k_{0,p_3}) \oplus \ldots \oplus (k_{0,p_{n-1}} \oplus k_{0,sc(s)})
 = (k_{0,p_2} \oplus k_{0,p_2}) \oplus (k_{0,p_3} \oplus k_{0,p_3}) \oplus \ldots \oplus (k_{0,p_{n-1}} \oplus k_{0,p_{n-1}}) \oplus k_{0,sc(s)}
 = 0 \oplus 0 \oplus \ldots \oplus 0 \oplus k_{0,sc(s)}
 = k_{0,sc(s)}
\]

After applying key $f(s) = k_{sc(s),tc(s)}$ on the whole image of $P$, the cells are modified as follows:

1. The target cell of $s$
\[
 C_{tc(s)}^s = C_{tc(s)}^0 \oplus K_s \oplus k_{sc(s),tc(s)}
 = (C_{tc(s)} \oplus k_{0,tc(s)}) \oplus k_{0,sc(s)} \oplus k_{sc(s),tc(s)}
 = C_{tc(s)} \oplus k_{0,tc(s)} \oplus k_{0,sc(s)} \oplus k_{0,sc(s)} \oplus k_{0,tc(s)}
 = C_{tc(s)}
\]
which means that $C_{tc(s)}$ is decrypted.

2. The source cell of $s$ is modified to

$$C^s_{sc(s)} = C^0_{sc(s)} \oplus K^s \oplus k_{sc(s),tc(s)}$$

$$= (C_{sc(s)} \oplus k_{0,sc(s)}) \oplus k_{0,sc(s)} \oplus k_{sc(s),tc(s)}$$

$$= C_{sc(s)} \oplus k_{sc(s),tc(s)}$$

which means $C_{sc(s)}$ is encrypted.

3. For any cell $C_i$ other than $C_{tc(s)}$ and $C_{sc(s)}$,

$$C^s_i = C^0_i \oplus K^s \oplus k_{sc(s),tc(s)}$$

$$= (C_i \oplus k_{0,i}) \oplus k_{0,sc(s)} \oplus k_{0,sc(s)} \oplus k_{0,tc(s)}$$

$$= C_i \oplus k_{0,i} \oplus k_{0,tc(s)}$$

$$= C_i \oplus k_{i,tc(s)}$$

$k_{i,tc(s)} \neq 0$ because $i \neq tc(s)$. Thus $C^s_i \neq C_i$, which means that $C_i$ is changed but still kept encrypted.

Having considered the effect of application of $f(s)$ on the whole image, we know that $f$ satisfies all the three requirement in (4.1) of the open-close property, which completes the proof.

Figure 4.2 shows an example of the key assignment process. The graph on the left is the original directed CFGC. It consists of four cells and five edges. After adding edges and ignoring the direction of all the edges we get the complete graph on the right. We are to generate six keys: $k_{0j}$, $k_{0l}$, $k_{0m}$, $k_{jl}$, $k_{jm}$ and $k_{lm}$.

$k_{0j}$, $k_{0l}$, $k_{0m}$ are generated randomly. The other three are calculated by Equation (4.1). Note that the graph is undirected so there is only one key $k_{0m}$ for the pair $<C_0,C_m>$ even though there are two edges between them in the original CFGC.
The initialization of the cells are

\[ C_i^0 = \begin{cases} C_i & \text{if } i = 0 \\ C_i \oplus k_{0,i} & \text{otherwise} \end{cases} \]

### 4.4 Implementation Steps

In our approach, \( \text{mPrcs} \) is the function that fulfills M-process' functionality. From now on we will use \( \text{mPrcs} \) in our discussion.

To enable \( \text{mPrcs} \) to modify itself (its image in the memory) during its execution, we move its code from the text section to an array (called the code array) in the data section. This is because the text section in the memory has several constraints. With the code in the data section, we are much freer to modify the code and extend its functionality.

There is also an important helper function named \( \text{xorAll} \). When a control flow switch \( s \) occurs during the execution of \( \text{mPrcs} \), \( \text{mPrcs} \) calls \( \text{xorAll} \). \( \text{xorAll} \) is separated from \( \text{mPrcs} \) and is an ordinary function residing in the text section. \( \text{xorAll} \) XOR's the whole code array, closes \( sc(s) \), opens \( tc(s) \), and redirects the control flow to the beginning of \( tb(s) \).

