NANYANG TECHNOLOGICAL UNIVERSITY

Software Code Protection through Software Obfuscation

Software Code Protection through Software Obfuscation

LIANG SHAN

LIANG SHAN

SCHOOL OF COMPUTER ENGINEERING

2010

2010

Software Code Protection through Software Obfuscation

LIANG SHAN

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Master of Engineering

2010

Abstract

The mode of platform-independent software running on client machine has various advantages. But it also introduces some problems, such as how to guard against software piracy, how to protect user from intellectual property theft and from attacks by malicious modifications. Development of reverse engineering techniques makes all these attacks easier. Therefore, obfuscation has gained a lot of interest since the obfuscation technique makes the cost of program reverse engineering prohibitively high with least program size and execution speed overheads.

In this report, we present our proposed obfuscation algorithms which are primarily based on self-modifying code. We have designed three software obfuscation algorithms. The first self-modifying proposal is integrated with Control Flow Flattening to realize the obscurity in instruction and execution control flow. The second one is based on Basic Blocks to improve both the performance of storage size overhead and the instruction disassembly errors metrics. And the third one is proposed at the function level to provide full protection to improve the security to both static and dynamic attacks. We implement these obfuscation algorithms at link time and evaluate them on standard benchmark suite. The novel techniques are evaluated with metrics referred by existing proposals to prove that our algorithms succeed in confusing the disassembler when reverse engineering the control flow and instructions of program. We also discuss how the novel methods improve the obfuscation efficiency in comparison with the existing competing obfuscation methods.

Acknowledgments

I wish to express sincere appreciation and deepest gratitude to Assistant Professor Sabu Emmanuel, my supervisor, for his patient and altruistic assistance, inspiration, and encourage throughout this research effort. Without his consistent and illuminating instruction, the research effort could not reach its goal.

Then, I would like to express my heartfelt gratitude to professors and teachers at the CeMNet Laboratory. They have instructed and helped me a lot in the past two years.

Last I would like to say thanks to my family for their considerations and confidence in me all through these years. I also owe my sincere gratitude to my friends and classmates who gave me their help and time in helping me work out my problems through the research work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The purpose of software reverse engineering is to recover the higher-level structure, especially the data flow and control flow graph, from a lower-level program. Reverse engineering technique has advantages in legitimate applications, such as re-construction of source code from binary executable file when the original code is lost or corrupted. But in most cases, the technique is used for illegal purposes. The challenge from reverse engineering has great impact on the development of platform-independent software, especially on the X86 platform. The impact includes, but not limited to, security vulnerabilities analysis, software malicious modification and intellectual property theft. The protection of software has evolved from hardware security, such as coprocessor[65] and Trusted Computing[26], to software security[39][40], such as encryption and obfuscation[6].

In this chapter, we discuss the motivation for the study of obfuscation techniques at first. Since encryption algorithm also provides protection for software, we discuss the advantage of obfuscation over encryption on program protection. Then, we discuss why we choose link time obfuscation, rather than source code level obfuscation or compile time obfuscation. The feasibility of dynamic mutation method to realize obfuscation at run time and the hardware support to dynamic obfuscation will also be discussed in this chapter.

1.1 Motivation for This Research

Tampering, piracy and reverse engineering constitute considerable threat to software security. Tampering[52] of software refers to the intended modification of software against the wishes of owner/distributer of software. The research in tamper resistant software technique is often classified into tamper-resistant and tamper proof, or self-certifying tamper resistant techniques[36][81]. There are many methods to obstruct software tampering, including hardware and software approaches[7][42][68][81]. Software piracy often refers to unauthorized copying of software. However, reverse engineering[71] is the basis of all issues relating to software security and copyright. Software obfuscation has been an emerging field against reverse engineering.

One example that necessitates protection from obfuscation technique is Digital Rights Management(DRM) system[51]. The architecture of DRM system is shown in figure 1.1. DRM system consists of content server, license server and DRM client, or DRM agent. The goal of DRM technique is to protect the content from unauthorized usage[57]. Since DRM agent contains almost all the security information about DRM system[80], including how the content is encrypted and how the license (key) to open the encrypted content is obtained, stored and used[56], these DRM agents need advanced techniques to hinder sensitive code analysis and to secure the DRM system[52][69]. Code obfuscation is an effective technique to protect agent codes from compromising sensitive DRM system information[51].

In most existing DRM technologies, the DRM agent, or DRM client, contains the content protection mechanisms applied to the system. However, the DRM agent that runs at the consumer side is often exposed to reverse engineering to compromise DRM agent or to obtain the secret license information kept at the DRM agent.

Except for protection of software from reverse engineering, obfuscation algorithm also benefits us on hiding program vulnerabilities[33][66] and key data and protecting pro-

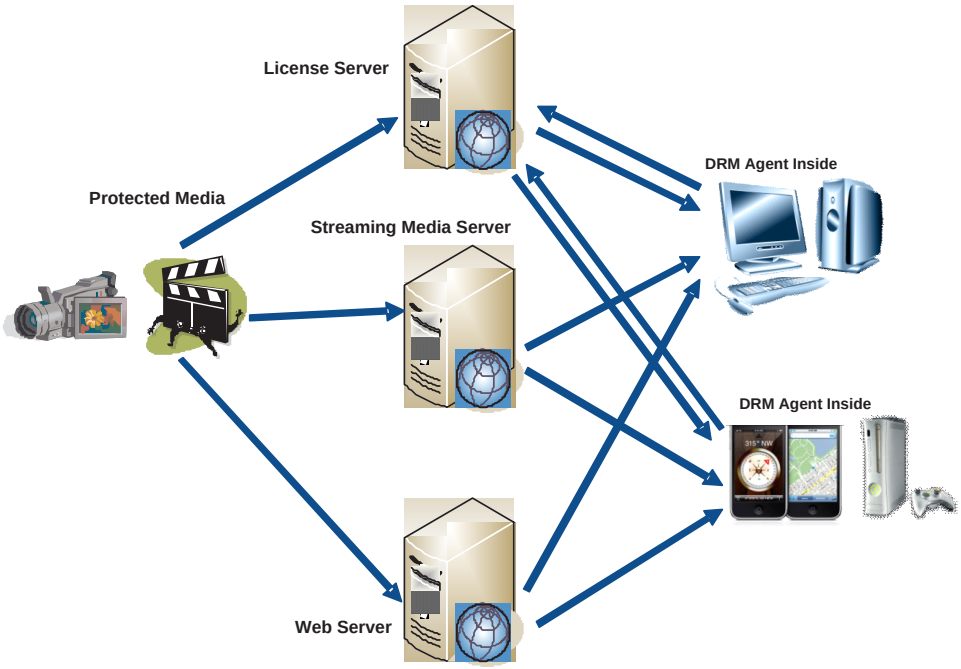

Figure 1.1: Architecture of DRM system

gram from attacking such as code lifting[77]. One motivation for the obfuscation technique research is database security. In both Oracle and Microsoft SQL Server systems, login and application role passwords stored in the system tables are always obfuscated. This prevents non-administrative users from viewing any passwords, including their own. Additionally, application role passwords can be obfuscated when the application role is activated before they are sent over the network.

There has been such existing ways as encryption and obfuscation algorithms to protect sensitive information from being discovered. Usually, encryption algorithm is black-box security based on cryptographic techniques[12]. Encryption is designed to secure critical data, either for short-term transmission or long-term storage[70]. It seems that encryption algorithm provides more security compared to obfuscation given that the key to decryption is not exposed. Thus, the question is, why we need obfuscation as we already have so many choices of encryption algorithms, such as RSA, DES, etc.

One reason is that obfuscation has more general applicability[79]. Encrypted computation is limited to specific problems such as polynomial evaluation. Usually, the requirement of program execution at client side necessitates symmetric encryption algorithm, such as DES, AES, and Blowfish. And the decrypted code is same with the original one exactly. The correct key must be present in order for the original information to be retrieved.

Obfuscation can be applied to any computation that can be expressed as a program. Obfuscation offers more versatility and efficiency. Obfuscation transforms program into a "messy" form that is hard to understand, but performs the same function as the original program.

Also, obfuscated software can be distributed alone without accessories such as decryption algorithm and key. Obfuscated software runs directly on client machine without decryption before execution. But encrypted software must be decrypted with corresponding decryption algorithm and key before execution. Actually, it is hard to control the spread of decryption algorithm and key once they are exposed. Such requirements as decryption not only burdens software owner/distributer, but also weakens the security of encryption.

A solution to protect the decryption algorithm and key is to integrate them together, or self-encryption technique. One approach to self-encrypting technique is proposed by Cappaert[12]. The algorithm embeds the decryption algorithm and key into program. Original code will be decrypted and obtained at execution time. However, the compromise to restrictions, such as loops, recursions and multiple callers, will definitely expose the source code to an attacker. Obfuscation is a lower cost approach to achieve software protection.

One more problem is that, when applied to software protection, encryption necessitates the program or some part of program to be encrypted and decrypted as a complete

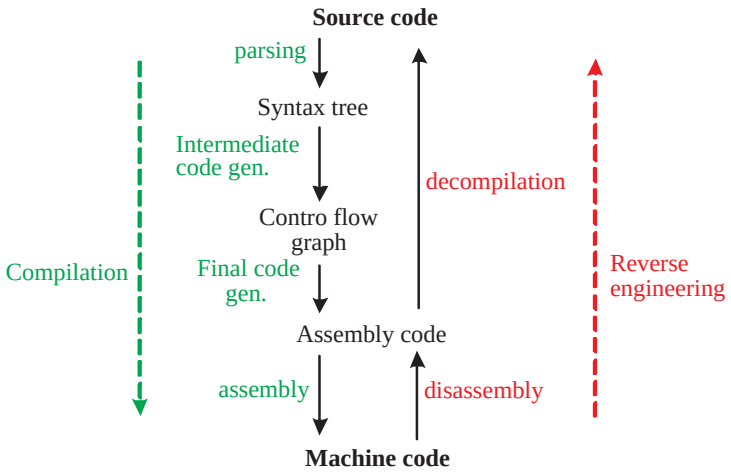

Figure 1.2: The process flow of compilation and reverse engineering

unit without touching internal architecture of program. This requirement limits the application of encryption on program protection. However, because of the requirement of lossless encryption and decryption, encryption only transforms protected data to another form. It does not change the data flow and control flow of protected program like obfuscation does.

Figure 1.2 illustrates the process flow of compilation and reverse engineering. Because of all the above mentioned considerations, it is necessary to design secure obfuscation techniques, rather than to apply the existing encryption algorithms to software protection directly.

It is worth noting that many obfuscators, sometimes partially, use what's called "Encrypted" obfuscation[5][25][42]. This should not be confused with true encryption. They're simply using an encryption algorithm as the mechanism to scramble the code.

1.2 An Overview of Obfuscation

The general obfuscating transformation definition by Collberg et al.[16] is: Let $T : P \rightarrow P'$ be a transformation of a source program P into a target program P'. $T : P \rightarrow P'$

is an obfuscating transformation, if P and P' have the same observable behavior. More precisely, in order for $T : P \rightarrow P'$ to be a legal obfuscating transformation the following condition must hold:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.
- Otherwise, P' must terminate and produce the same output as P.

We have reviewed the proposed algorithms on obfuscation. Most of them apply obfuscation transformation in a static manner. Static obfuscation also includes such techniques as arrays re-construction[22], inheritance relationship modification[67], function and component inline and outline[10], variables re-construction[43], branch function[38] and jump table spoofing[41]. Then what limits the extension to dynamic obfuscation? And what is the best place and time in the code to apply obfuscation? Before we have a deep investigation into various obfuscation techniques, it is necessary to make these preliminary definitions clear.

1.3 Problem Definition

Firstly, we discuss the relationship between the efficacy of obfuscation and the time point when obfuscation takes place.

Different types of obfuscation can be applied depending on the time point when program is obfuscated ranging from source code level through link time. There are source code and intermediate code level obfuscation, such as the byte code obfuscation in Java protection[15][23]. When software is distributed as native code, or binary executable file format[53], link time obfuscation should be used[20][44][48], as shown in figure 1.3. Comparing with link time obfuscation, source code level and compile time obfuscation is easy to implement since it is not necessary to re-construct the control flow graph from

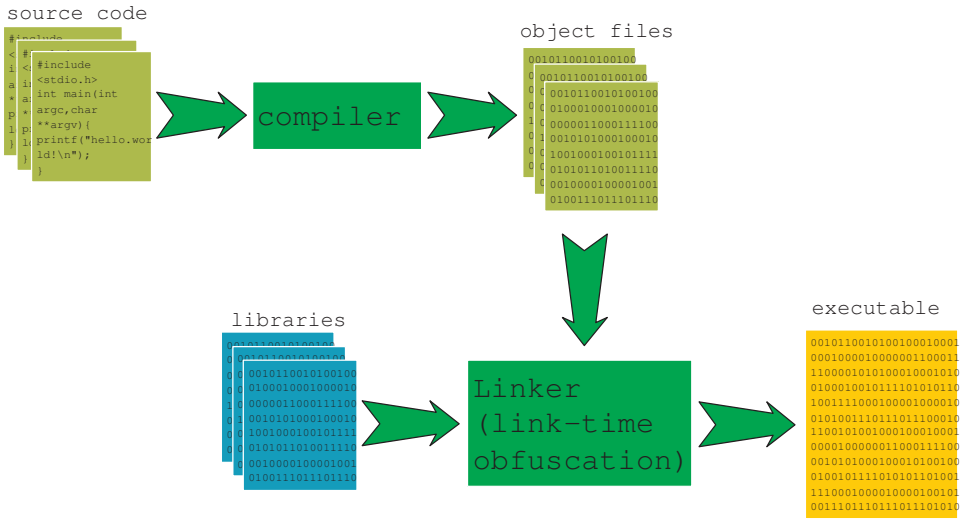

Figure 1.3: Link-time obfuscation of program

machine code. But the advantage becomes a drawback if we take account of program optimization. The efficacy of source code level or compile time obfuscation may be weakened or erased by link time optimizer. An attacker could do the following to remove the obfuscation. Firstly, the attacker disassembles the obfuscated program in binary executable file format to get assembly code as shown in figure 1.2. Then the attacker assembles the assembly code and links the code with optimization algorithm to remove redundant code. The method works effectively to remove the obfuscation code and restore the original code. Link time obfuscation could escape this attack in that it is applied after code optimization and no subsequent step will erase the effect of the transformation.

One more important reason why we prefer link time obfuscation is the lack of a whole program overview and the absence of final address computation at source code level or compile time. The address computation is only available at link time and the address computation is necessary for obfuscation, especially for dynamic obfuscation techniques.

Various dynamic obfuscation techniques are emerging because static obfuscation is losing advantage in competition with software reverse engineering techniques.

Firstly, dynamic obfuscation thwarts the effort of extracting static view by disassembling machine code directly. Static view is the basis of software analysis, especially for a systematic analysis. For a static obfuscation, attacker can disassemble the machine code to extract static view directly. But for dynamically obfuscated program, the static view differs from that during the execution of the program. In order to construct a static view of the dynamically obfuscated program, attacker have to emulate the instructions modifying the code. Therefore, dynamic obfuscation is forcing the attacker to perform program understanding on a debugging trace instead of to perform a static graph structure analysis only. This is also because that the traditional control flow construction algorithms fail in the presence of dynamic mutation code, such as self-modifying code and self-generating code. Almost all tools for program analysis and reverse engineering cannot deal with dynamically mutating code either. For example, many analyses, such as constant propagation or liveness analysis require a conservative control flow graph of the program. Dynamic obfuscation technique has such an impact on software dynamic emulation that the time to launch emulation is increased to being prohibitively long. Emulation of a particular program point requires a breakpoint at the program point. In practice, a break placed on an address blocks the execution. The break instruction may be either overwritten and never be executed or triggered frequently without having the right instruction in place. Because of these difficulties, there is no disassembly techniques that has been proposed until now for dynamic mutation based obfuscated programs.

Then we discuss hardware support consideration to dynamic obfuscation technique. One important reason is that modern operating systems enforce “WX policy” to protect system from security vulnerabilities analysis[59]. The policy means a memory page is either writable (data) or executable (code), but not the both. But self-modifying code requires memory pages to be writable and executable at the same time. The hardware support limitation can be bypassed by a syscall “sys_mprotect”, which allows a program

to change the flags for nearly every page. It seems that this call is not necessary to make a page in the section .bss executable because on x86 a readable page is also a executable page, and the section .bss is read/write. But this behavior may be changed by the appearance of the NX-flag inside modern Central Processing Unit(CPU). NX-flag, or Non Executable flag, is intended to mark various areas of the computer's memory so flagged contains *non-executable data*, or not code. Processing will halt if program counter points at such an area that is so flagged. So the "sys_mprotect" is mandatory to make self-modifying code feasible.

Thus as part of this research we focus on the dynamic mutation based obfuscation of codes at link time.

1.4 Our Contributions

The thesis presents proposals for sensitive information protection generally for program software based on self-modifying code. We designed three algorithms, namely control flow obfuscation based on Control Flow Flattening, obfuscation based on basic blocks and obfuscation at function level. We implemented and evaluated them with evaluation metrics in a standard way which is used in existing proposals[41][54].

1.4.1 Self-Modifying Code Obfuscation Based on Control Flow Flattening

The first proposed algorithm is based on self-modifying code and Control Flow Flattening. The proposal makes full use of the preceding and succeeding in Control Flow Flattening as well as junk bytes concepts to improve the reverse engineering difficulty. The algorithm protects program by introducing difficulties in re-constructing both basic block execution sequence and control flow. Although Control Flow Flattening does not introduce disassembly errors, the algorithm efficiently resists static analysis in that it

successfully camouflages the original control flow by re-directing the program control to fake address during disassembling. At obfuscation time, normal instruction in flattened basic blocks will be obfuscated to control flow instructions or vice versa. Modifying and restoring instructions are inserted to preceding and succeeding blocks also at the obfuscation time. A set of rules is presented and discussed to guarantee that the behavior of the program is same as that of original program.

Comparing with other self modifying code based technique[37], it is not necessary for the technique to calculate the program control flow precisely and to consider the influence of loop control flow. Also, the technique will not introduce any new modules to the original program since the new modules may expose the intention of obfuscation. These advantages improve the algorithm's resistance to disassembly technique in that higher potency, resilience and stealth are achieved with lower space and time costs. With experimental results, we show that the technique is effective in confusing the disassembler on control flow analysis.