Very similar to the implementation of the CFI-hiding scheme discussed in Section 3.2, Machine-SUIF [23] passes are used to generate CFG IR for \( \text{mPrcs} \). With the annotated assembly code, we can get the profiling information of the blocks and divide them into cells. Then the cells
are moved to the code array with the initialization method described in the proof of Theorem 4.3.1. Keys are also hard-coded into the cells. Because the data section is not used for program text, several modifications need to be done to enable mPrCs to run in the code array.

4.4.1 Decide the Cells

Before we divide mPrCs into a group of cells, we need to get the information about the basic blocks of M-process. Several kinds of information is required for the blocks. Sizes are needed to decide the cells; the labels at the beginning are used to decide the CFGC; the last instruction’s type and its target (if the last instruction is a CTI) are also indispensable for the analysis of the relationship between the cells.

Relationship between the blocks is very important for program efficiency. A cell contains several blocks but not all of call xorAll – only the source blocks of the control flow switches do. We add instructions at the end of these switches for calling xorAll. Similarly, we add instructions to clean the stack and restore the registers only at the beginning of the target blocks of the control flow switches. In order to speed up the obfuscated program, we try to avoid control flow switches, since each control flow switch runs xorAll which modifies the whole code array. Fewer control flow switches also hide the code better, because a new cell is open only when the control flow enters it.

There are two properties about the cells that need to be decided, the total number and their sizes. Since there is only one cell open at any point of time, approximately \((n-1)/n\) of the code for xorAll is encrypted, where \(n\) is the total number of cells. The higher the number of the cells, the more we protect M-process from observation. But as the number of cells increases, function xorAll is called more frequently which is pretty expensive in terms of time.

In order to decrease the number of executions of xorAll, we should carefully design the boundaries of the cells. A cell should be chosen so that most blocks in a cell only jump inside the cell to avoid control flow switches.

Because mPrCs is small (it contains only twenty nine basic blocks), we analyze its CFI
manually to make the decision about the cells. If the same self-modifying scheme is used to protect large applications, profiling tools could be used. The advantage of manual handling is better performance.

Another concern in cells determination is that the total size of all the blocks in each cell should be approximately the same. If it is not the case, much memory will be wasted by the padding at the end of the cells.

With all these concerns in mind and according to the mPrcs's CFG (see Figure 4.3), we
divide \texttt{mPrCs} into six cells, each of which contains three to six basic blocks. They are listed in Table 4.1. Note that \( C_0 \) and \( C_{30} \) are two dummy blocks which are not included in any cell.

<table>
<thead>
<tr>
<th>Cell ID</th>
<th>Block ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 - 3</td>
</tr>
<tr>
<td>1</td>
<td>4 - 9</td>
</tr>
<tr>
<td>2</td>
<td>10 - 14</td>
</tr>
<tr>
<td>3</td>
<td>15 - 18</td>
</tr>
<tr>
<td>4</td>
<td>19 - 23</td>
</tr>
<tr>
<td>5</td>
<td>24 - 29</td>
</tr>
</tbody>
</table>

4.4.2 Key Generation

The key generation method we used is the one used in the constructive proof for Theorem 4.3.1. Because there are six cells, we generate six random keys \( k_{0,1}, k_{0,2}, \ldots, k_{0,5} \). All the other keys are generated by Equation (4.1). For example, \( k_{3,5} = k_{0,3} \oplus k_{0,5} \). Each key corresponds to a group of control flow switches. The keys are inserted at the end of the source block of the corresponding control flow switch.

4.4.3 Function \texttt{xorAll}

The XOR operation and control redirection job in the self-modifying scheme is fulfilled by a function named \texttt{xorAll}. \texttt{xorAll} is subroutine of \texttt{mPrCs} but it is separated from \texttt{mPrCs} and does not reside in the code array. When a control flow switch \( s \) occurs, \texttt{mPrCs} calls \texttt{xorAll} with the key corresponding to \( s \). \texttt{xorAll} XOR’s all the cells to close the source cell and to open the target cell. Then \texttt{xorAll} redirects the control flow to the target block of \( s \). The code in \texttt{mPrCs} continues to execute until the next control flow switch which again calls \texttt{xorAll}.

Another scheme is to put the code for XOR’ing and redirection in \texttt{mPrCs} directly at the end of the source blocks of the control flow switches. The difficulty of this approach is that the XOR operates on the \textit{whole} code array which includes the code for XOR’ing and redirection. Assume \( B \) is the \( B_{sb(s)} \) and ends with a CTI. After the XOR’ing, \textit{instr} is also encrypted so it
could not be executed to redirect the control flow. Although there could be a way to distinguish
the CTI instruction from others during the XOR’ing such that it could be always kept open,
using a separate xorAll function makes more sense. Another advantage of using xorAll is that
it makes future extension easier.

There are two arguments that xorAll needs to help in the self-modifying scheme. The
first argument needed is the key $k(s)$ which is used to XOR the whole code array. The second
argument required is the offset of $B_{tb(s)}$ in $C_{tc(s)}$. We use $off(s)$ to denote the offset. After
XOR’ing the code array with $k(s)$, the whole $C_{tc(s)}$ is opened. $off(s)$ is used to calculate the
start address of $B_{tb(s)}$.

The first part of xorAll’s task is simple. It just XOR’s the whole code array with $k_s$. Note
that the size of a cell should be divisible by the size of the keys. For example, the key argument
in our current implementation is four bytes long, so the size of the cell should be divisible by
four. xorAll uses the key to XOR each four-byte-long word in code array.

The second part of xorAll’s job, however, is a little trickier. After the XOR operation, it
redirects the control flow to $B_{tb(s)}$. Each cell starts with a magic number so that xorAll can
distinguish the newly opened cell ($C_{tc(s)}$). Then xorAll adds the address of $C_{tc(s)}$ and offset
$off(s)$ to get the address of $tc(b)$. Finally, xorAll modifies the return address in its stack frame
and returns so that the control flow goes to the beginning of $B_{tb(s)}$. xorAll does not return to
the end of $B_{sb(s)}$ followed by a jump instruction because $B_{sb(s)}$ is already encrypted by $k(s)$.

In-line assembly in C is used to generate two instructions at the end of the assembly code
for xorAll:

\[
\begin{align*}
\text{movl} & \quad -4(\%ebp), \%eax \quad \# \text{code for redirection} \\
\text{movl} & \quad \%eax, 4(\%ebp) \quad \# \text{code for redirection} \\
\text{leave} & \\
\text{ret} &
\end{align*}
\]

In the code above, $-4(\%ebp)$ contains the address of the beginning of the $B_{tb(s)}$. $4(\%ebp)$
contains the return address for xorAll in the stack. After the modification of the return address, instructions leave and ret redirect the control flow to the beginning of \( B_{tb(s)} \).

But we are not done yet. We still need to guarantee a healthy stack before and after calling xorAll. Before a function call, all the live registers should be saved; after the call, they should be restored. In an ordinary function call, the stack cleaning and register restoring instructions follow the call instruction. However, after the function call to xorAll, the control flow will return to the beginning of \( B_{tb(s)} \), not the end of the call instruction at the end of \( B_{sb(s)} \). Thus the code to clean the arguments on the stack and restore the registers should be moved to the beginning of \( B_{tb(s)} \). Also notice that \( B_{tb(s)} \) may be a target block of a set \( S \) of control flow switches and \( B_{tb(s)} \) is only one of them. When the control flow goes to the beginning of \( B_{tb(s)} \), \( m\text{PrCs} \) cannot know which control flow switch in \( S \) is the one that just occurred. Because of this, the registers set \( R \) saved for each control flow switch in \( S \) should be the same. Each control flow \( s \) requires save and restore of a set of registers. Let \( r(s) \) denote these registers. For good performance, we should analyze all the switches in \( s \) to get the union of all the sets \( r(s) \). For the sake of simplicity, we save and restore all the general purpose registers and the flag for every call to xorAll.

### 4.4.4 Modification for Each Basic Block

After cells determination, the next step is to modify some of the basic blocks to enable them to call xorAll. Code may be added at the beginning and at the end of a block.

First let us consider the modification at the beginning of a basic block \( B_i \). There are three cases:

1. If \( B_i \) is the entry block in \( m\text{PrCs} \), no modification is needed.

2. If \( B_i \) is the target block of a control flow switch \( s \), i.e., \( i = sb(s) \), code is inserted at the beginning of \( B_i \) as follows.

\[
\text{lbl_clnStk}_i:
\]
clean_the_stack
restore_the_registers

lbl_thru_i:
    B_i
    ....

3. Else the code is modified as follows.

    lbl_thru_i:
    B_i
    ...

In the pseudo-code above, `lbl_clnStk_i` is the label at the beginning of the code which cleans the stack and preserves \( r(s) \). If the control flow enters \( B_i \) from other cells, the return address of `xorAll` targets `lbl_clnStk_i`. `lbl_thru` is the label marking the beginning of the original code for \( B_i \). It is needed because \( B_i \) may be target of some CTI within the same cell. These in-cell CTIs should go to \( B_i \) directly without executing the code for stack cleaning and register restoring.