1.4.2 Self-Modifying Code Obfuscation Based on Basic Blocks

Then, we continue the work and propose to obfuscate software based on basic blocks without Control Flow Flattening. The algorithm is applied without Control Flow Flattening, but it does not loss the property of the randomness in block execution sequence. It introduces more randomness in splitting the basic block at random locations and thus increase the difficulties in re-constructing basic block and in identifying edge connections.

In evaluation metrics, such as instruction disassembly errors and control flow disassembly errors, the algorithm based on Control Flow Flattening achieves better performance comparing with existing proposals. But the storage size overhead is 3 times that of the original size. The algorithm based on basic blocks solves the storage size overhead problem and it also bring us better performance in control flow disassembly error metric.

The algorithm first splits a selected basic block into multiple blocks, creates the control flow connecting the split blocks and obfuscates these edges. By employing this algorithm and one to many modification, the proposal can help to protect the control flow and thus the sensitive information without having the large storage size overhead. The experimental result shows that the performance of the algorithm is better than signal-based obfuscation[54] in all metrics, especially on protecting the edges and instructions.

1.4.3 Self-Modifying Code Obfuscation At the Function Level

We then proposed this algorithm to provide protection to control flow between functions. We choose to apply the obfuscation algorithm to caller function since the obfuscation in callee function will be weakened by the symbol table. We propose to obfuscate function call instruction to normal instruction for more security, since the abnormal stack analysis in a static disassembly code will expose the protection schedule. The algorithm integrates with the algorithm based on basic block in chapter 4 to protect program control flow from being reverse engineered. We analyzed that the algorithm is resistant to both static and dynamic attacks. The total performance of the algorithm and basic block algorithm is evaluated together. It brings in a little loss of performance. But in most cases, the algorithm brings us the benefits that disassembler will fail to produce a report since it is confused by the obfuscated control flow.

1.5 Organization of The Thesis

The remainder of this report is organized as follows: chapter 2 presents the background theory of obfuscation technique and has a literature review of related work. Chapter 3 proposes a control flow obfuscation based on self-modifying code and Control Flow Flattening. In chapter 4, we propose to mutate the program based on Basic Blocks. We also propose a function level protection based on self-modifying code in chapter 5. Chapter 6 concludes this thesis and presents views on future works.

Chapter 2

Literature Survey and Background Theory

In this chapter, we review the existing obfuscation techniques and discuss reverse engineering techniques in section 2.1. Then, we discuss the obfuscation evaluation metrics in section 2.2. In section 2.3, the platform for our implementation and evaluation is presented.

2.1 Literature Survey

In this section, a literature survey of existing proposals and an investigation on reverse engineering techniques are presented.

2.1.1 Control Flow Obfuscation

Obfuscation is designed to protect program from reverse engineering. There are code obfuscation techniques based on control flow obfuscation[25][54][74] and self modifying codes[5][37][46]. Most of the proposed obfuscation techniques focus on control flow obfuscation. Control flow obfuscation is to conceal the control flow information of program to confuse disassembler on control flow re-construction. The techniques such as opaque predicates[17][49], double-process obfuscation[25] and signal based obfuscation[54] belong to control flow obfuscation.

One technique of control flow obfuscation is opaque predicates[17][49]. Opaque predicates was proposed by Collberg et al. in [17]. It is often constructed with a complex Boolean expression, pointer or indirect control flow, whose value of true or false is hard to deduce for an automatic de-obfuscator. The number of predicates increases complexity. But if after adding opaque predicates, the code differs significantly from the original one, it will be easy to be detected. The opaque predicate could be used to various scenarios, such as to be inserted before dead or irrelevant code, to extend the loop condition.

Majumdar explored the application of opaque predicates in distributed systems[49]. Different with local system, distributed system have the feature of system communication. With this advancement, the paper comes out with system's state machine. The state of every system represents as a number on a doubly circular linked-list. When the numbers representing the state add up to a const, the predicate will be true. The state machine will be stimulated by send and receive obfuscation-specific message. Because it is hard to predict whether message delivery happens or what is the message delivery order, and so those predicates inserted to original code could not be precisely predicted, the adversary will have little chance to do static analysis. However, the problem is the coverage of the method in program. The method is hard to be applied all over the software as other techniques because the time delay in every communication between two distributed system accumulate to an unacceptable level if too much of the technique is applied.

Another technique is to implement two-process scheme to realize control flow based obfuscation[25]. The idea of the algorithm is to spawn a monitor process, M-process, to communicate with the original program process, P-process, with Inter-Process Communication. The communication happens when P-process queries an address from M-process at each obfuscation point. M-process stores the requested address info of P-process in a table named *Jump Table*. M-process is stored in data section, rather than text section, to be self-modified. Actually, the idea of these two papers[25][49] is similar in that the sys-

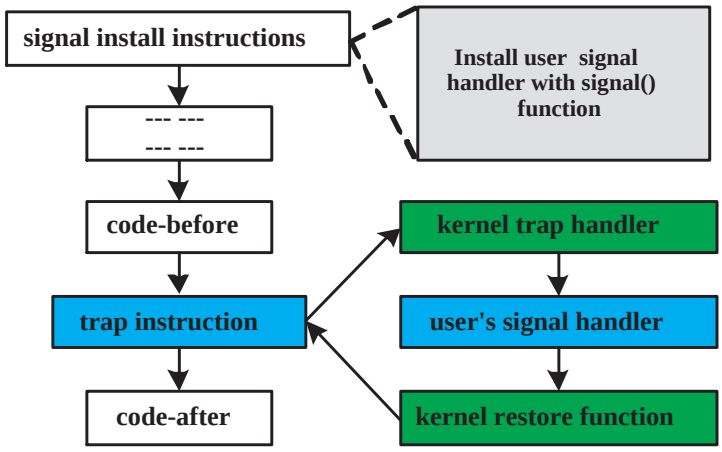

Figure 2.1: Signal in Operating System

tem state in them relies on Inter-Process Communication, locally or remotely. However, the focus is to protect M-process from various attacks.

Popov et al. introduces the idea of *signal* based obfuscation technique[54]. Signal is an asynchronous notification mechanism between processes in operating system[2]. The internal mechanism of signal handling is shown in figure 2.1.

Signal handler is installed at the beginning of program with function *signal()*. When program runs to a trap instruction, a trap will be raised and the kernel trap handler will take over the program control flow to the installed user signal handler. After user signal handler finished execution, program runs to kernel restore function. This function restores the machine state when trap was raised and then program control flow returns back to the original program execution.

The obfuscation technique takes advantage of signal's feature that the trap instruction transfers program control flow. It transfers control flow instructions, such as jump, call and return instruction, to trap instructions.

Figure 2.2 shows a sample of applying signal-based obfuscation on call instruction. Setup code sets a flag to enable the signal handler to identify obfuscation signal from system signal. At obfuscation time, the call instruction in original program is modified

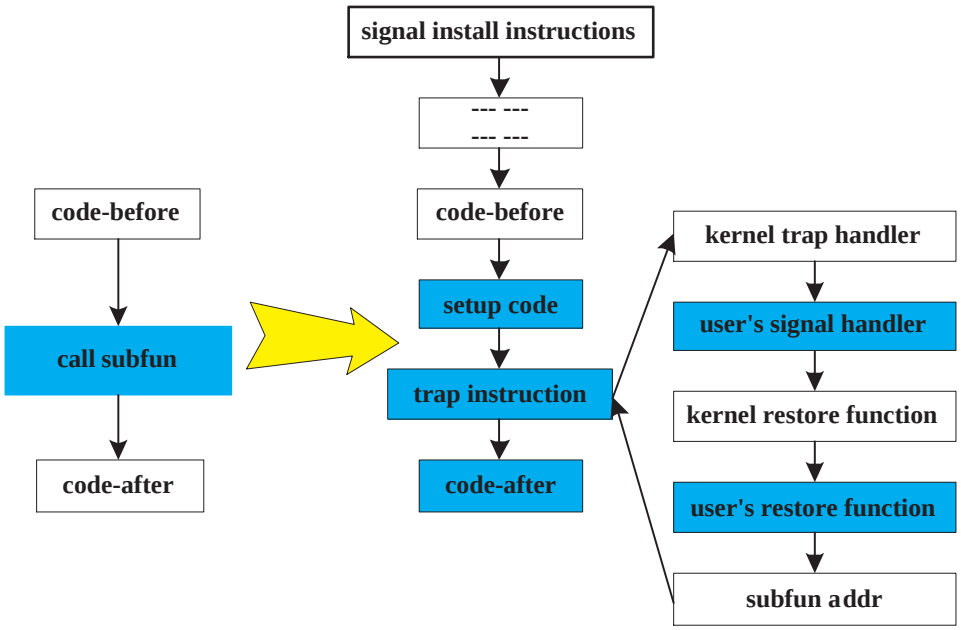

Figure 2.2: A sample of signal based obfuscation

to a trap instruction. Setup code is inserted before trap instruction. Signal handler is also installed at the beginning of program. The subfunction address is saved in table and the table is indexed by the hash value of the trap instruction address. At execution time, "user's signal handler" overwrites the kernel restore function's return address with user's restore function's address. Then, after execution of "user's signal handler" and "kernel restore function", the program runs to "user's restore function". User's restore function calculates the hash value with the original subfunction address and searches address table. Then, "user's restore function" pushes the subfunction address into stack and transfers program control to subfunction address by a return instruction. Finally, the subfunction returns and program control flow comes back to the context of trap instruction.

2.1.2 Liveness Analysis

The liveness analysis is studied for the reason that there are limited registers and multiple temporary variables in real program. We decide the liveness of a variable at a particular point by checking whether it is assigned one more time within the path from the point to where the variable is used in the source of an assignment or other statements, such as function call, return statement, etc.

In self-modifying code proposal, at the place where the modifying code and restoring code is inserted, we need to check the liveness of those variables affected for that the behavior of obfuscated program is same exactly with the original one's. The liveness of these variables needs to be protected by methods such as push them to top of stack and pop them up before and after the modifying code is executed.

2.1.3 Reverse Engineering Techniques

Before we make clear whether a protection mechanism is actually robust, we should know which attacks it can protect against, or what type of reverse engineering obfuscation must confront. Reverse engineering techniques can be classified into two types - static disassembly analysis and dynamic analysis. Dynamic analysis is to construct the program control flow graph by dynamic information obtained by observing the program execution. Static analysis is the process to build a model of state of the program and to determine how the program reacts to this state, including disassembly or de-compilation[73]. There are mainly two types of static disassembly technique available for the purpose of reverse engineering, namely linear sweep and recursive traversal[45][61].

Linear sweep, which is the way *objdump* works, analyzes byte by byte from the first execution byte of text section. The algorithm of linear sweep is shown in figure 2.3. It will keep bytes that do not belong to instruction as data embedded in code section until

```
DisasmLinear:
    startAddr = address of the first executable byte;
    endAddr = startAddr + text section size;
    addr = startAddr;

    while (startAddr <= addr <= endAddr)
    {
       Instr = decode instruction at address addr;
       addr += length( Instr );
    }
```

Figure 2.3: Linear sweep algorithm

it meets the next meaningful byte. Because it can not stride over the data embedded between instructions, it will mistakenly interpret them as executable code. So one problem of this technique is that the disassembler will not be aware of the error until it constructs an illegal instruction. Another is that the technique will leave many unprocessed bytes as data without securing the identification of them, while recursive traversal disassembly will leave as minimum unprocessed data as possible.

Recursive traversal, which is the disassembly technique *IDA pro* applies[34], starts to analyze from the program's entry point. By the control flow analysis that based on the basic block recognition, the technique will trace following all the possible succeeding execution path, as shown in figure 2.4.

2.1.4 Junk Bytes

So a method to break recursive traversal is to disturb the recognition of basic blocks and obfuscate the execution path of these basic blocks. Junk bytes is such an effective technique to obstruct recursive traversal disassembler[41]. Junk bytes is defined to have two properties,

- Instructions in a junk bytes should be functional partially, not completely functional.
- Instructions in a junk bytes should not be reachable at run time.

```
DisasmRecursive( ADDRESS startAddr, ADDRESS endAddr )
{
    recur_addr = startAddr;

    while( startAddr <= recur_addr <= endAddr )
    {
        if( addr has been visited already )
            return;

        Instr = decode instruction at address recur_addr;
        mark recur_addr as visited;

        if( Instr is a return )
            return;

        if( Instr is a branch or function call )
        {
            for each possible succeeding addr_branch of Instr
            {
                DisasmRec( addr_branch, endAddr );
            }
            return;
        }
        else
        {
            recur_addr += InstrLength;
        }
    }
    return;
}

main:
{
    startAddr = address of the first executable byte;
    endAddr = startAddr + text section size;

    DisasmRecursive( startAddr, endAddr );
    return;
}
```

Figure 2.4: Recursive traversal algorithm

Following these rules, a method to construct junk bytes insertion is illustrated in figure 2.5. The junk bytes in figure 2.5 will not have execution fall through it.

For our proposed algorithm we will use an enhanced method of junk bytes as shown in figure 2.6. The fake conditional jump instruction in preceding *BBL*(Basic Block) is constructed with a self-modifying code. Disassembler will mistakenly recognize the fake execution path from the fake conditional jump instruction into the junk bytes BBL. Since recursive traversal is not aware of illegal instructions, our method will lead disassembler

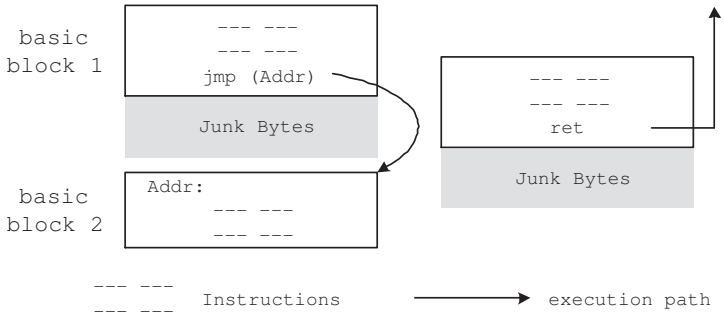

Figure 2.5: Construction of junk bytes

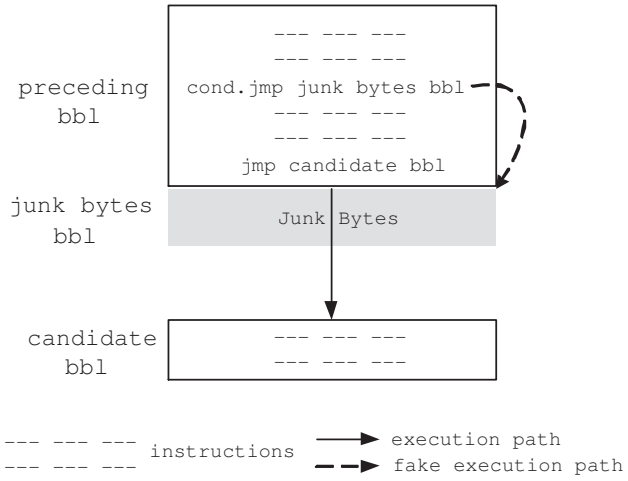

Figure 2.6: Enhanced construction of junk bytes

into junk bytes. Because junk bytes is partial function, the complexity of analysis increases and an unexpected disassembly result will be produced.

2.2 Obfuscation Evaluation Metrics

We evaluate the efficacy of obfuscation with programs from the *SPECint-2006* benchmark suite for a standard evaluation comparing with other related techniques. When a new obfuscating transformation is proposed, it is unclear how to measure the quality of such transformation as there is no general agreement on this matter in this new research

domain. So far, the metrics discussed in [17] is regarded as an all-around evaluation standard. The four aspects of metric include potency, resilience, cost and stealth.

Potency measures how complex the software will be after obfuscation transformation. Complexity grows with the number of software's predicates, inter-basic block variable dependencies, formal parameters and the complexity of software data structures. Resilience measures how well an obfuscation transformation holds up under attack from an automatic de-obfuscator. The metric mainly measures the performance of anti-attack from automatic de-obfuscator. Cost measures time and space penalty which a transformation incurs on an obfuscated application. Stealth measures how much obfuscated code resemble the original code. We have reviewed most of the proposed literature, although it is stated that wild differences between obfuscated code and original code is not accepted, none of the existing experiment provided substantial data to evaluate this metric until now.

However, we need quantitative approach to assess the complexity and cost of obfuscation. The evaluation of obfuscation is also represented as instruction and control flow graph disassembly errors. We use such experimental data to evaluate our proposed algorithm.

Control flow disassembly errors originate from that disassembler mistakenly constructs control flow graph, which does not exist in original program since control flow graph has been changed by our self-modifying code. We experiment this metric by comparing the control flow graph(CFG) of the disassembled program CFG_{disasm} and that of the original program CFG_{orig}[54]. Evaluation of control flow disassembly error is expressed in terms of confusion factor CF_{cfg}, which is computed using Eq. 2.1

$$CF_{cfg} = \frac{|CFG_{disasm} - CFG_{total}|}{CFG_{total}} \qquad \text{(Eq. 2.1)}$$

Where CFG_{total} refers to the set of all edges in original program and CFG_{disasm} refers to the set of all perceived edges in obfuscated program. $|CFG_{disasm} - CFG_{total}|$ is the set of edges that are the extra ones re-constructed by the disassembler erroneously.

We evaluate the instruction disassembly errors with a measure named confusion factor CF_{instr}. Confusion factor CF_{instr} measures the fraction of instructions in the obfuscated code that were incorrectly identified by a disassembler. Let T_{total} be the set of all actual instruction addresses in original program and T_{disasm} be the set of all disassembled instruction address. $|T_{total} - T_{disasm}|$ is the set of instruction addresses that are not identified by the disassembler. Confusion factor CF_{instr} is defined to be the fraction of instruction addresses that the disassembler fails to identify[54], as shown in Eq. 2.2.

$$CF_{instr} = \frac{|T_{total} - T_{disasm}|}{T_{total}} \quad \text{(Eq. 2.2)}$$

We evaluate the effect of obfuscation on execution speed with $Time_{eff}$ defined as,

$$Time_{eff} = \frac{T_{obf}}{T_{ori}} \quad \text{(Eq. 2.3)}$$

Where T_{ori} refers to the execution time of original file and T_{obf} refers to that of obfuscated code[5][46][54].