Another thing that needs to be mentioned is that the first block of each cell is special. If \( B_i \) is at the beginning of a cell other than Cell 0, the control flow that goes from \( B_{i-1} \) to \( B_i \) must go through `mPrcs` even when the last instruction of \( B_{i-1} \) is not a CTI. That is the way the first block in a cell is categorized into Case 2.

The modification at the end of the blocks is a little more complicated than that at the beginning. For the code at the beginning of \( B_i \), we only consider whether \( B_i \) is target of a control flow switch or the first block in a cell. But for the modification at the end of \( B_i \), we need to consider the last non-directive instruction `instr` of \( B_i \). If `instr` is a CTI, we check its target to see whether it is in the same cell or not. If the target is in another cell, we substitute `instr` by instructions which call `xorAll`. If `instr` is `ret`, nothing needs to be added. If `instr` is a general-purpose instruction other than CTI (add and `push` for example, see [12]), we check to
see whether $B_i$ is the last block in the cell. The algorithm is described here briefly for several cases and sub-cases. Suppose $B_i$ is in cell $C_i$. The code modification at the end of $B_i$ is divided into four cases.

1. If $instr$ is ret, then modify nothing

2. If $instr$ is jmp target, where target is in block $B_k$, then there are two sub cases.
   
   i. If $B_k$ is also in $C_i$, then modify nothing
   
   ii. If $B_k$ is in another cell other than $C_i$, then substitute $instr$ with the following code

   ```
   save_registers
   pushl k_ik
   pushl $0x7000000
   call xorAll
   ```

3. If $instr$ is jcc target (Jump if Condition Is Met such as jc, jne), where target is in block $B_k$, there are four sub cases.
   
   i. If $B_k$ is also in cell $C_i$ and $B_i$ is the last block in $C_i$, then after $instr$ append code to save the registers and make call to xorAll, with $k_{i(i+1)}$ as the key argument.
   
   ii. If $B_k$ is also in cell $C_i$ and $B_i$ is not the last block in $C_i$, then after $instr$ append an instruction jmp lb1_thru_(i+1).
   
   iii. If $B_k$ is not in cell $C_i$ and $B_i$ is the last block in the cell, then $instr$ is substituted by the following instructions

   ```
   jcc lb1_svRgs_i
   save_registers
   pushl k_i(i+1)
   pushl $0x7000000
   call xorAll
   ```
lbl_svRgs_i:
  save_registers
  pushl  k_ik
  pushl  $0x7000000
  call  xorAll

iv. If $B_k$ is in a different cell and $B_i$ is not the last block in the cell, then substitute $instr$
with the following code:

  jcc  lbl_svRgs_i
  jmp  lbl_thru_(i+1)

lbl_svRgs_i:
  save_registers
  pushl  k_ik
  pushl  $0x7000000
  call  xorAll

4. If $instr$ is other general-purpose instructions except for CTI, then there are two sub cases.

i. If $B_i$ is the last block in $C_i$, then append the following code after $instr$

  save_registers
  pushl  k_i(i+1)
  pushl  $0x7000000
  call  xorAll

ii. If $B_i$ is not the last block in $C_i$, then after $instr$, append instruction

  jmp  l_thru_i(i+1)

In this algorithm, save_registers stands for the instructions to save the registers. There
are two pushl instructions before call xorAll. The first argument is the key. The immediate
number 0x7000000 should be the offset of the target block in the target cell. Without generating
the binary code, it is very hard for us to get the real offset of a block in the cell where the block is located. So we just push an arbitrary integer here as a placeholder. The placeholder will be substituted by the real offset when we copy the binary code to the code array.

4.4.5 Copy Binary Code to the Code Array

After the modification of the blocks, we can generate the binary code for mPrCs and copy it to the code array. The structure of a cell is shown in Figure 4.4.

```
<table>
<thead>
<tr>
<th>Magic Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block</td>
</tr>
<tr>
<td>Block</td>
</tr>
<tr>
<td>push key</td>
</tr>
<tr>
<td>push offset</td>
</tr>
<tr>
<td>call xorAll</td>
</tr>
<tr>
<td>pop</td>
</tr>
<tr>
<td>Block</td>
</tr>
<tr>
<td>push key</td>
</tr>
<tr>
<td>push offset</td>
</tr>
<tr>
<td>call xorAll</td>
</tr>
<tr>
<td>pop</td>
</tr>
<tr>
<td>Block</td>
</tr>
<tr>
<td>nop</td>
</tr>
<tr>
<td>nop</td>
</tr>
<tr>
<td>nop</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>nop</td>
</tr>
</tbody>
</table>

Figure 4.4 the Structure of a Cell

First we generate assembly code with annotation for each basic block with help of Machine-SUIF. Similar to the implementation of the CFI-hiding scheme, it is important to make sure that all the CTIs in the assembly code will only jump to a label. This is a must because when
we copy the code into the array, the relative addresses and absolute addresses of the instructions will probably be changed. If a CTI’s target is an absolute or relative address instead of a label, it could not jump to the correct target after the code is copied into the code array.

Secondly, we should get the relationship between the binary code blocks and the assembly code blocks. One assembly block may map to several sequential binary blocks because we have added labels and CTI’s in the basic block modification step. The executable format in our implementation is ELF and we could make use of the BFD library to read the binary code. However, we don’t need much information (and the objdump tool of GNU Binutils generates all information we need). We parse the output of objdump and get all information such as the start line index, end line index, start line address, end line address, etc. We can calculate the offset of any block in the cell it belongs to with this information.

The last part is copying the binary code into the code array in a template file to get the obfuscated xorAll. For each cell $C_i$, we insert a magic number at its beginning. Then all the binary code blocks in this cell are copied to the code array. During this step the placeholder number 0x70000000 at the end of $B_{sb(s)}$ is substituted by the real offset of $B_{tb(s)}$ in $C_{tb(s)}$ with correct byte order (little-endian or big-endian). Finally, several nop instructions are appended after the last binary block in this cell. For the sake of simplicity, we use nop for padding. In fact, what instruction is used for padding doesn’t matter for the correct result, since they will never be executed. In future work, we will put some fake blocks in the padding area to enhance obfuscation.

There is still one more difficulty in moving the binary code into the code array. We need to be careful of the global variables and functions used in mPrs. If mPrs uses them directly, they may not be found at run-time. The reason for this is same as the one why we should use labels instead of absolute or relative addresses in mPrs: the linker and loader won’t relocate the symbols, function names or global variables in a data section. If mPrs is in text section, information of the labels, function names and the global variables are in the relocation table. Linker and loader will find the correct addresses for them when the final executable is generated.
However, mPrcs is in data section so that it is considered as simple data and no relocation work is performed for it.

Our workaround for this problem is writing mPrcs as a function which is called by a main function. main resides in the text section. Before main calls mPrcs, it initializes two arrays: one contains the pointers to all the functions used by mPrcs; another contains pointers to all the global variables used in mPrcs. When main calls mPrcs, it passes the two arrays as arguments. mPrcs accesses all these functions and the global variables via the pointers in the arguments\(^2\). When the linker and loader generate the executable, they will relocate addresses for all the pointers in the two arrays, since they are located in the text section. These pointers are then passed to mPrcs and correctly used there.

### 4.5 Correctness of the Self-modifying Scheme

To establish the correctness of the self-modifying scheme used to protect M-process, we need to prove two things. First, the key assignment algorithm satisfies the open-close property. Second, the CFI of mPrcs and the data information should be preserved after it is partitioned into cells and moved to the code array. The former is stated as Theorem 4.3.1 and has been proved in Section 4.4.2.

Because we did not modify the global data, it is not changed. The local variables in the code array for mPrcs are not changed either because the code being executed in mPrcs is always decrypted to its original state.