And, the effect of obfuscation on space is defined as $Space_{eff}$. Obfuscation techniques, such as Control Flow Flattening[72] and two-process obfuscation[25], have space bloat in code section and data section. The definition of $Space_{eff}$ is,

$$Space_{eff} = \frac{S_{code1} + S_{data1}}{S_{code0} + S_{data0}} \quad \text{(Eq. 2.4)}$$

Where S_{code0} and S_{code1} is the size of code section before and after obfuscation. S_{data0} and S_{data1} is the size of data section before and after obfuscation.

2.3 Experimental Environment

An experimental environment is basis of obfuscation algorithm evaluation. The goal of our research is dynamic obfuscation at link time as discussed in section 1.3. There are many link-time optimizers available, including Diablo from Ghent University, Dynamo from HP, Mojo from Microsoft[13], PLTO from University of Arizona and Spike from Compaq. Among them, Diablo and PLTO are source code open and for Intel Pentium platform. We select Diablo and PLTO to apply our obfuscation algorithm.

The benchmark that we use to evaluate is run on a system with 2.6 GHz Pentium and 2GB main memory running *Ubuntu* Linux. The programs were compiled with *GCC* version 3.2.2 at optimization level -O3. We measure the efficiency of obfuscation techniques using two metrics, control flow (edge) disassembly errors and instruction disassembly errors[54]. IDA Pro, a commercial disassembly tool is used to disassemble obfuscated code in comparison with original clear code. We evaluate control flow disassembly errors and instruction disassembly errors of our technique with disassembly result of IDA Pro 4.7.

2.3.1 PLTO(Pentium Link-Time Optimizer)

PLTO is a binary rewriting system working on Intel IA-32 architecture[3][62]. PLTO's primary function is to optimize program at link-time at machine code level with goal-directed value profiling technique[47][76].

The procedure that PLTO processes on a source file is shown in figure 2.7. Usually, the source file is a binary statically relocatable file, namely file.O3, which is compiled at optimization level 3 with GCC compiler. Since code addresses will be updated to reflect the results during optimization, source file to be processed by PLTO should be a statically linked executable file with relocation information. Relocation information

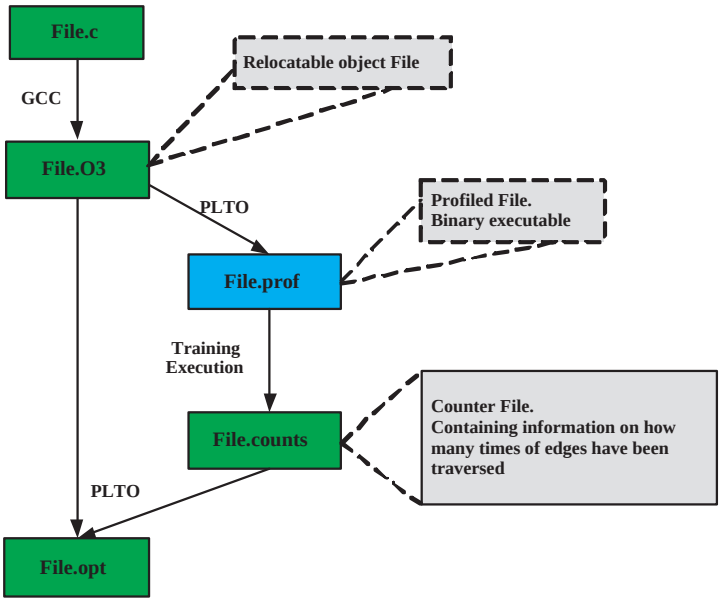

Figure 2.7: PLTO processing procedure

helps to distinguish addresses from data. The object file is processed in three stages to produce an optimized version of the program.

1. Profiling

In the first stage, PLTO inserts profiling block, or instrumentation code, into the binary file to gather execution profile. Profiling blocks is defined to be a sequence of instructions to be inserted between two basic blocks connected with edge. Profiling block updates a counter created in data segment each time when the block is traversed. Correspondingly, profiling blocks will be inserted along each edge in the inter-procedural control flow graph. The counter in data segment records how many times the corresponding edge has been traversed at training execution time. This stage output an executable file named "File.prof" as shown in figure 2.7.

2. Training execution

Before obfuscation, we need to run the profile program to get profile information. In the simulation run of the profiled program, it is executed on different training input

set respectively. Profiling block along each edge will be triggered to count representative weights for the edges in the inter-procedural control flow graph of the program. Counters in data segment record how many times the edge is executed. The profile data gathered by training execution will output to a file. Finally, we combine all the profile file by adding together the execution time of same block and edge. This stage out a file named "File.counts" in figure 2.7. It is processed together with original relocatable object file in optimization or obfuscation stage.

3. Optimization or obfuscation stage

In the final stage, PLTO uses the profile gathered by training execution to analyze and optimize the program. We assume the simulated input set covers all the situations. And the optimizer is enabled to remove the dead code at link time with the help of profiling technique. That is why we give the reason that link time obfuscation provides more security than source code level and compile time obfuscation.

2.3.2 Universal Patching Mechanism

Usually, for link time optimization, linker often moves instructions from one address to another address, or removes unnecessary dead code for speed or storage size performance. In order to keep the program being synchronized in accordance to the original program, PLTO introduces universal patching mechanism.

Universal patching mechanism applies to image of the program at final stage, which means mutation could happen to program several times prior to the final calculation of address. It keeps record of all the operations related to the address calculation. At final stage, linker calculates final address relocation with several iterations.

With the help of universal patching mechanism, the optimizer is enabled to freely move instructions, sections, and other objects at link time. The operation relates to instructions containing address as operand, such as instruction that loads memory address

into register, jump instruction with offset in operand, call instruction with offset in operand, etc. The category of these instructions is necessary for self-modifying code algorithm implementation.

2.3.3 Obfuscation Techniques Based on PLTO

Many algorithms have been proposed previously based on PLTO(Pentium Link-Time Optimizer)[41][54]. Linn et al. discusses a method on the protection of disassembly, especially on how to prevent code from Recursive Traversal attack[41]. Recursive Traversal processes a function as a block and works on the level of function call to ignore the error caused by interpreting data and invalid opcode. This paper focuses on the idea of inserting confusion code into original code. The inserted confusion code is junk byte code which has no relation with original code and the junk byte code will not be executed.

2.3.4 DIABLO(Diablo Is A Better Link-time Optimizer)

Diablo is a link-time binary rewriting system[1][21]. Similar to PLTO, the source file of Diablo is also a binary statically relocatable file. Diablo necessitates some patches to be applied on GCC-based tool chain to preserve information of mapping about the code and data segment at compile time[4]. The mapping information should not be lost during processing since they are necessary for Diablo.

Loco is a branch of Diablo with the GUI Lancet and extended to apply some obfuscation algorithm, such as Control Flow Flattening[75] and opaque predicates[17]. We evaluated the efficacy of Control Flow Flattening with Diablo.

2.4 Conclusion

In this chapter, we have a literature survey on obfuscation, including the various algorithms and reverse engineering techniques. We also present the evaluation metrics and

our implementation environment. In the following several chapters, we will propose new algorithms and evaluate them with the reverse engineering techniques and evaluation metrics.

Chapter 3

Self-Modifying Code Obfuscation Based on Control Flow Flattening

In this chapter, we introduce the concept of self-modifying code firstly. By analyzing the mechanism of self-modifying code, we discuss how this technique advances in software protection, especially in obstructing the static disassembly method. Then we look into how Control Flow Flattening acts on program and how we can use the technique as basic of our algorithm. After that, we propose our novel obfuscation algorithm. A specification on our algorithm is given with details on some selection strategies on different choices of model. Experimental results on the evaluation metrics discussed in chapter 2 are presented before we discuss the security issues with the proposed algorithm. We also explored the efficiency of obfuscating single instruction to many instructions based on Control Flow Flattening and discuss the improvement.

3.1 Concept of Self-Modifying Code

Self-modifying code refers to program, which has the capacity to mutate at runtime[5][11]. Self-modifying code can be used for malicious purposes, such as in virus evolution, which transform virus in every generation to evade detection. But it also can work as a mechanism for software protection if properly handled. Conservative disassembly methods,

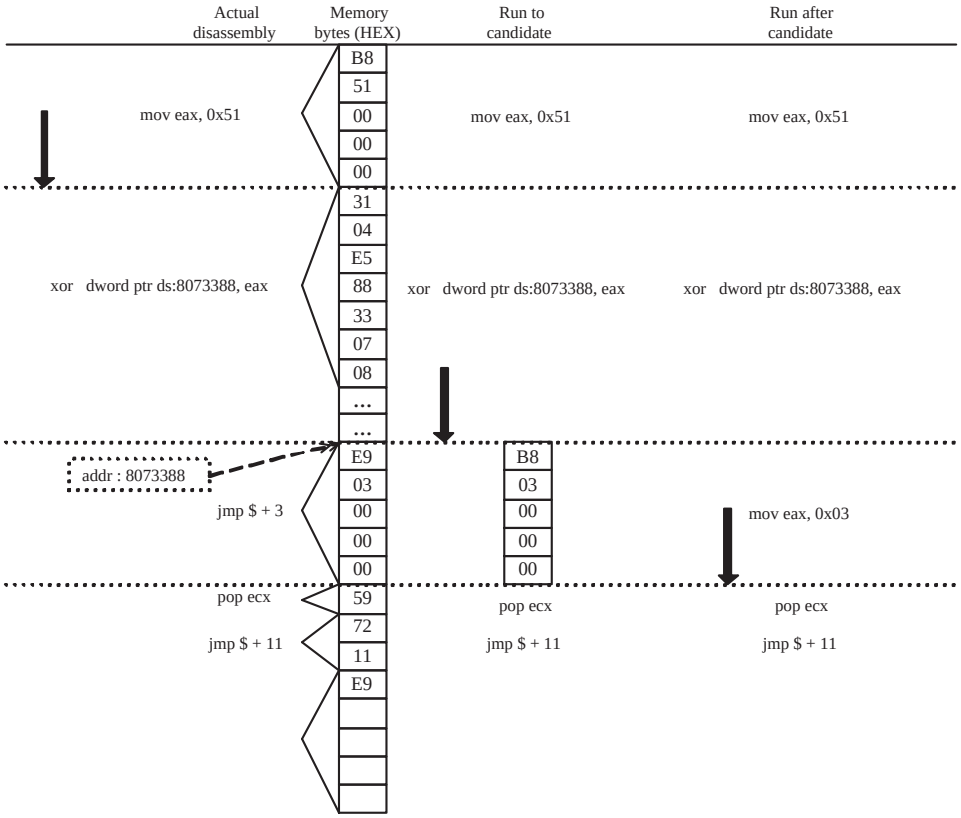

Figure 3.1: Self-modifying code actions

such as linear sweep, are not efficient at self-modifying code identification. The working mechanism of self-modifying code is shown in figure 3.1.

Figure 3.1 shows an example running on 32-bit little-endian CPU. The hexadecimal data in "Memory bytes" column is the machine code loaded to memory by loader before execution. The three vertical hollow arrows represent where the program counter runs to. When program counter runs to a modifying code, the "xor" instruction, it will calculate the exclusive OR result of the value in register eax against the value in specified address. This modification results in that the bytes occupying the first 8 bits of specified address is modified. And then, the modified memory bytes produce a "mov'" instruction and get executed immediately after "xor" instruction. In other words, the control flow of the program has changed significantly after the self-modifying code acts. It is hard for

a static disassembler to predict such a change of control flow at execution time, not to mention the situation when this technique is employed throughout the program. In one word, self-modifying code technique improves code on resistance to reverse engineering in that it is hard to collect all the control-flow since target address value is changing consequently in execution.

3.2 Concept of Control Flow Flattening

Control Flow Flattening is proposed by Chenxi Wang[72][75]. It is an industrial obfuscation tool developed by Cloakware Inc.[14]. The main idea is to flatten the control flow graph by introducing a dispatcher variable to guide the execution sequence of basic blocks. We define *basic block* as a sequence of instructions that are executed under the same control conditions, which means if the first instruction of the block is executed, the others are executed as well. In other words, *basic block* is defined to contain single entry and single exit instruction. Program control flow graph consists of basic block and edge information, so they become the basis of program analysis. To implement Control Flow Flattening, the start address of each basic block in program will be extracted to a jump table and then execution logic of the program, or these basic blocks, is controlled by the dispatcher variable. The value in dispatcher variable represents the offset of the address of next basic block to be executed in jump table. A sample of Control Flow Flattening is shown in figure 3.2 and figure 3.3.

The initial idea of Control Flow Flattening at source code level obfuscation is proposed by Chenxi Wang et al in [74]. They proposed the construction of data-flow on source code level, and the control-flow will be effectively available when made co-depend with data-flow. Control flow transformation is realized by decomposing high-level control structures into if-then-goto constructs and then re-direct the target addresses of the goto's to that determined dynamically by the predicates which is decided by the variables of routine.

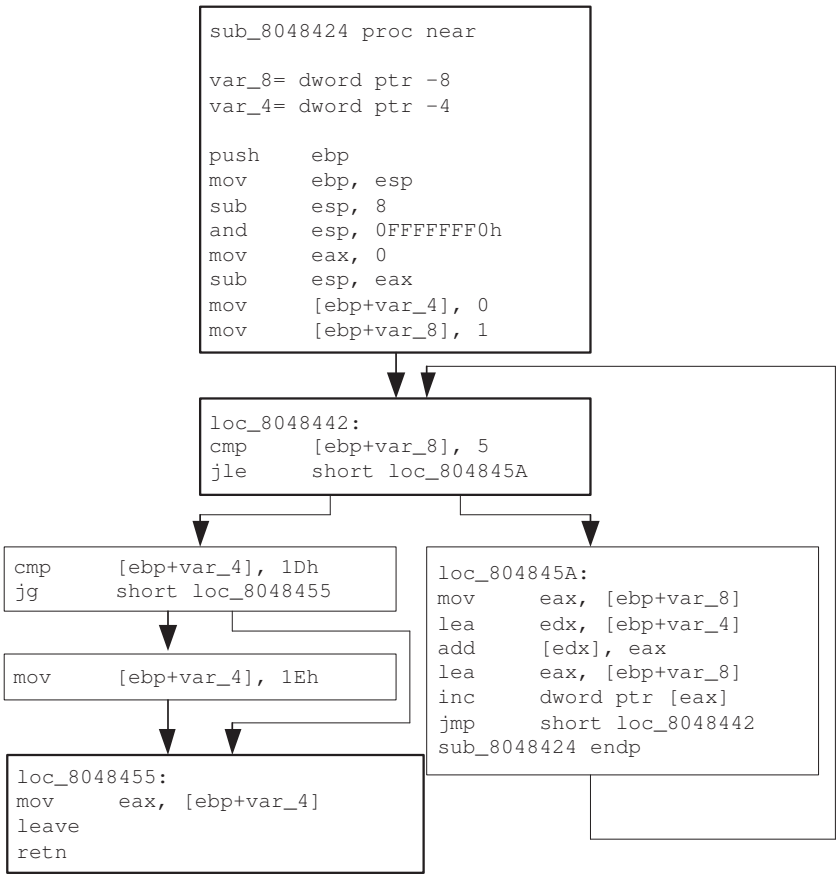

Figure 3.2: Original Control Flow Graph(CFG)

On this way, the general linear order executed routine could be degenerated to a flattened control-flow. And the branch of routine going to be executed is random selected based on the dynamic state of variables. So the problem lies on data-flow transformation.

Because precise alias detection is hard, especially in the presence of general pointers and recursive data structures, data-flow transformation will introduce non-trivial aliases into program. The transformation includes two techniques: Dynamic computation of branch targets and Alias through pointer manipulation. Dynamic computation of branch targets introduces more accessorial computation variables, which need more computation mixing the program intrinsic variables to get the value of accessorial, to decide control-flow. So the computation data and the data that decide control-flow have alias

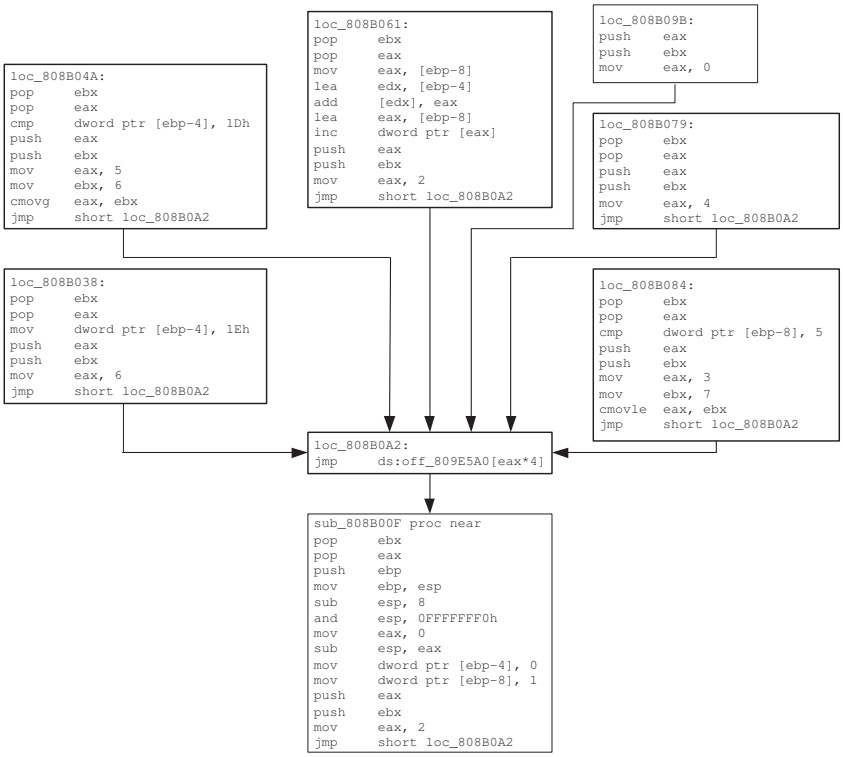

Figure 3.3: CFG after Control Flow Flattening

relationship. Alias through pointer manipulation introduces aliases by use of the pointer type variables replacing the intrinsic variables and array elements. So assignments to variables and array elements will appear as assignments to the pointer type variables on static analysis. However, the Alias through pointer manipulation will be efficient only when the language has powerful support of pointer type variables on source code level.

Control Flow Flattening does not introduce disassembly errors. Disassembler such as IDA pro can distinguish basic blocks and figure out the edge connection between switch block and any respective basic blocks, as shown in figure 3.3. The algorithm is designed to confuse the attacker on the execution sequence of basic blocks and so that of the program since all basic blocks appear to have equally weighted predecessors and successors. Beyond obfuscation, Control Flow Flattening is also used for watermarking[78] and self-protecting mobile agents[19].