As for the control flow information, if the control flow moves inside one cell without executing a CTI, it is obvious that the control flow is preserved because nothing is changed in our algorithm; if the control flow moves inside the same cell by executing a CTI, the control flow is preserved by the `lbL.thru_` and `jmp lbL.through` instructions. If the control flow moves between different cells, right before the instruction in the new cell is executed, xorAll is called. xorAll opens the new cell (closing the old cell at the same time) and redirects the control

\(^2\)In fact, the format strings of the `printf` statement should also be passed as arguments in the pointer arrays
flow to the next instruction to be executed. Before and after calling \texttt{xorAll}, live registers are preserved and restored respectively so that the stack is clean. This means that the control flow is also preserved, which completes the proof that the self-modifying scheme is correct.

4.6 Resilience of the Self-modifying Scheme

First we claim that the self-modifying scheme protects M-process from static attack. In the binary for the obfuscated M-process, only the entry cell is observable. The keys are embedded in the code. Only by executing the M-process can the adversary observes the other cells. The CFI of M-process is also hidden from static attack. Very similar to Section 3.4, we can prove that the reachability problem of our obfuscator is a PSPACE-Complete problem by reducing LBTM-ACC to it. Because the only property used in the proof of Theorem 3.4.1 is CFI hiding, there is no difference between the proof for the self-modifying scheme and the CFI-hiding scheme. Thus we omit the proof.

The strong protection we got from the self-modifying scheme is that it prevents the M-process from observation in a dynamic attack. Although researchers have developed several kinds of obfuscators to protect a source program from static attacks ([24][19][5]), there is not much progress in preventing dynamic attacks. Our self-modifying scheme, though, works well in preventing dynamic attacks.

Compare the unprotected M-process and the obfuscated program in the view of an attacker who deploys a dynamic attack. For a non-obfuscated program, we can view it as being obfuscated by a trivial obfuscator \( O_1 \) which doesn’t modify the source program at all, i.e., \( O_1(P) = P \). The BOOM of \( O_1 \) is \( 1/n \), because once it is loaded into the memory, the whole image is observable.

Let \( O_m \) be our self-modifying obfuscator. The obfuscated program is \( P_m = O_m(P) \). Let \( c \) be the number of cells in \( P_m \). Consider the binary execution tree of the CFGC of \( P' \). There are about \( c/2 \) leaf cells. Hence the BOOM of \( O_m \) is \( c/2n = O(c/n) \). If we choose \( c = O(n) \), then the BOOM of \( O_m \) is \( O(1) \), which is much higher than that of \( O_1 \).
CHAPTER 5. EFFICIENCY OF THE OBFUSCATOR

Functionality, efficiency and resilience are the most important properties of an obfuscator. The correctness and resilience of our obfuscator have been evaluated in Chapter 3 and 4, respectively. This chapter evaluates its efficiency.

We used empirical methods to evaluate the efficiency. The two metrics defined in Definition 2, space cost $C_s(O, P)$ and time cost $C_t(O, P)$, are calculated for four different applications. When choosing applications, we favor CPU-bound applications instead of I/O-bound ones. Our hot node selection scheme is not related to the percentage of I/O instructions in all the instructions, so the more the I/O instructions, the lower the time cost. To show the extra time taken by our obfuscator, we choose the “worst cases”. The applications are:

- tsort: Topological sort. This version is from GNU Core-utils.
- compress42: Compress data utility.
- test: A program written by us.
- bunzip021: An early version of bunzip utility.

For each application, we consider its space cost in terms of the assembly code and the binary code. We also calculate the cost with or without considering mPrCs.

Table 5.1 records the space cost in terms of assembly code. $P$ is the source program, $C_s^A$ is the space cost without considering the size of mPrCs and $C_s^{\ast A}$ is the space cost with mPrCs. ‘A’ stands for “in terms of assembly code”.
Table 5.1  Space Cost Efficiency in Terms of Assembly Code

<table>
<thead>
<tr>
<th>P</th>
<th>Size of Source Program in Assembly</th>
<th>Size of Protected Program in Assembly</th>
<th>Size of mPrCs in Assembly</th>
<th>(C_s^A)</th>
<th>(C_s^{A'})</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsort</td>
<td>36k</td>
<td>41k</td>
<td>16k</td>
<td>0.139</td>
<td>0.583</td>
</tr>
<tr>
<td>compress42</td>
<td>118k</td>
<td>131k</td>
<td>16k</td>
<td>0.110</td>
<td>0.246</td>
</tr>
<tr>
<td>test</td>
<td>182k</td>
<td>207k</td>
<td>16k</td>
<td>0.137</td>
<td>0.225</td>
</tr>
<tr>
<td>bunzip021</td>
<td>210k</td>
<td>226k</td>
<td>16k</td>
<td>0.076</td>
<td>0.152</td>
</tr>
</tbody>
</table>

Table 5.2 records the space cost in terms of binary code. \(P\) is the source program, \(C_s^B\) is the space cost without considering the size of mPrCs and \(C_s'^B\) is the space cost with mPrCs. ‘B’ stands for “in terms of binary code”.

Table 5.2  Space Cost Efficiency in Terms of Binary Code

<table>
<thead>
<tr>
<th>P</th>
<th>Size of Source Program in Binary</th>
<th>Size of Protected Program in Binary</th>
<th>Size of mPrCs in Binary</th>
<th>(C_s^B)</th>
<th>(C_s'^B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tsort</td>
<td>28k</td>
<td>31k</td>
<td>33k</td>
<td>0.107</td>
<td>1.286</td>
</tr>
<tr>
<td>compress42</td>
<td>52k</td>
<td>55k</td>
<td>33k</td>
<td>0.058</td>
<td>0.692</td>
</tr>
<tr>
<td>test</td>
<td>62k</td>
<td>71k</td>
<td>33k</td>
<td>0.145</td>
<td>0.677</td>
</tr>
<tr>
<td>bunzip021</td>
<td>72k</td>
<td>74k</td>
<td>33k</td>
<td>0.028</td>
<td>0.486</td>
</tr>
</tbody>
</table>

From the two groups of data we get two conclusions:

- Both the space cost in terms of assembly code and binary code is around 1. That is, \(C_s(O, P) = O(1)\). This means that the extra space need of the obfuscated program compared with the source program is about 100%.

- As the size of \(P\) increases, the space cost with mPrCs decreases. This is because the size of mPrCs is fixed, and the size of P-process is proportional to the source program (when the shuffling rate is fixed).

To formulate the conclusion, let \(s_m\), \(s_p\) and \(s_{O(p)}\) denote the size of mPrCs, the source
program and P-process, respectively. The space cost with mPrCs is

\[ C_s(O, P) = \frac{s_0(p) + s_m}{s_p} - 1 \]

\[ = \frac{s_0(p)}{s_p} + \frac{s_m}{s_p} - 1 \]

\[ = C + \frac{s_m}{s_p} - 1 \]  

(5.1)

where \( C \) is a constant. When \( s_p \) is much larger than \( s_m \), \( C_s(O, P) \) becomes \( C - 1 \), which means that the space cost is very low.

The valuation of the time cost is shown in Table 5.3. \( C_t \) is only calculated for the real run time. For each application, two groups of execution time are calculated. The first group is for P-process working together with a non-obfuscated M-process (we coded a function to simulate mPrCs without the self-modifying scheme). The second is for P-process working with the obfuscated M-process, which is the combination of both parts of our obfuscator. The purpose of separating two groups of time costs is to separate the overhead incurred by the CFI-hiding scheme and the self-modifying scheme.

<table>
<thead>
<tr>
<th>P</th>
<th>Running Time of Not Protected Programs</th>
<th>Running Time of PP With an un-protected MP</th>
<th>Running Time of PP With Protected MP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>real</td>
<td>user</td>
<td>sys</td>
</tr>
<tr>
<td>tsort</td>
<td>4.90</td>
<td>0.29</td>
<td>0.05</td>
</tr>
<tr>
<td>compress42</td>
<td>2.05</td>
<td>0.42</td>
<td>0.52</td>
</tr>
<tr>
<td>test</td>
<td>34.33</td>
<td>2.45</td>
<td>1.40</td>
</tr>
<tr>
<td>bunzip021</td>
<td>6.46</td>
<td>0.62</td>
<td>0.54</td>
</tr>
</tbody>
</table>

From the data collected, the time cost is less than a factor of 10. Because the shuffling rate we choose is pretty high (20%), the user could choose a small shuffling rate to achieve a faster obfuscated program. When the shuffling rate is set to 5%, the time costs for the four applications are listed in Table 5.4. Take tsort as an example. The time cost drops 83.2% without obfuscating M-process and 61% with obfuscated M-process. With the small shuffling rate, all the time costs are less than a factor of 3, which is acceptable for most programs.
Table 5.4 Time Cost Efficiency (shuffling rate: 5%)

<table>
<thead>
<tr>
<th>P</th>
<th>Running Time of Not Protected Programs</th>
<th>Running Time of PP With a unprotected MP</th>
<th>Running Time of PP With Protected MP</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>user</td>
<td>sys</td>
<td></td>
</tr>
<tr>
<td>tsort</td>
<td>4.90 0.29 0.05</td>
<td>5.73 0.65 0.29</td>
<td>10.20 0.66 0.31 1.08</td>
</tr>
<tr>
<td>compress42</td>
<td>2.05 0.42 0.52</td>
<td>3.58 0.52 0.75</td>
<td>7.933 1.46 1.55 2.87</td>
</tr>
<tr>
<td>test</td>
<td>34.33 2.45 1.40</td>
<td>42.57 4.06 2.36</td>
<td>98.53 4.40 4.56 1.87</td>
</tr>
<tr>
<td>bunzip021</td>
<td>6.46 0.62 0.54</td>
<td>11.18 1.44 1.89</td>
<td>20.16 2.20 2.33 2.12</td>
</tr>
</tbody>
</table>

Table 5.4 also demonstrates the relationship between the overhead of the self-modifying scheme and the CFI-hiding scheme. The former is a little higher than the latter. This is because the overwriting of the whole M-process is pretty expensive. It is even more expensive than IPC, the main overhead in the CFI-hiding scheme.
CHAPTER 6. SUMMARY AND FUTURE WORK

6.1 Summary