3.3 Review of Self-Modifying Code Obfuscation Schemes

The technique of implementing self-modifying code in obfuscation has been discussed in previous research[5][37][46]. Madou et al.[46] proposed the idea of mutating a program as it executes. The mutation is realized by an editing engine running edit scripts. Edit script is defined to mirror different codes, of which must have sufficient similar instruction sequences, to a same memory. The more similarity on the different codes, the more simple is the edit script. The technique discussed that a region of memory would be occupied by many different code sequences during the execution. Two types of mutation are specified, namely one-pass mutation and cluster-based mutation. One-pass mutation is to alter portions of a procedure in the program and place a stub at the entry point of the procedure. At the first running, these portions will be restored as the routine will go into the stub to execute the editing engine and the stub will be removed later. Then the program counter will return to the called address and routine will execute normally at next time. Applying one-pass mutation to cluster of procedures with same edit script is cluster-based mutation. In the cluster-based mutation, each call to the cluster of procedures is replaced by a stub that invokes the edit engine. But how to construct the arguments of edit engine to distinguish different cluster procedure is not specified in the paper. If the arguments are defined as the entry point address of procedure, there may be security problem, such as disassembler can detect entry point address. Also, other than edit script, how to protect the edit engine is another consideration for the technique.

Another technique is to insert dummy instructions to original code at fixed intervals, which is indicated by a defined step[37]. Dummy instructions will be restored to original code at execution time. This technique suffers from several limitations. One is that the method necessitates a precise decision on control-flow since loop operation may plunge the state of the dummy instructions into chaos. And it seems that the suggestion of simply setting a number of camouflaged instructions with a definite interval will confuse

the obfuscator itself significantly or even collapse the obfuscator if the control flow of program becomes complex.

3.4 Control Flow Obfuscation Based on Self-Modifying Code

We propose a dynamic mutation technique based on the concept of preceding and succeeding blocks in control flow graph analysis. The preceding or succeeding block means that it executes immediately before or after the selected block. The proposed algorithm takes advantage of Control Flow Flattening and junk bytes concepts to improve the reverse engineering difficulty.

In the proposal, obfuscator works on the control flow graph within each basic block by modifying instructions to camouflage the edge link of control flow graph. Normal instructions are obfuscated to control flow instructions and vice versa, which will introduce control flow disassembly errors. Here, normal instructions refer to assembly instructions except control flow instructions such as jump, call and return. Program behaves as original one since these control flow instructions will be modified to original clear code before execution, and restored to obfuscated code after execution to obstruct memory copy attack.

Our proposal does not introduce any additional modules to the program. However, obfuscation techniques, such as double-process obfuscation[25] and dynamic code mutation with edit engine[46], need to spawn an ancillary process or script to realize program transformation. The problem is how to protect additional module effectively and the protection will introduce time and space overhead. In our algorithm, the code itself brings in confusion on the control flow. It is hard for a disassembler to distinguish self-modifying code and modified code from normal code in both static and dynamic attacks since the modified code itself will not cause program interruption or segmentation fault. Finally,

our technique makes it difficult for the disassembler to identify the control flow of the program due to the employed Control Flow Flattening technique.

3.4.1 Proposed Obfuscation Technique

The obfuscation technique we propose is based on self-modifying code and the analysis of execution logic of the preceding and succeeding blocks in agent code control flow. Here, the concept of preceding or succeeding refers to edge link, rather than physical connection, between two basic blocks. The preceding or succeeding block means that it executes immediately before or after the candidate block at execution time.

Firstly, we define the terms candidate, modifying and restoring instructions:

- *Candidate Instructions*: a set of instructions that is obfuscated, which will introduce confusion when the program is under reverse engineering. *Candidate block* is the block containing candidate instructions. The instructions at A2 and B2 in figure 3.4(a) are candidate instructions.
- *Modifying Instructions*: a set of instructions that are inserted into the preceding block of candidate block. Modifying instructions modifies candidate instructions to original code. *Modifying block* is the block that contains modifying instructions. The instructions at A1 and B1 in figure 3.4(a) are modifying instructions.
- *Restoring Instructions*: a set of instructions that are inserted into the successors of the candidate block. Restoring instructions restores candidate instructions back to obfuscated code. *Restoring block* is the block that contains restoring instructions. The instructions at A3 in figure 3.4(a) are restoring instructions.

Here, the preceding or succeeding block refers to logical connection, rather than physical connection. The preceding or succeeding block means that it executes immediately before or after the candidate block at execution time. At program execution time, candidate instructions are modified to original code immediately before execution and restored

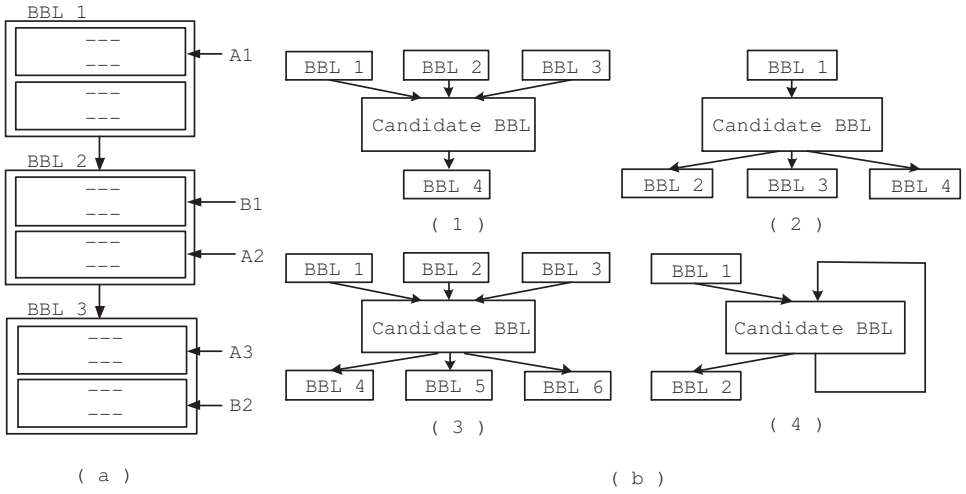

Figure 3.4: Our Self-Modifying Technique

to obfuscated code immediately after execution. Modifying and restoring instructions work on the candidate instructions before and after their execution respectively. The procedure is shown in figure 3.4(a). The entry and dispatcher block is excluded from implementation.

In figure 3.4(a), we assume that BBL1, BBL2 and BBL3 execute in sequence and BBL2 is selected as candidate block. The working mechanism of obfuscator is described as below,

1) At program obfuscation time, instructions at A2 location are obfuscated to fake instructions. And modifying instructions are inserted at A1 and restoring instructions at A3.

2) At program execution time, A1 instructions modify obfuscated instructions at A2 to original code. When program counter goes to A2, the program runs as original code does. After that, instructions at A3 restore instructions at A2 to obfuscated instructions as if they are executed in obfuscated form for an attacker.

Figure 3.4(a) shows a straightforward edge link of basic blocks. Actually, the situation is complex in constructing control flow graph. We show various edge link models in

figure 3.4(b).

- One candidate block may follow several modifying blocks depending on the number of its predecessors, and
- One candidate block may be followed by several restoring blocks depending on the number of its successors, and
- One candidate block may have several modifying blocks and restoring blocks simultaneously, and
- The candidate block may be the predecessor or successor of itself simultaneously.

Since the execution of these blocks is only available at run time, the algorithm has following rules to define the action of modifying and restoring blocks,

1) Modifying instructions should take into account candidate instructions in each succeeding block. And restoring instructions restore candidate instructions in all preceding blocks.

2) Modifying/restoring instructions do not work if the succeeding/preceding block is modifying block itself.

3) Modifying/restoring instructions in candidate block should not be obfuscated. Obfuscator does not work on the modifying/restoring instructions in candidate block.

Following these rules, the obfuscator is enabled to distinguish where to implement self-modifying code automatically among flattened basic blocks. A procedure of instruction and block evolution through our technique is shown in Figure 3.5 and 3.6.

3.4.2 Incorporation of Control Flow Flattening

An enhancement to the proposed algorithm is to implement it and the Control Flow Flattening[72][75] together. Control Flow Flattening is designed to confuse the attacker on the execution sequence of basic blocks and so that of the agent code since all basic

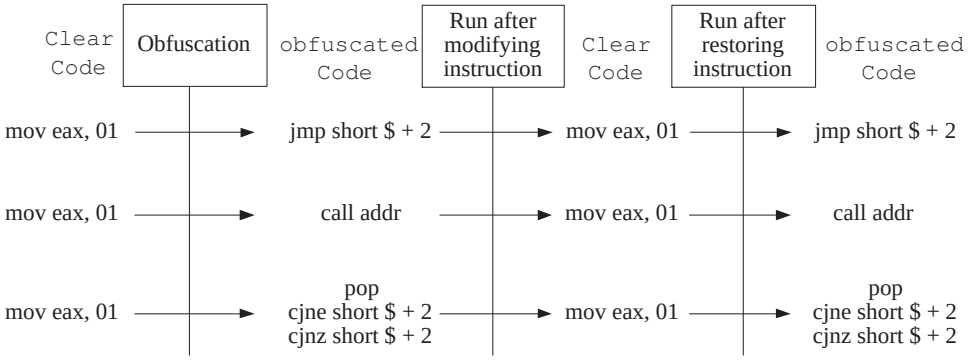

Figure 3.5: Example of instruction evolution in obfuscation

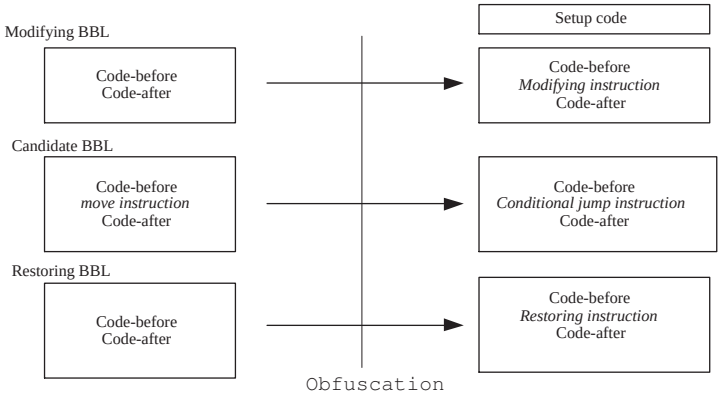

Figure 3.6: Block evolution in our obfuscation

blocks appear to have equally weighted predecessors and successors. The attacker cannot identify the execution sequence of basic blocks. This is the reason why we combine our algorithm and Control Flow Flattening since the attacker can not figure out the execution sequence of modifying instructions, candidate instructions and restoring instructions.

However, our algorithm incorporates Control Flow Flattening to increase the disassembly errors significantly. Control Flow Flattening does not introduce disassembly errors. Disassembler such as IDA pro can re-construct basic blocks and figure out the edge link between dispatcher block and any respective basic blocks, as shown in figure 3.3. Our algorithm introduces both control flow and instruction disassembly errors. The fig-

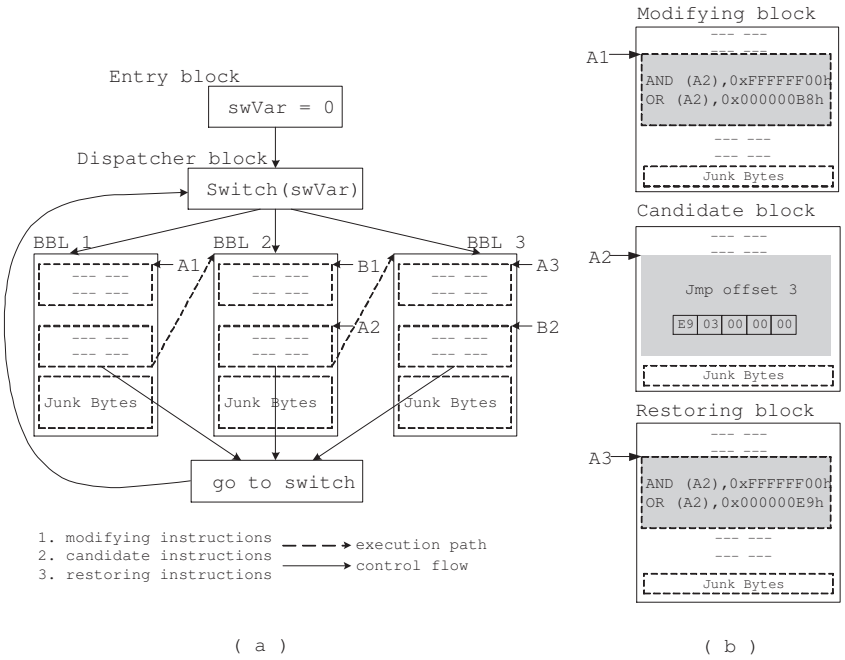

Figure 3.7: Junk bytes in our obfuscation

ure 3.7(a) shows the enhanced proposed algorithm, including Control Flow Flattening and Junk Bytes.

3.4.3 Incorporation of Junk Bytes

The junk bytes can introduce more confusion to the disassembler, which then makes more errors during disassembling. Therefore we further enhance the performance, in terms of reverse engineering difficulty, of the proposed algorithm by adding junk bytes in addition to the Control Flow Flattening. The basic block in Control Flow Flattening has the effect of an unconditional control transfer to the dispatcher block as shown in figure 3.3. To implement our algorithm, Control Flow Flattening and junk bytes is shown in figure 3.7(a).

In that case the junk bytes immediately following the basic block are not reachable at execution time. The junk bytes seems to be traceable for such disassembler if we introduce

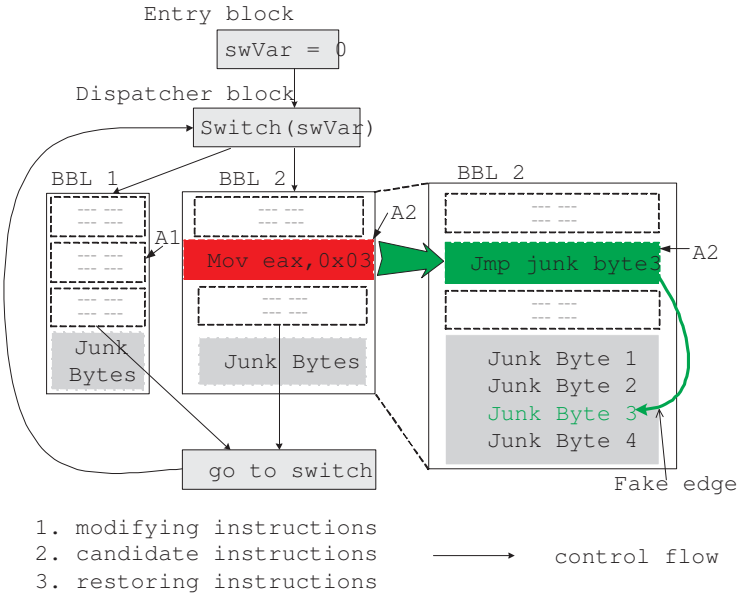

Figure 3.8: Junk bytes strategy 1

fake control flow instructions, which means we replace some control flow instructions with normal instructions, or replace some normal instructions with control flow instructions. Two options of the technique are provided as following. They have similar performance and either of their statistical evaluation is equal to or better than existing techniques.

The first option is to replace normal instruction with control flow instruction as in figure 3.8. A "mov" instruction is obfuscated to an unconditional jump instruction. Since junk bytes are partial instructions, it will camouflage the target address of the jump instruction to be the address of one instruction in junk bytes. The disadvantages of recursive traversal have been discussed in chapter 2. Recursive traversal is not aware of the error edge until it constructs an illegal instruction. So a static recursive traversal will not be aware of the correctness of the junk bytes instructions and the following instructions. But at the execution time, the obfuscated control flow instruction will be modified to original instruction and the junk bytes will not be executed.

The second option is to replace control flow instructions with normal instructions.

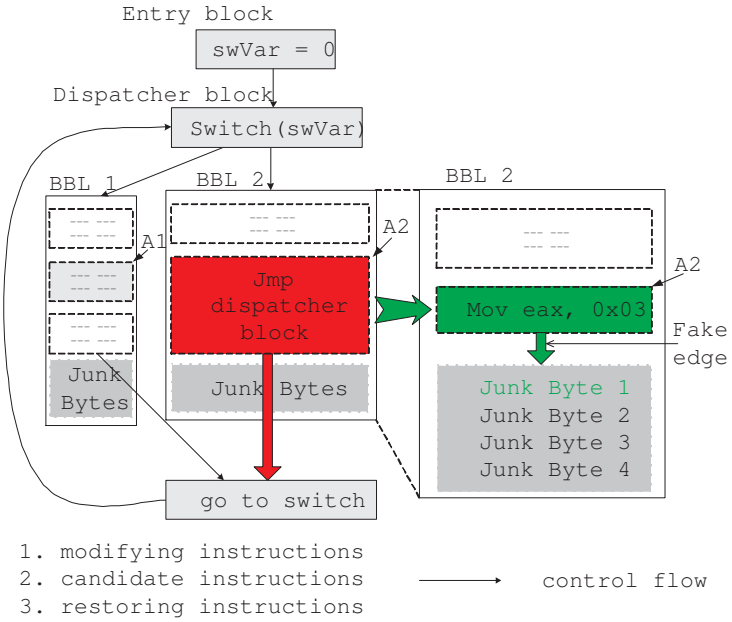

Figure 3.9: Junk bytes strategy 2

The method is shown in figure 3.9. The last instruction in each flattened block is an unconditional jump instruction. The unconditional jump instruction is obfuscated to a normal instruction. Disassembler will be misled to junk bytes instruction although the execution path never goes into junk bytes at execution time.

In previous discussion, we provide two types of candidate instructions selection strategy, to obfuscate a normal instruction or a control flow instruction. Also, there is a selection strategy for restoring instructions, such as A3 instruction in figure 3.4. However, if the self-modifying code is left in system memory as original form after execution, software is vulnerable to several kinds of attack.

Firstly, attacker can disassemble the code read out from memory after program execution.

Secondly, with the binary code read out from memory after execution, attacker could compare it with the original binary executable file. Although the comparison result does not reveal our algorithm directly, it helps a human intervention to locate candidate

instructions.

Therefore, it can be seen, restoring instructions are needed and would provide a more secure solution in that it enable our technique to leave no trace of self-modification in memory. The proposal of Kanzaki[37] also has this advantage, while Madou's[46] does not have. With restoring instructions, the cost to collect candidate instructions and restoring instructions through program is inestimably high even by human intervention, since restoring instructions execute immediately after candidate instruction. The disadvantage of restoring instructions is in little more execution time overhead.