The technology to protect software from malicious reverse engineering is called software obfuscation. A software obfuscator is a program $O$ that takes the source program $P$ as input and produces an obfuscated program $O(P)$. An obfuscator should be correct ($O(P)$ has same functionality as $P$), resilient ($O(P)$ tolerates attacks) and effective ($O(P)$ is not too much slower than $P$).

In our work, we have proposed and implemented an obfuscator consisting of two parts. The first part of the obfuscator, a CFI-hiding scheme, which extracts the CFI from the source program and hides it in another process called the M-process. The source program, with crucial CFI extracted, becomes a P-process. The two processes coordinate to fulfill the functionality of the original program during the execution. By hiding the CFI we protect the program from malicious reverse engineering. We proved that the obfuscator is correct by showing that the obfuscator does not change any data flow or control flow of the source program. The resilience of the approach in evaluated in two ways. Firstly, to statically understand the obfuscated program, the adversary should be able to solve the reachability problem in the obfuscated program. By showing $\text{LBDMT-ACC} \leq_p \text{MPROCESS-REACH}$, we proved that the CFI-hiding approach is effective against static attacks. Secondly, we define two metrics DTOM and SCOM about static obfuscation and calculated them for the CFI-hiding scheme.

The second part of our work proposed a self-modifying scheme to protect M-process, which is in charge of the important CFI from the P-process. By only opening a small part of the binary code in memory, the scheme prevents the adversary from observing the binary code at
once. We proved the correctness of the scheme also by showing that both the source program's data flow and control flow are preserved in the obfuscation. We proposed a metric BOOM to evaluate the resilience of the obfuscation against dynamic attack. If a program $P$ is not obfuscated at all, we can assume that it has been obfuscated by a trivial obfuscator which just copies $P$. Thus the BOOM value of an un-protected program is $O(1)$. The self-modifying scheme, however, has a BOOM value of $O(c)$, where $c$ is the total number of cell in $P$. If we let $c = \Theta(n)$, where $n$ is the total number of $P$'s basic blocks, the BOOM value becomes $O(n)$, which is improved a lot. The ability to protect a program from dynamic problem makes this scheme promising.

The final result of our obfuscator is the combination of the two schemes, so it protects a program from both static and dynamic attacks without relying on a secure components in hardware. The efficiency of the obfuscator is evaluated by the time cost and space cost metrics. After experimenting with four applications, we showed that the space cost is pretty low and the time cost is acceptable, if we don’t hide too much CFI in the source program.

### 6.2 Future Work

We propose several possible directions for further research in software obfuscation.

- Develop a standard set of metrics for obfuscators. Until now there was no wide accepted standards.

- Formally model dynamic attacks.

For possible enhancement on our two schemes, we propose the following.

- Shuffling all the cells during the execution. Because xorAll figures out the target cell of a control flow switch by checking the magic number at the begin of each cell, it does not matter where the cells are.
• Generate faked instructions in the padding part of each cell. Because the areas will never be reached during the execution, what is in there does not matter. Probably opaque constructs [5] are good code chunks to be inserted.

• Only modify part of the code in M-process, while preserving the open-close property of the keys. Currently the whole M-process is modified every time a control flow switch occurs, so it is pretty expensive. Only modifying part of the code will improve the efficiency.
BIBLIOGRAPHY


ACKNOWLEDGMENTS

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and the writing of this thesis. First and foremost, I am indebted to the two major professors in my graduate career: Dr. Soma Chaudhuri and Dr. Akhilesh Tyagi. I am grateful to them for their guidance, patience and support throughout this research and the writing of this thesis. They believe in me and guided me through the most difficult times throughout this work. I thank Dr. Chaudhuri for her superb patience in putting up with me.

I would also like to thank my committee member, Dr. Thomas Daniels, for his valuable comments and timely flexibility. I would additionally like to thank Allada Veerendra and Dr. Glenn Holloway. Allada implemented the encryption of Jump Table and Glenn helped me a lot with Machine-SUIF. Without them, this project would not have been completed as smoothly and successfully as it has been. At last, I thank my friends Balaji, Brian, Ge and Chang for their help and suggestions for this thesis.