3.5 Implementation and Experiment

We have implemented the proposed algorithm on PLTO platform and tested the performance of the technique with evaluation metrics discuss in chapter 2.

3.5.1 Implementation

The implementation is designed to camouflage instructions and insert modifying instructions and restoring instructions respectively at link-time. Our implementation contains following steps,

Step 1. We choose the last "mov" instruction in each flattened block as candidate instruction. This is because, in control flow flattened program, there is at least one "mov" instruction, which will store the offset of address of the next basic block to be executed in jump table.

Step 2. The candidate instruction is obfuscated to an unconditional jump instruction. We obfuscate the opcode of a "mov" instruction, that is "0xB8" with the opcode of unconditional jump instruction -"0xE9". We do not obfuscate operands of candidate instructions.

Step 3. To insert modifying instructions to each preceding blocks. The choice of where to insert the modifying instructions is decided after liveness analysis of the block. The insertion of modifying instructions at A1 is shown in figure 3.7(b). Here, we use the unconditional jump (to same segment) instruction to obfuscate move data (immediate to register) instruction. So, we insert modifying instructions"AND (A2), 0xFFFFFF00h" and "OR (A2), 0x000000B8h" in modifying block. After the "AND" instruction is executed, the opcode of obfuscated instruction will become "0x00h" from "0xE9h" and it will change to be "0xB8" after the "OR" operation.

Step 4. To insert restoring instructions at A3 to each succeeding blocks. The insertion of restoring instructions is shown in figure 3.7(b). We insert restoring instructions"AND (A2), 0xFFFFFF00h" and "OR (A2), 0x000000E9h" in restoring block. After the "AND" operation, the opcode of candidate instruction will become "0x00h" from "0xB8h" and it will be restored to unconditional jump instruction opcode "0xE9" after the "OR" operation.

Step 5. To insert junk bytes immediately after the last unconditional jump instruction of candidate block.

The procedure is repeated until all candidate blocks through agent code is implemented with self-modifying code. Another option to the implementation of our algorithm is to select the last instruction, which is always an unconditional jump instruction, in each flattened basic block as candidate instruction. Replacing the last jump instruction with normal instruction will also make disassembler fail to construct basic block since it fails to identify exit point.

3.5.2 Evaluation of the algorithm

We measure the efficiency of obfuscation using two metrics, control flow edge disassembly errors and instruction disassembly errors[54].

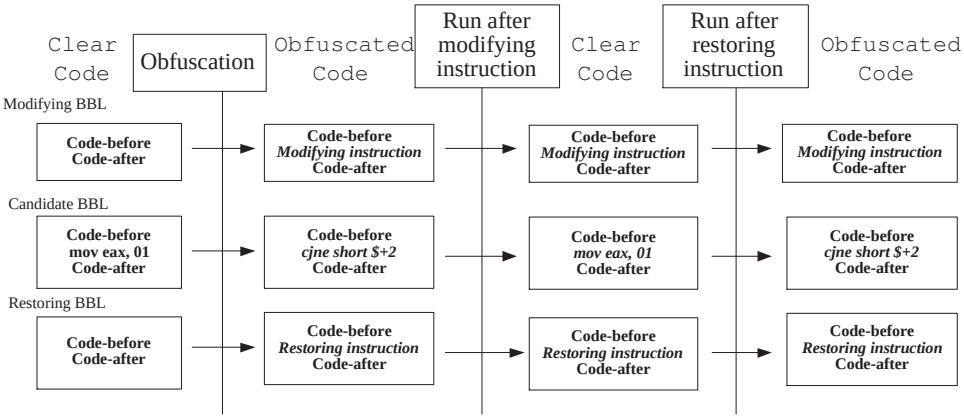

Figure 3.10: Overview evolution in our obfuscation

Table 3.1: Control flow disassembly errors

PROGRAM	CFG_{total} (E1)	$\|CFG_{disasm} - CFG_{total}\|$ (E2)	CFG_{smc} (E3)	Confusion factor ((E2-E3)/E1)
bzip2	21,694	28,248	14,618	62.83%
h264ref	37,976	55,214	30,552	64.94%
hmmer	32,417	51,386	28,045	72.00%
lbm	19,400	26,448	14,045	63.93%
mcf	19,941	26,698	14,220	62.57%
sjeng	24,876	31,368	16,022	61.69%

Control flow and instruction disassembly errors are expressed in the manner of control flow confusion factor CF_{cfg} and instruction confusion factor CF_{instr}. Besides the edges that originating from candidate instructions, our algorithm also introduces more than 61.69% control flow disassembly errors as shown in table 3.1.

The experimental data of instruction confusion factor CF_{instr} is listed in table 3.2. The reason for this high degree of disassembly errors is that IDA Pro only disassembles addresses that can be identified as instruction addresses. This identification method has two effects: first, large portions of the code that are reached by branch function addresses are simply not disassembled, being presented as hex data. And secondly, candidate instructions are treated as control flow instructions, and this causes basic blocks in obfuscated program to be re-constructed erroneously and some junk bytes to

Table 3.2: Instruction disassembly errors

PRO-GRAM	T_{total} (T1)	T_{disasm} (T2)	Confusion factor ((T1-T2)/T1)
bzip2	301,766	138,809	54.00
h264ref	611,251	284,421	53.47
hmmer	529,851	238,466	54.99
lbm	279,650	125,525	55.12
mcf	284,171	128,026	54.95
sjeng	331,454	152,954	53.85

be erroneously disassembled together with some other actual instructions. We can see that besides the control flow edge errors, our technique also confuse disassembler on instruction identification.

Table 3.3: Effect of obfuscation on text and data section sizes (KB)

PRO-GRAM	TEXT SECTION			DATA SECTION			COMBINED TEXT+DATA		
	Ori (T0)	Obf (T1)	Change (T1/T0)	Ori (D0)	Obf (D1)	Change (D1/D0)	Ori(C0) (T0+D0)	Obf(C1) (T1+D1)	Change (C1/C0)
bzip2	362.0	1,005.7	2.78	6.5	103.7	15.99	368.5	1,109.3	3.01
h264ref	780.9	2,100.3	2.69	12.4	209.6	16.8	793.3	2,309.9	2.91
hmmer	534.7	1,755.3	3.28	10.7	195.1	18.27	545.4	1,950.4	3.58
lbm	309.9	925.7	2.99	3.3	96.5	29.4	313.2	1,022.2	3.26
mcf	317.5	941.0	2.96	3.3	97.6	29.72	320.8	1,038.6	3.24
sjeng	407.7	1,114.5	2.73	9.9	116.8	11.8	417.5	1,231.3	2.95

Table 3.3 shows the impact of the obfuscation on text and data section sizes. The size of the text section increases from 2.91 to 3.58 times of original size. This increase arises from the insertion of modifying blocks, restoring blocks and junk bytes. The growth in the size of the initialized data section is considerably larger, ranging from a factor of 11.8 (*sjeng*) to a factor of 29.72 (*mcf*). The large growth in the size of the initialized data section has two reasons. One reason is that the size of data section in original program is small and so an increase would look considerably large. Another reason is that the jump table in Control Flow Flattening is stored in the initialized data section. Therefore, we evaluate the impact of the obfuscation with the total increase in program size, or the

Table 3.4: Effect of obfuscation on execution speed

PROGRAM	Original (T_{ori})	Obfuscated (T_{obf})	Slowdown (T_{obf}/T_{ori})
bzip2	202.51	270.9	1.34
h264ref	1882.56	3015.76	1.60
hmmer	1204.09	1215.08	1.01
lbm	1253.39	1291.84	1.03
mcf	462.83	486.26	1.05
sjeng	980.14	995.14	1.02

sum of text section and initialized data section. We can see that program will bloat to 3 times of original size.

The speed performances of the obfuscated programs are shown in table 3.4. Overall, the experiment data proved that our proposed technique is effective even against the best state-of-the-art disassembly tools, especially on the obfuscation of control flow and instructions.

3.6 Improved Algorithm

In order to improve the performance of the algorithm, we propose to mutate single instruction to multiple instructions. The proposed algorithm makes use of self-modifying codes for the purpose of program obfuscation. In this proposal the self-modifying codes are realized by camouflaging single normal instruction to multiple control flow instructions.

We propose to make use of the redundant bytes in operand of instructions, especially of Control Flow Flattening instructions, to obfuscate single instruction to multiple instructions. Redundant bytes refer to the bytes within an instruction that does not carry information, such as “0x00” in many instructions. In flattened blocks, an example of the instruction containing redundant bytes is the “mov” instruction which stores the offset of address of the next basic block to be executed in jump table. Typically, the “mov”

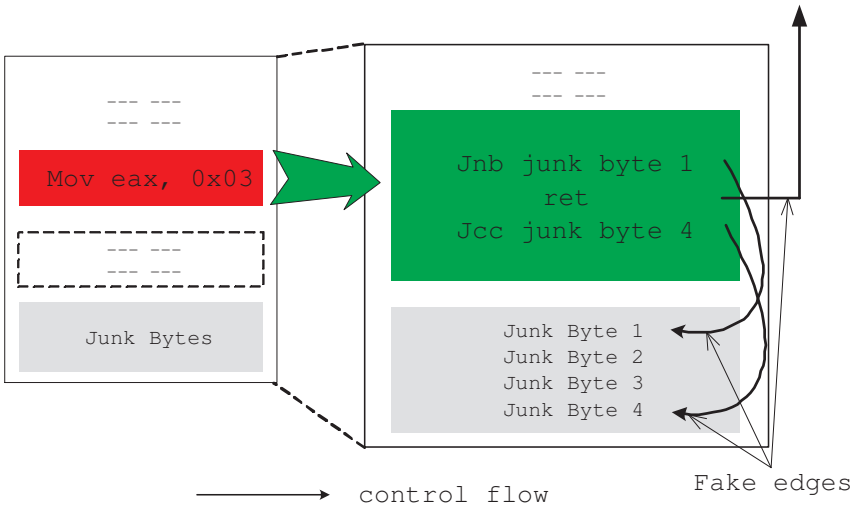

Figure 3.11: Fake edges introduced by the algorithm

instruction carries three redundant bytes - "0x00", and they can be obfuscated to carry redundant branch control flow for a disassembler.

To explain the algorithm, we take the "mov" and "jmp" instructions for example as illustrated in figure 3.11. Firstly, we obfuscate the opcode of candidate instruction. For example, in case of non-branch control flow, the opcode and operand for a "mov eax, 0x5" instruction is "0xB8 0x05 0x00 0x00 0x00". An option is to obfuscate the "mov" instruction to an unconditional jump instruction by obfuscating the opcode. The opcode for obfuscated instructions could be obfuscated to "0xE9 0x05 0xFF 0xFF 0xFF", which will be disassembled as "jmp near + 0xffffff05", same disassembly result for both recursive traversal and linear sweep.

For the "mov" instruction, besides the option to obfuscate opcode, one more option is to obfuscate it to three control flow instructions - a return instruction and two conditional jump instructions. For example, the opcode and operand could be obfuscated to "0x73 0x01 0xC3 0x73 0x05", which will be disassembled as "jnb +1", "*retn*" *and* "*jnb* + 05". The obfuscated instruction could be modified to original clear code easily with boolean calculation AND and OR.

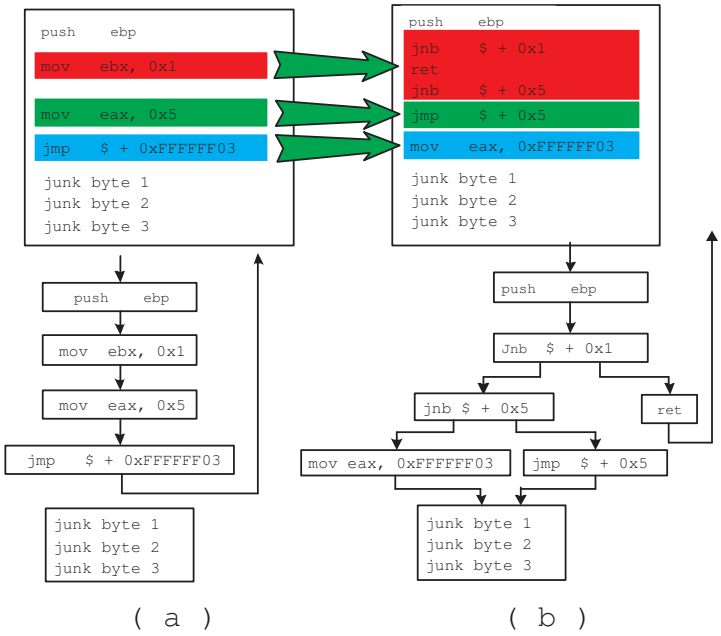

Figure 3.12: An overview of control flow graph mutation

We give models to obfuscate control flow graph by working on the opcode and redundant operand of instructions as illustrated in figure 3.12. In fact, the models could be applied to other instructions as well to camouflage instructions that construct the control flow graph of program so long as the linker does not complain.

We evaluate the performance of the improved algorithm with such metrics as instruction confusion factor CF_{instr} and control flow confusion factor CF_{cfg}. The experiment result of the two metrics is shown in table 3.5 and table 3.6. The CF_{cfg} performance comparison with original proposal is shown in table 3.7. The algorithm improves CF_{cfg} about 8.06% at the cost of instruction disassembly error performance.

The speed and program size performances of the obfuscated programs are shown in figure 3.13 and figure 3.14. The large growth in the program size is because that the jump table in Control Flow Flattening is stored in the initialized data section. We can see that program will bloat up to 3.23 times of original size.

Table 3.5: Control flow disassembly errors

PROGRAM	CFG_{total} (E1)	$\|CFG_{disasm} - CFG_{total}\|$ (E2)	CFG_{smc} (E3)	Confusion factor ((E2-E3)/E1)
bzip2	21,694	30,167	14,618	71.67%
h264ref	37,976	55,277	27,552	73.00 %
hmmer	32,417	54,211	27,369	82.80%
lbm	19,400	28,163	13,992	73.05 %
mcf	19,941	29,376	14,648	73.86 %
sjeng	24,876	33,460	15,946	70.41%

Table 3.6: Instruction disassembly errors

PROGRAM	T_{total} (T1)	T_{disasm} (T2)	Confusion factor ((T1-T2)/T1)
bzip2	322,599	155,565	51.78 %
h264ref	615,636	310,332	49.59 %
hmmer	560,531	268,992	52.01 %
lbm	298,839	141,579	52.62 %
mcf	310,240	147,235	52.54 %
sjeng	353,296	170,010	51.88 %

Table 3.7: Performance comparison with original proposal

PROGRAM	CF_{cfg} 1	CF_{cfg} 2	Improvement
bzip2	62.83%	71.67%	8.84 %
h264ref	64.94%	73.00%	8.06 %
hmmer	72.00%	82.80%	10.8 %
lbm	63.93%	73.05%	9.12 %
mcf	62.57%	73.86%	11.29 %
sjeng	61.69%	70.41%	8.72 %

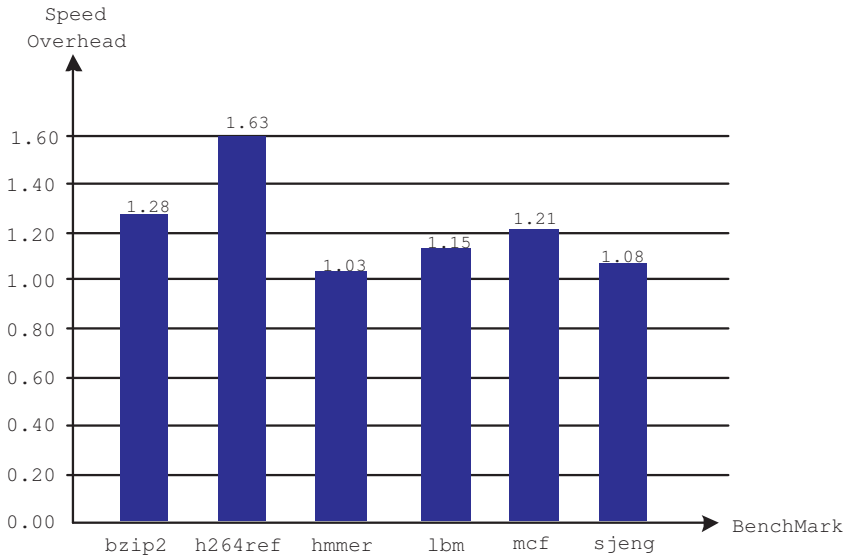

Figure 3.13: Speed overhead of obfuscation

3.7 Security Analysis of The Algorithm

We proposed a new algorithm to protect program sensitive information and implemented a model that obfuscates normal instruction to control flow instruction. The experimental data based on benchmark suites show that the performance of our algorithm is statistically equal to or better than that of the obfuscation using signals[54], especially on protecting the control flow graph and instructions from attacking with comparable space and time overhead.

Our algorithm performs better on confusing the disassembler on re-constructing 61.69 % more edges that does not exist. Also, our algorithm has a considerably improved performance of instruction protection. More than 53.47% of the total instruction addresses in original can not be disassembled correctly. We can see that program will bloat up to 3 times of original size, this is partially because the algorithm has been applied all over basic blocks in the program and so it provides wide protection.

Except the statistical data, our algorithm has the advantage that a disassembler will not be aware of the whole set of instructions since it fails to identify basic blocks. The

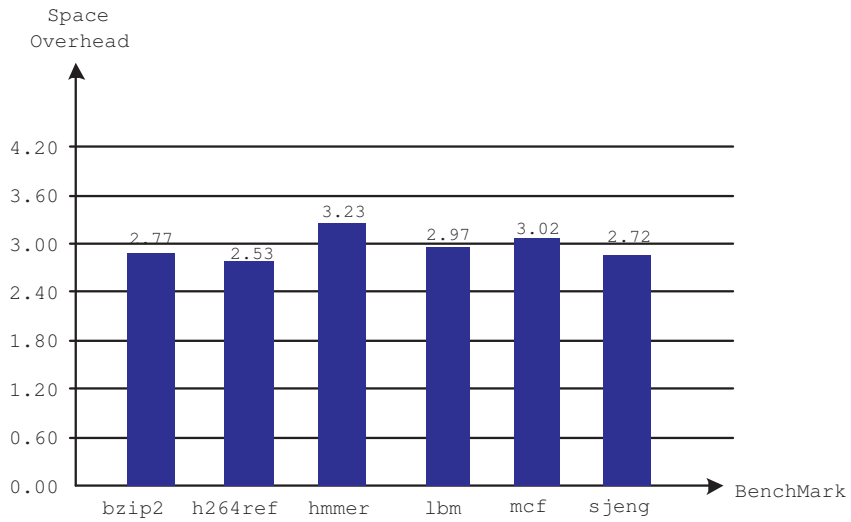

Figure 3.14: Space overhead of obfuscation

candidate instruction(s) that we choose to obfuscate will generate an exit instruction and disassembling techniques based on basic block and edge construction will then collapse. We provide two models to confuse the disassembler on constructing basic blocks. Either of the method will camouflage the structure of a basic block. Also, the candidate instruction(s) will introduce confusion edge, which misleads the disassembler on the execution logic of the program.

Besides the advantage discussed, one more important point of our technique is that it will hide the clue to reverse engineering even if the program is cracked by human intervention. The beneficial point comes from the fact that the set of self-modifying code is tight and it executes continuously. A set of self-modifying code never reveals its modifying and mutating code at the same time. Also, a large number of such self-modifying code are widely distributed over the program. Therefore, it is a huge job for a disassembler to identify them from normal instruction precisely. So it is hard for a disassembler to figure out the basic block and identify the self-modifying code, not to mention extract high level semantic information and design an automatic reverse engineering tool.

Our algorithm has the advantage that a disassembler will not be aware of the whole set of instructions if it fails to identify basic blocks. With the feature that a basic block has only one exit point, we provide two ways to trap the disassembler on constructing basic blocks. One is to replace instructions before exit instruction with control flow instructions. Another is to replace the exit instruction, which is definitely a control flow instruction, with a non-control flow instruction. Either of the method will camouflage the structure of a basic block.

Then we will discuss how the technique is resistant to static disassembly technique discussed in chapter 2. The reason why our method resists to these static reverse engineering techniques is that the disassembler will fail to reconstruct the basic block. The candidate instruction that we choose to obfuscate generates an exit instruction. Disassembling techniques based on basic block will then collapse. Also, the candidate instruction introduces some confusion edges, which mislead the disassembler on the execution logic of the program.

The obfuscation is implemented at link-time. If handled properly, it could be applied to instructions in every basic block all over the program because it is possible to find at least one reasonable predecessor and one successor for every basic block except the very beginning one and the last one. An enhancement to the method is to replace normal instructions with control transfer instructions, such as jump and call. The control transfer instructions will confuse the disassembler when calculating the displacement of the instruction. Moreover, instructions can be replaced by several instructions so long as the opcode and displacement before and after modification remains the same. This will confuse the disassembler not only on the recognition of basic block, but also on the discovery of the execution logic.

Our proposed method does the obfuscation efficiently. Firstly, it is not necessary to introduce any more module to the program. Obfuscation techniques, such as two-process

obfuscation[25] and dynamic code mutation with edit engine[46], need to spawn an ancillary process or script to realize program transformation. But in our algorithm, the code itself brings in confusion on the control flow. Secondly, the technique disables the intention of designing an automatic disassembler. It is hard for a disassembler to distinguish self-modifying code and modified code from normal code in both static and dynamic attacks since the modified code itself will not cause program interruption or segmentation fault immediately after its execution. Finally, our technique does not necessitate precise calculation of control flow. One may argue that the self-modifying code may expose the execution logic of basic blocks in Control Flow Flattening. However, the fact is that it is hard to collect all the control-flow since target address is changing consequently at execution time.

3.8 Conclusion

We proposed a new algorithm to apply self-modifying code based on Control Flow Flattening. The algorithm protects program by introducing difficulties in re-constructing both basic block execution sequence and control flow. Although Control Flow Flattening does not introduce disassembly error, the algorithm proves itself to be efficiently resistant to static analysis in that it successfully camouflage the original control flow by re-directing the program control to unpredicted address during disassembly. We developed the algorithm on PLTO at link time and evaluated the experiment data with metrics discussed in chapter 2. With the experimental results, we showed that the technique is effective in confusing the disassembler on control flow analysis. Also, the algorithm is easy to implement, self secured, and being robust against human intervention to crack. The research work in this chapter has been published in the paper [63].

Chapter 4

Self-Modifying Code Obfuscation Based on Basic Blocks

In this chapter, we proposed an obfuscation algorithm based on basic blocks. However, the storage size overhead of the algorithm based on Control Flow Flattening in chapter 3 is 3 times of that of the original program. We explored a new algorithm to reduce the storage size overhead and to achieve better performance in evaluation metrics.

4.1 Motivation

In chapter 3, we propose to obfuscate program with self-modifying code based on Control Flow Flattening. The performance of instruction disassembly error, control flow(edge) disassembly error and execution time is competitive to techniques such as signal-based obfuscation. But, the storage size overhead is not satisfied that it is 3 times of the original program. We need to explore a new algorithm to restraint the overhead. Also, we need to introduce random property to increase the complexity of our protection schedule.

4.2 Proposed Obfuscation Technique

In this section, we propose a new algorithm based on self-modifying code to obfuscate instructions with more efficacy, especially on hindering the identification of basic blocks

and edge connections. We also present an explanation on the implementation of the proposed algorithm in details and discuss the incorporation of our algorithm with junk bytes in this section.

4.2.1 Obfuscation Process

In the new algorithm, we propose to create new basic blocks and edge connections at obfuscation time. Some additional control flow instructions are inserted after the creation of new BBL, since they are necessary for the purpose of keeping the behavior of program. Then, basic blocks and control flow are secured by obfuscating control flow instructions with self-modifying code technique. In this way, the control flow obfuscation is realized by camouflaging control flow instructions with normal instructions. A global view of the algorithm on a basic block is shown in figure 4.1. The algorithm is discussed in the following paragraph.

Firstly, we create new basic blocks and edge connections by splitting each basic block into separate parts. The split is operated before a randomly selected instruction within the basic block. After the split operation, each of the separated parts is still a basic block because *basic block* is defined to be consisting of a sequence of instructions that are executed under the same control conditions. The fact that the separation instruction is selected randomly does not introduce complexity to the program. But, the cracker will spend much more time to re-construct basic block since it is not an easy job to make out where the separation instruction is. We randomly select the instruction where to split basic block for the purpose of disrupting the cracker's actions, rather than building a complex program like conventional protector-centric obfuscation techniques do.

Then, we insert necessary control flow instructions for two motivations. The first one is to obfuscate the control flow graph. Here, we say that the inserted control flow instruction has obfuscation effect because it is possible to design the control flow branch

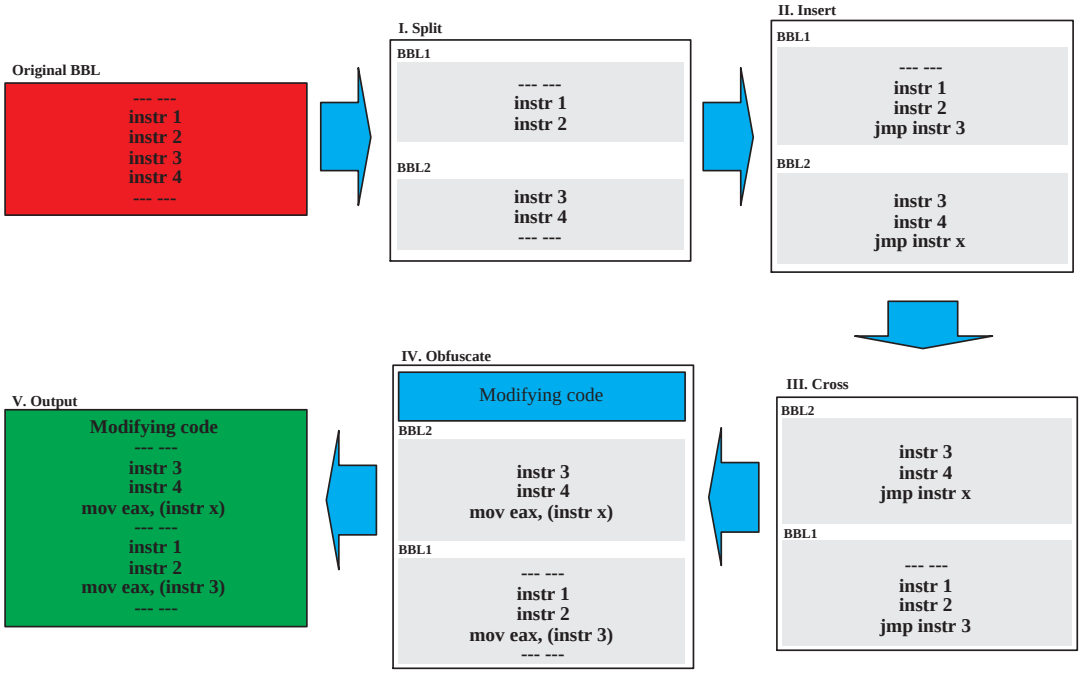

Figure 4.1: An overview of the proposed algorithm

in accordance to the control condition of the basic block. In that way, the inserted instruction, such as an unconditional jump branch instruction, works in the way of opaque predicates[17][49].

The second motivation is to make sure that the program, of which basic blocks and edge connections have been split, behaves the same way as the original one does.

We define two types of edge connections, named direct edge connection and branch edge connection. These edge connections are shown in figure 4.2.

1) For direct edge connection, we add the control flow instruction connecting the direct succeeding instruction address, as shown in figure 4.2.

2) For branch edge connection, we add the control flow instruction connecting the FALSE branch succeeding instruction address, as shown in figure 4.2.

Further, we obfuscate split basic blocks in a random order as shown in step III in figure 4.1. The step III can intertwine split basic blocks from different parent basic blocks,

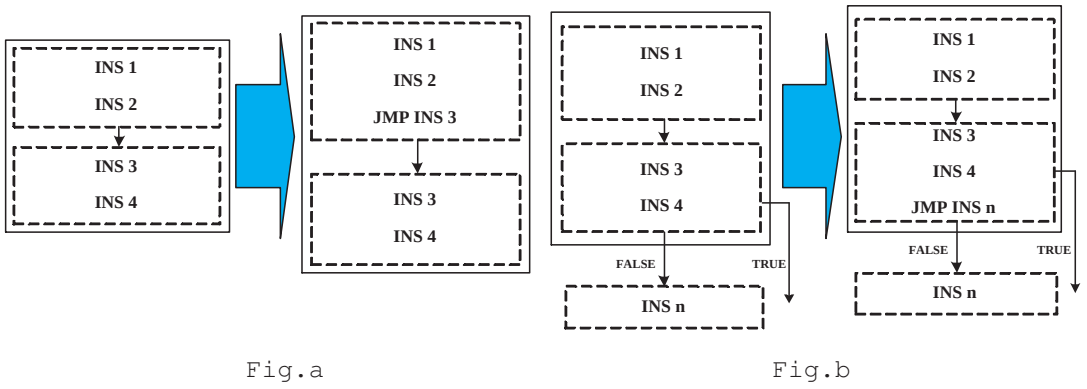

Figure 4.2: Types of Edge Connection

more than two split basic blocks. As a whole, the split and insert operation is shown in step I, II and III in figure 4.1. For return or halt edge, we do not split the basic block and add the control flow instruction.

Finally, we use self-modifying code to prevent the inserted control flow instructions, or in other words, the instruction from where basic block is split, from being reverse engineered. We select the additional control flow instruction as *candidate instruction* or *mutating instruction* and obfuscate them with self-modifying code technique. The working mechanism of the proposed algorithm is discussed as below,

1) At program obfuscation time, control flow graph of basic block, for example BBL2 in figure 4.1, are obfuscated to fake control flow graph. This can be done in the way that the inserted control flow instructions are obfuscated to normal instructions.

2) Modifying instructions are inserted at any location that executes before the obfuscated instruction. At program execution time, candidate control flow graph are modified to original state immediately before execution. Modifying instructions work on the candidate control flow graph before its execution.

4.2.2 One To Many Modification

In [63], we proposed a set of rules to guarantee the proper behavior of obfuscated program with least execution time and storage size overhead. These rules are,

1) Modifying instructions should take into account candidate instructions in each succeeding block. And restoring instructions restore candidate instructions in all preceding blocks.

2) Modifying/restoring instructions do not work if the succeeding/preceding block is modifying block itself.

3) Modifying/restoring instructions in candidate block should not be obfuscated. Obfuscator does not work on the modifying/restoring instructions in candidate block.

However, the set of rules will not perform as expected when basic blocks are split since more protection is applied to the software system.

The algorithm of [63] is a technique that one basic block only works on the succeeding self-modifying code. There comes a problem that it is necessary to calculate all the control flow to make sure that modifying and restoring instructions are generated under these rules. And the calculation will become complex when branch edge connections increase after we insert a great deal of control flow instructions. Also, it is not easy to protect the distributed block of instructions. Since the control flow graph becomes complex after the algorithm is applied, we propose to modify multiple blocks of mutating code, such as all the basic blocks in one function, in one modifying block.

As shown in figure 4.3, within the modifying block, a loop operation is used to modify all of those blocks of mutating code. The offset of mutating instruction from modifying instruction is saved in data segment in random order and they do not expose any information on the execution sequence of these mutating code to crackers. Moreover, the modifying block can be in any function that executes before the function containing candidate blocks, such as the preceding function or the caller function. Besides the security enhancement, one more benefit of the one to many modification is the restraint of storage size overhead.

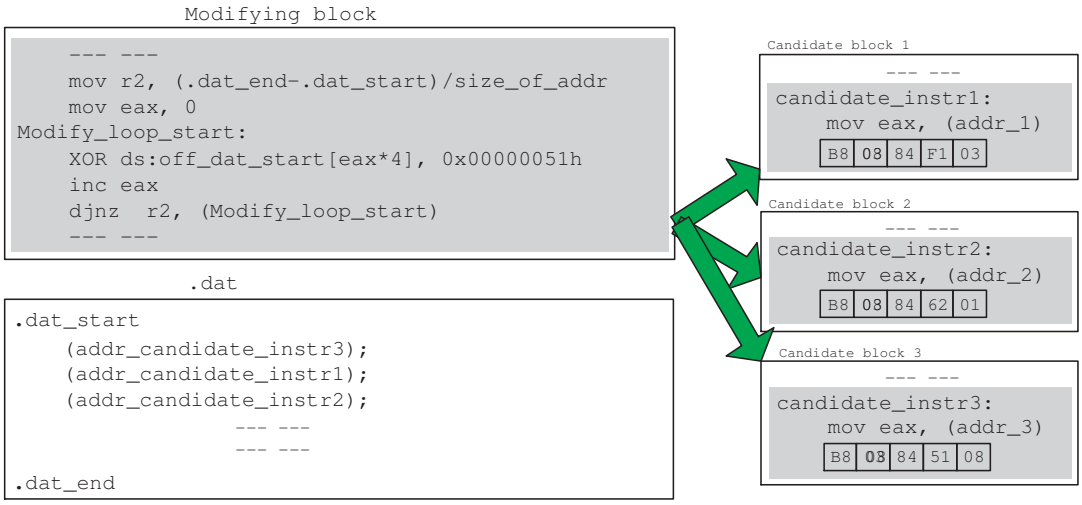

Figure 4.3: One to many obfuscation

4.2.3 Junk Bytes

Although control flow and instruction disassembly errors introduced by our proposal thwart the effort of the attacker, we propose to use junk bytes[41] in addition to protect the modifying code and increase the complexity of the obfuscated program. All these factors will contribute to making reverse engineering both effort and time consuming.

Junk bytes are constructed by inserting partial instructions in the way that they are not executed at runtime. Our proposal introduces more confusion in that the junk bytes seems to be executable for attacker after the obfuscation algorithm since it introduces fake edges by the candidate instruction, which means the candidate instruction constructs fake edges connecting basic block with junk bytes. This can be realized by obfuscating control flow instructions to normal instructions, as shown in figure 4.4.

4.3 Implementation and Experiment

We implement the proposed algorithm at link time. The general method is to implement the self-modifying code on control flow instructions which connect the split basic blocks.

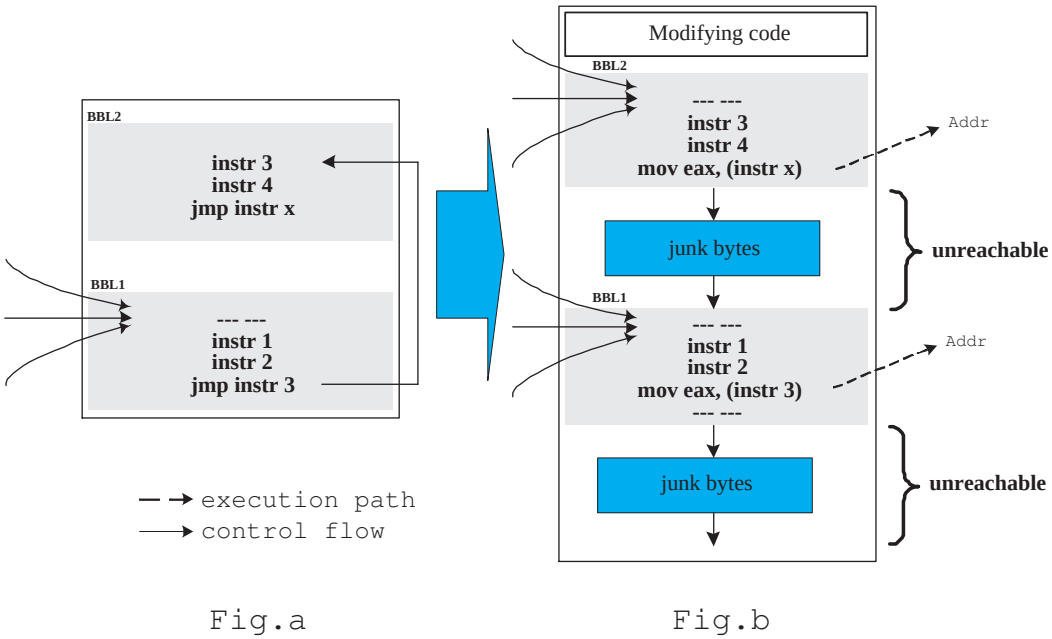

Figure 4.4: Bogus code insertion

4.3.1 Implementation

Our implementation contains following steps,

Firstly, we split basic blocks before a randomly selected instruction. The instruction could be any one within basic block.

Let the position of this instruction be the i^{th} within the basic block and the total number of instructions in a BBL be M. Let N be a pseudo-random number generated by a pseudo-random generator[8]. Then the position I can be computed as "i = N mod M". We split basic block before the i^{th} instruction.

Then, we get two basic blocks and connect them with control flow instruction. The control flow instruction could be a conditional jump instruction or an unconditional jump instruction. The former one acts as opaque predicate, of which the condition is same with that of the basic block.

We select the inserted control flow instruction as candidate instruction and obfuscate the opcode of it as shown in figure 4.3. On this occasion, we obfuscate the opcode of

a "jmp" instruction, that is "0xE9" with the opcode of uncondition jump instruction -"0xB8". Correspondingly, modifying instructions are inserted to preceding functions. We use the move instruction to obfuscate unconditional jump instruction. So, we insert the "XOR" modifying instructions in modifying block. After the "XOR" instruction is executed, the opcode of obfuscated instruction will become "0xE9h" from "0xB8h". The choice of where to insert the modifying instructions is based on liveness analysis of modifying block. Liveness analysis is a technique to calculate whether variables are live or not at a definite place in program. Based on control flow graph, liveness analysis follows the edge connections to run a data flow analysis. With the help of liveness analysis, the inserted modifying instructions do not interrupt the behavior of the program.

Finally, insert junk bytes immediately after the last unconditional jump instruction of candidate block.

The procedure is repeated until all candidate blocks through program are implemented with self-modifying code.

4.3.2 Experimental Results

We measure the efficiency of our technique using two metrics, control flow (edge) disassembly errors and instruction disassembly errors[54] with disassembly result of IDA Pro 4.7. We list the edge disassembly errors in table 4.1. In table 4.1, "E1" refers to all edges in original program. "E2" refers to the set of edges that are extra re-constructed by the disassembler erroneously. And "E3" refers to the set of edges that originates from the obfuscated instructions. Besides the edges that originating from candidate instructions, our algorithm also introduces more than 51% control flow disassembly errors.

The experimental result of instruction confusion factor CF_{instr} is listed in table 4.2. The reason for this high degree of disassembly errors is that IDA Pro only disassembles addresses that can be identified as instruction addresses. This identification method

Table 4.1: Control flow disassembly errors

PROGRAM	CFG_{total} (E1)	$\|CFG_{disasm} - CFG_{total}\|$ (E2)	CFG_{smc} (E3)	Confusion factor ((E2-E3)/E1)
bzip2	21,694	31,231	19,879	52.33 %
h264ref	37,976	55,764	35,569	53.18 %
hmmer	32,417	46,321	29,787	51.00 %
lbm	19,400	27,680	17,520	52.37 %
mcf	19,941	28,518	18,099	52.24 %
sjeng	24,876	35,294	22,324	52.14 %

Table 4.2: Instruction disassembly errors

PROGRAM	T_{total} (T1)	T_{disasm} (T2)	Confusion factor ((T1-T2)/T1)
bzip2	201,919	49,196	75.64 %
h264ref	382,391	99,646	73.94 %
hmmer	305,001	77,287	74.66 %
lbm	177,712	42,912	75.85 %
mcf	182,397	43,193	76.32 %
sjeng	223,828	54,218	75.78 %

has two effects: first, large portions of the code that are reached by branch function addresses are simply not disassembled, being presented as hex data. And secondly, candidate instructions are treated as control flow instructions, and this causes basic blocks in obfuscated program to be re-constructed erroneously and some junk bytes to be erroneously disassembled together with some other actual instructions. We can see that besides the control flow edge errors, our technique also confuses disassembler on instruction identification.

The speed and program size performances of the obfuscated programs are shown in figure 4.5 and figure 4.6. We can see that program will bloat about to 2 times of original size. The improvement of storage size overhead compared to that in [63] is achieved by the one to many proposal. Also, the shift of our algorithm from Control Flow Flattening to basic blocks contributes to the storage size overhead improvement.

The performance of the proposed algorithm is compared with the algorithm with

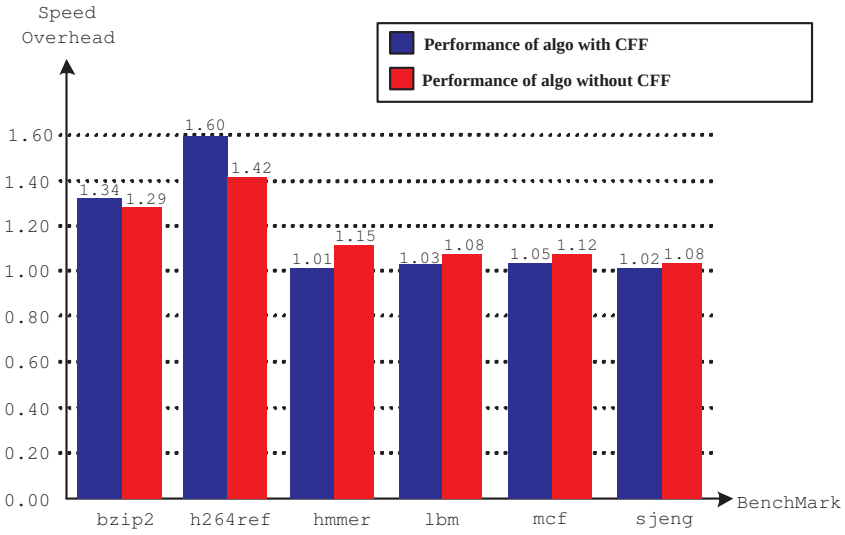

Figure 4.5: Speed Performance of Obfuscation

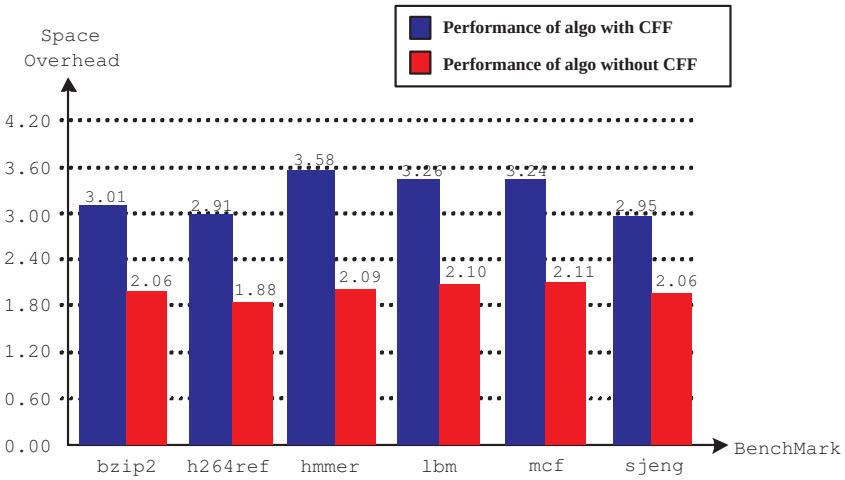

Figure 4.6: Storage Space Performance of Obfuscation

Control Flow Flattening[63] and signal based obfuscation[54]. And the experimental result is shown in table 4.3 and table 4.4. The result in table 4.3 and table 4.4 is the geometric mean of experimental result from benchmarks. The experiment results in table 4.3 prove that compared to the algorithm based on CFF, the proposed algorithm has significant enhancement of instruction disassembly errors(20.97%) and storage size overhead(34.92%), with the cost of 12.36% control flow disassembly errors.

Table 4.3: Algorithm performance comparison

Comparison Item	CFF based obfuscation	BBL based obfuscation	Improvement
Instr disass error	54.39 %	75.36 %	20.97 %
Edge disass error	64.57 %	52.21 %	-12.36 %
Storage size	3.15	2.05	34.92 %

Table 4.4: Algorithm performance comparison

Comparison Item	Signal based obfuscation	BBL based obfuscation	Improvement
Instr disass error	57.28 %	75.36 %	18.08 %
Edge disass error	41.18 %	52.21 %	11.03 %
Storage size	2.39	2.05	14.23 %

As shown in table 4.4, comparing with the signal based obfuscation[54], the proposed algorithm introduces average 18.08% instruction disassembly errors, 11.03% control flow disassembly errors and 14.23% improvement on storage size overhead.

4.4 Security Discussion

Our proposed technique is resistant to static disassembly technique for the following reasons.

Most of the current control flow obfuscation techniques suggest that the central algorithm is in one single module, such as the edit engine and edit script in [5], the M-process in double-process scheme[25] and the signal handler function in signal based obfuscation[54]. This has a problem that the integrated central algorithm is vulnerable to attack and it reveals the intention of obfuscation easily. But, in the algorithm proposed in this chapter, obfuscation algorithm is embedded into program with distributed diversity. Obviously, the gain of the diversity is that the time for disassembly will increase significantly and then achieve the goal to make the cost of program reverse engineering prohibitively high both on time and complexity metric.

The obfuscation technique on single instruction which is selected with definite interval[37] has the limitation that a static disassembler will be able to re-construct control flow before and after the instruction. The proposed algorithm avoids these limitations in that,

- Our proposal targets high performance with a simple implementation and it is not influenced by the loop and branch control flow graph. In this chapter, we discuss that the complexity in program control flow does not act upon our proposal. And so it is possible to achieve a high performance with simple application.
- Our proposal is a control-flow-oriented scheme with high-granularity of instruction coverage. We say that control-flow-oriented scheme means that the algorithm focuses on obfuscating control flow instructions to normal instruction or vice versa. Our experiments reveal that it brings us a higher confusion with lower performance overhead. High-granularity of instruction coverage means that the proposal protects program with higher precision and coverage, but without introducing complexity to the algorithm.

The candidate block that we choose to obfuscate will generate multiple branch after obfuscation and disassembly analysis will then fail to identify basic block precisely. Also, the obfuscated control flow in candidate block will introduce confusion edge, which misleads the disassembler on the execution logic of the program. Our algorithm has the advantage that a disassembler will not be aware of the whole set of instructions, including modifying instructions and candidate control flow graph. Disassembler fails to identify basic blocks since the fake edge connections within original basic block mislead the reconstruction of basic blocks. With the feature that a basic block has only one exit point, we provide two models to confuse the disassembler on constructing basic blocks. One is to replace instructions before exit instruction with control flow instructions. Another is to replace the exit instruction, which definitely is a control flow instruction, with a

non-control flow instruction. Either of the method will camouflage the structure of a basic block.

As to the human intervention attack, the proposed technique will hide the clue to reverse engineering. The point comes from the fact that the set of self-modifying code is tight and it executes continuously. It is a huge job for an attacker to identify them from normal instruction precisely. Basic block consists of a sequence of instructions that are executed under the same control conditions. So it is hard for a disassembler to figure out the basic block and identify the self-modifying code after we break out the control conditions, not to mention extract high level semantic information.

Besides, the proposed method obfuscates software efficiently. Firstly, it is not necessary to introduce any more modules to the program. Obfuscation techniques, such as two-process obfuscation[25] and dynamic code mutation with edit engine[46], need to spawn an ancillary process or script to realize program transformation. But in our algorithm, the code itself brings in confusion on the control flow. It is hard for a disassembler to distinguish self-modifying code and modified code from normal code in both static and dynamic attacks since the modified code itself will not cause program interruption or segmentation fault. Finally, our technique makes it difficult for the disassembler to identify the control flow of the program due to the separation and disorder of basic blocks.

One more advantage of our proposed algorithm is that it is applied through the program with randomness and diversity. We introduce diversity in software protection in order to prevent the risk of breaking one instance will lead to breaking all instances. The mutation is realized by obfuscating the control flow instructions, such as jump, call or return, with self-modifying code. Disassembler is not aware of the fake instructions, and so the fake control flow since the correct control flow information has been eliminated by self-modifying code. But program behaves as original one does since these control flow instructions will be modified to original clear code before execution. Experimental results

show that the algorithm performs well. Also, we extend our algorithm to incorporate junk bytes, since adding junk bytes would cause more complexity during reverse engineering.

4.5 Conclusion

We proposed an obfuscation algorithm based on basic blocks that solves the storage size overhead problem and it also bring us better performance in metrics, such as instruction disassembly errors. The algorithm is applied without Control Flow Flattening, but it does not loss the property of the randomness in block execution sequence. It introduces more randomness in splitting the basic block at random location and thus increases the difficulties in re-constructing basic block and in identifying edge connections. We have implemented the algorithm on PLTO and evaluated with standard *SPECint-2006* benchmark suite. The experimental result shows that the performance of our algorithm is equal to or better than that of the obfuscation using signals[54] in all metrics, especially on protecting the edges and instructions from attacking.

Chapter 5

Self-Modifying Code Obfuscation At Function Level

In chapter 3 and chapter 4, we have explored two types of obfuscation techniques based on self-modifying code. One incorporates the Control Flow Flattening and the other one bases itself on basic blocks. These two algorithms focus on security of the basic block, especially the control flow identification within a function and they perform well in comparison with the competitive signal-based obfuscation[54]. We will discuss the limitation of protection at function level provided by these two algorithms. However, we need an algorithm to protect the program at function level to thwart attacker's effort to construct higher level semantic information.

5.1 Motivation

In this chapter, we investigate an algorithm to obfuscate the program at function level to achieve the advancement in performance. The reason why we need to investigate obfuscation at function level is that the influence on the function relationship of program from the protection on BBL level needs to be strengthened. Attacker may reverse engineer the program on function level, sometimes with the help of symbol table. The gain in such attack is that the errors that committed in disassembling one function does not recursively transfer to the disassembly of other functions.

In chapter 3 and chapter 4, the proposals based on Control Flow Flattening and BBL are designed to hinder the identification of basic blocks and edge connections between basic blocks. Here, edge connections refer to control flow connections introduced by conditional or unconditional jump instructions. But, these algorithms work mainly within a function, so we need to explore the feasibility on obfuscating the control flow of function call.

In applications, some advanced techniques, such as *hook*[50][64] and *callback*[35][58], are necessary to meet the advanced requirement of system. *Hook* is a concept of Operating System that describes the platform on which application is enabled to launch sub-function to monitor definite message or event of other application. *Callback* mechanism assigns the address of callee function to caller function to realize dynamic binding at run time depending on the detail of the message or event that triggers the function call. Sensitive information relating to the system security is often in these special applications with *hook* and *callback*. It is not good to expose these sensitive information to an attacker. A function level protection is necessary for these scenarios.

One more beneficiary party of function level obfuscation technique is mobile software agent protection[9][19][24][60]. In mobile agent model, as shown in figure 5.1, a program suspends their execution on one platform, transits to another platform and resumes the execution. The platform that initiates the program is referred to as home platform, where agent originates. Mobile agent technique enables the mobility of cooperating application among platforms to compute in a large-scale, loosely-coupled distributed system. Security issues on mobile agent includes masquerading, tampering etc. Masquerading agents pose as authorized agent to get access to services and resources which they are not entitled. Tampering means compromised or malicious platforms modify agent's code, state, or data.

Moreover, the threat does not limit to the agent code itself only. It also includes the tampering with agent communications, such as changing the financial transactions of

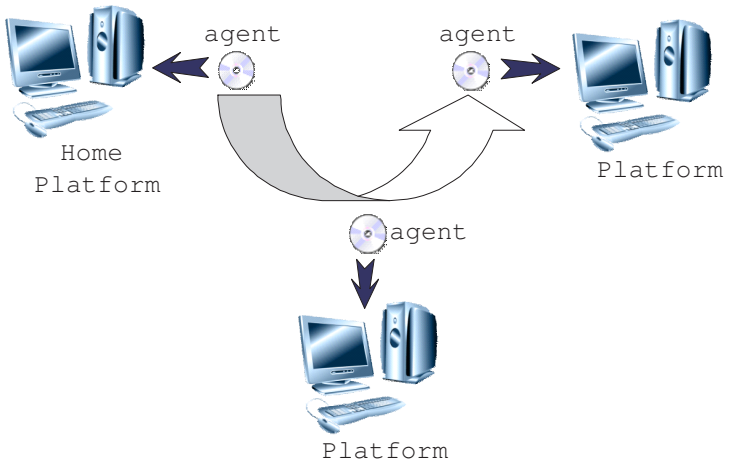

Figure 5.1: Mobile agent system

a "drawback" to a "save in" message intentionally. These issues are related to running agent codes on a compromised platform. Obfuscation plays roles in agent code protection in that the agent does not expose its code to the platform in a manner of easy understanding. The obfuscated code also hides the sensitive information on how to construct communication message. The communication protocol stack often uses techniques such as *hook* and *callback*.

There are proposals on obfuscating the function as a whole, especially encrypt the function body as a whole. This is partially out of the reason that it is easy to keep the integrity of stack in the function. One more proposal is to obfuscate the program's function names, rather than the control flow, at function level. As far as we know, there is no effective control flow protection at function level currently.

For obfuscating the control flow at function level, one way is to protect at the entry point of the function. Alternative way is to protect at the caller point in caller function. The first type of protection encounters symbol table problem. Symbol table is produced by the linker and serves to name functions, variables and data items, to allow the system to do run-time linking, resolving open references to those names to the location where

the library is loaded in memory. The function names and information to locate functions in symbol table are visible to users. Since the symbol table reveals function entry point, especially in a shared library or DLL, security of the first approach will be weakened. Therefore, we propose the protection algorithm by obfuscating the caller point in caller function.

5.2 Proposed Obfuscation Technique

In this section, we propose a new algorithm based on self-modifying code to obfuscate at function level. The goal of the algorithm is to protect program from reverse engineering by thwarting the effort of attacker on constructing the relationship between caller function and callee function.

5.2.1 Obfuscation Process

We present the process of the obfuscation algorithm at function call level. The obfuscation technique we proposed is based on self-modifying code and the analysis on context of the function call instructions.

In the proposal, obfuscator works on the function call relationship between caller and callee function by modifying instructions to camouflage the edge link of control flow graph. Function call instructions are protected either using normal instructions or control flow instructions, which introduce control flow disassembly errors. Here, normal instructions refer to assembly instructions except control flow instructions such as jump, call and return. At run time, the executed function call in program is similar to the original clear code. The proposed algorithm integrates with the algorithm based on basic blocks to improve the reverse engineering difficulty.

We have two options to obfuscate the function call instruction,

- To obfuscate the function call instruction to normal instruction, or

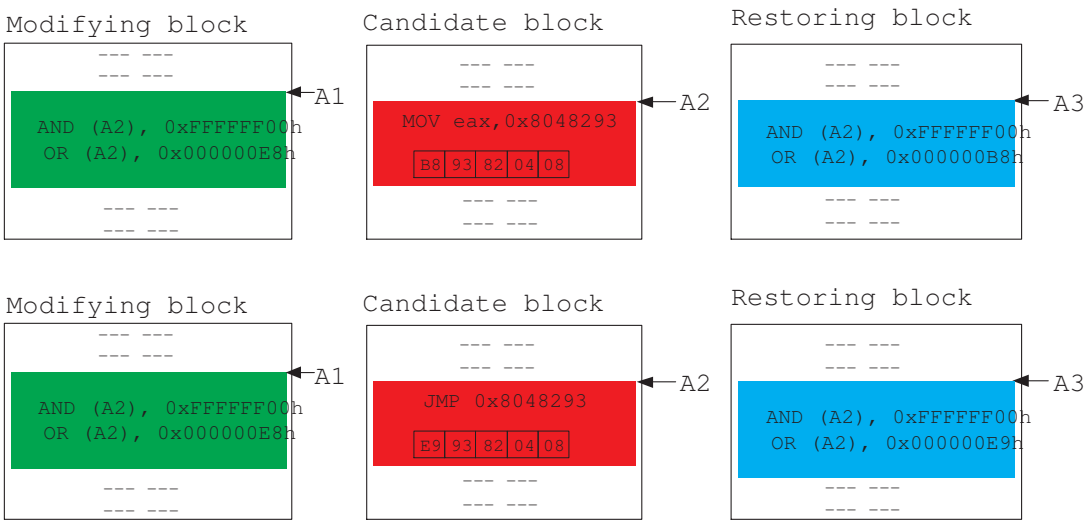

Figure 5.2: Instruction evolution of the algorithm

- To obfuscate the function call instruction to control flow instruction, such as conditional or unconditional jump instruction.

These two options of function call instruction obfuscation is shown in figure 5.2.

In figure 5.2, modifying code and restoring code are inserted into preceding and succeeding blocks after liveness analysis. Here, the preceding or succeeding block means that it executes immediately before or after the candidate block at execution time. Liveness analysis is to decide the liveness of a variable at a particular point by checking whether it is assigned one more time within the path from the point to where the variable is used in the source of an assignment or other statements, such as function call. So, the modifying block should not change the liveness of all the function call variables, since the behavior of program will change if function variables liveness is corrupted.

5.2.2 Stack Analysis

In the case that function call instruction is obfuscated to normal instruction, the proper stack management of caller function will not be influenced. However, in the case that

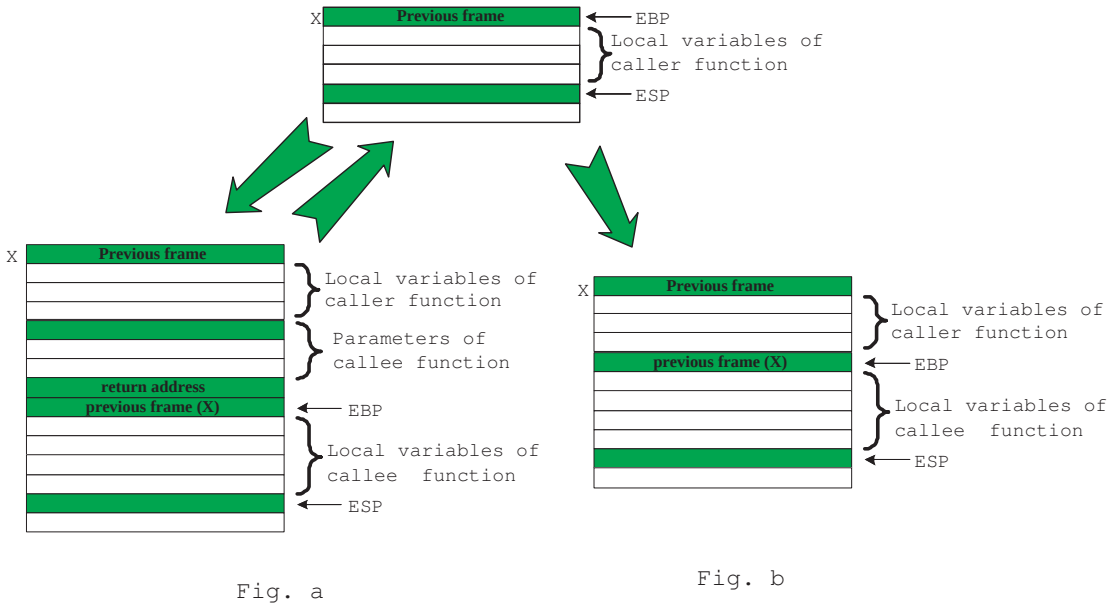

Figure 5.3: Stack analysis of obfuscation

function call instruction is mutated to control flow instruction, such as an unconditional jump instruction, obfuscation is easily detected by the disassembler with stack analysis.

Executing a function is made up of three steps, namely function call, function prolog and function return. Assuming function call instruction is obfuscated to jump instruction, only first two steps are executed and the stack will be occupied with parameters and local variables of called function within each function call. An example on function call stack analysis is in figure 5.3. A stack relocation during function call is shown in figure 5.3(a). However, it is easy for disassembler to detect the abnormal situation in stack since the " EBP" and "ESP" do not restore to the location where they are before the callee function is executed if it is obfuscated to jump instruction, as show in figure 5.3(b). Disassembler can do it even by observing the code statically, and report the abnormal disassembly code to the observer. We give up the effort to obfuscate function call into control flow instruction for this reason.

5.2.3 Junk Bytes

Two cases of applying junk bytes with our proposal are shown in figure 5.4. The code segment on left side is original clear code. After obfuscation, junk bytes are inserted after unconditional jump instruction and return instruction. On right side is the disassembly result of obfuscated code. Junk bytes are marked in red font. We see that the disassembler mistakenly identify the junk bytes and the following clear code together. Since junk bytes are partial function and the property of RISC instruction set, the instruction that followed after junk bytes will be disassembled to many parts. Part of the instruction is identified together with junk bytes and the left part is identified with the part of next instruction. The disassembly procedure can go looping.

Also, the disassembler will identify the code as data mistakenly, being presented as hex data as shown in figure 5.5, if it can not figure out the code and junk bytes separately. This can be a disaster for an attacker if the identification failure is widely distributed in obfuscated program. The mistakenly disassembled data in code section has worse influence if it contains the target address of other control flow instruction. The disassembler stops tracing the control flow since it find that is not an effective address. This is the reason why we construct the fake edge connecting candidate block and junk bytes.

However, the contribution of junk bytes is limited until it is used in situation that the disassembler is recursively misled to it, as the case we proposed in chapter 3 and chapter 4. An example of applying junk bytes, function level obfuscation and basic block based obfuscation is shown in figure 5.7.

5.3 Implementation and Experiments

We implement the function call obfuscation together with the basic block based obfuscation proposed in chapter 4 to evaluate the performance of a total protection to program.

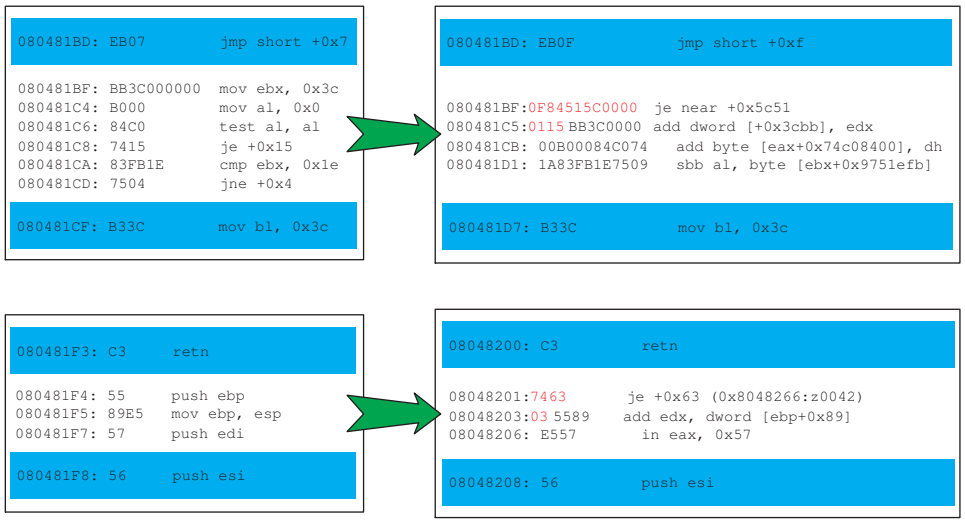

Figure 5.4: Construction of junk bytes

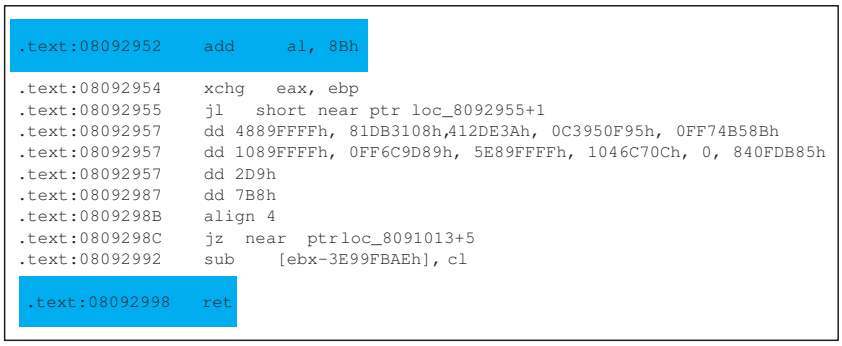

Figure 5.5: Construction of junk bytes

5.3.1 Implementation

We implement the function call obfuscation at link time. The implementation is in the following steps.

Firstly, we select the function call control flow as candidate instruction and obfuscate the opcode of it as shown in figure 5.6. For this, we obfuscate the opcode of a "call" instruction, that is "0xE8" with the opcode of move instruction -"0xB8". Correspondingly, modifying instructions are inserted to preceding functions. We use the move instruction

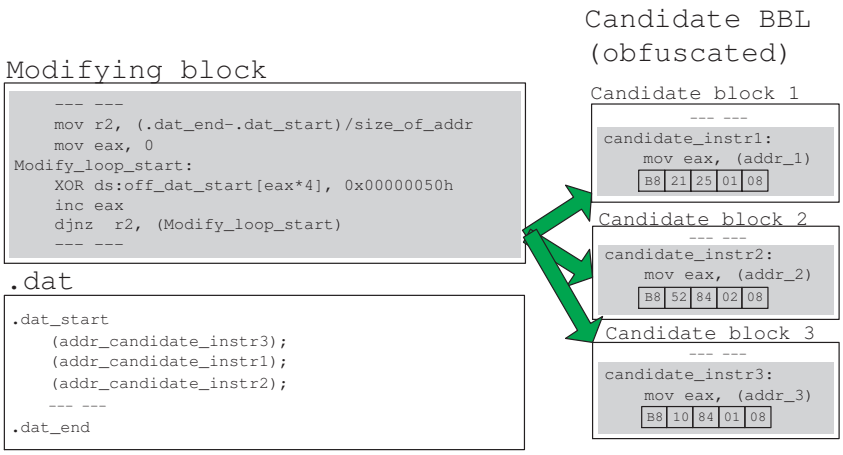

Figure 5.6: One to many obfuscation

to obfuscate function call instruction. So, we insert the "XOR" modifying instructions in modifying block. After the "XOR" instruction is executed, the opcode of obfuscated instruction will become "0xE8h" from "0xB8h". The decision on where to insert the modifying instructions is based on liveness analysis of modifying block, as discussed in chapter 2 and chapter 5.

Then, we patch the address of function call, relatively or absolutely, to operand of the normal instruction with universal patching mechanism. The linkage of the instruction operand and the address of the callee function is added into the global list of universal patching mechanism. At final stage of link operation, the operand will be updated to be the displacement of the address with universal patching mechanism.

Finally, we implement the algorithm based on basic block as described in chapter 4.

We also use one to many technique to restraint storage size overhead. The implementation of using one to many technique on basic block based obfuscation is presented in chapter 4. Figure 5.6 shows the implementation of the technique on function call obfuscation.

In chapter 4, we proposed a self-modifying code algorithm based on basic block. We proposed to mutate the edge connecting two split basic blocks and thus change the

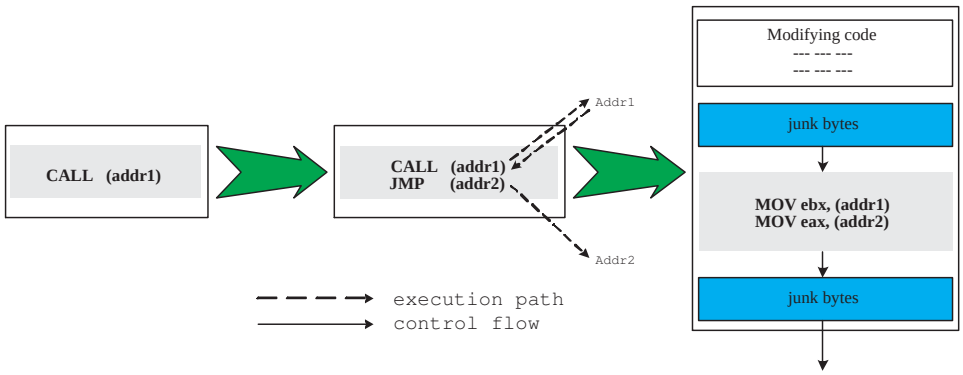

Figure 5.7: Incorporation of two algorithms

control flow of the program completely. The incorporation of the algorithm with function call obfuscation work on basic block is shown in figure 5.7. Usually, single function call instruction construct a basic block. So in the block, both the function call instruction and the inserted control flow instruction are obfuscated to normal instructions. And the unreachable junk bytes immediately inserted before and after the obfuscated basic block is also presented.

5.3.2 Experimental Results

Similar to the evaluation in chapter 3 and chapter 4, the efficiency of the proposal is measured with two metrics, instruction disassembly errors and control flow (edge) disassembly errors[54]. We apply obfuscation the benchmark from *SPECint-2006*, disassemble the obfuscated program with IDA Pro 4.7, and calculate a statistic data comparing the disassembly code and original clear code. Then, we calculate the result in these two metrics with standard evaluation method discussed in signal-based obfuscaton[54].

The experimental result of instruction confusion factor CF_{instr} is listed in table 5.1. Comparing with algorithm based on basic block, the performance of instruction disassembly errors decreases about 2%.

We list the edge disassembly errors in table 5.2. Our algorithm introduces more than 50.16% control flow disassembly errors.

Table 5.1: Instruction disassembly errors

PROGRAM	T_{total} (T1)	T_{disasm} (T2)	Confusion factor ((T1-T2)/T1)
bzip2	209,103	54,333	74.02 %
h264ref	394,954	107,875	72.69 %
hmmer	318,331	87,265	72.59 %
lbm	184,488	47,214	74.41 %
mcf	189,206	47,826	74.72 %
sjeng	232,043	59,691	74.28 %

Table 5.2: Control flow disassembly errors

PROGRAM	CFG_{total} (E1)	$\|CFG_{disasm} - CFG_{total}\|$ (E2)	CFG_{smc} (E3)	Confusion factor ((E2-E3)/E1)
bzip2	21,694	31,188	19,879	52.13 %
h264ref	37,976	55,621	35,569	52.80 %
hmmer	32,417	46,047	29,787	50.16 %
lbm	19,400	27,611	17,520	52.02 %
mcf	19,941	28,418	18,099	51.75 %
sjeng	24,876	35,205	22,324	51.78 %

The speed and program size performances of the obfuscated programs are shown in figure 5.8 and figure 5.9. We can see that program will bloat up to 2.33 times of original size.

We compare the proposed algorithm with the algorithm with Control Flow Flattening[63], listing the result in table 5.3. The basic block based and function level obfuscation introduces 19.38% more instruction disassembly errors, 31.75% improvement on storage size overhead with the cost of 12.8% control flow disassembly errors.

As shown in table 5.4, comparing with the signal based obfuscation[54], the proposed algorithm introduces average 16.49% instruction disassembly errors, 10.59% control flow disassembly errors and 10.04% improvement on storage size overhead.

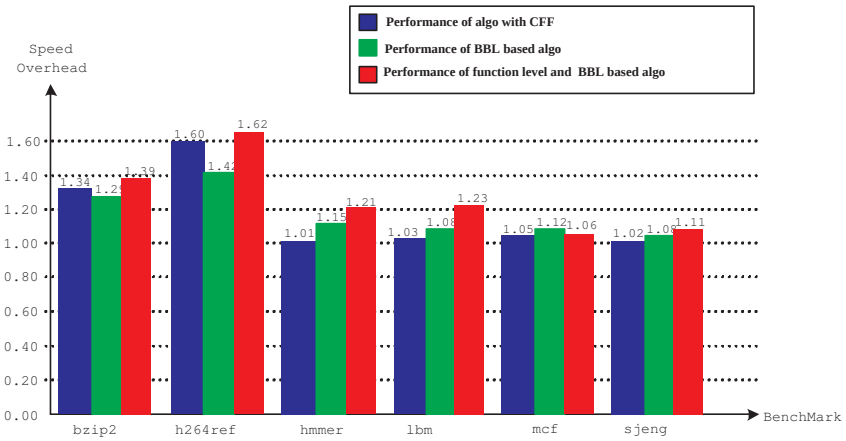

Figure 5.8: Speed Performance of Obfuscation

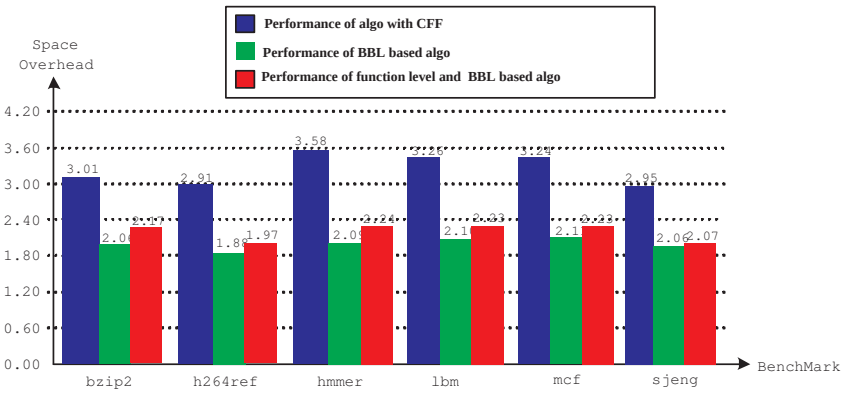

Figure 5.9: Storage Space Performance of Obfuscation

5.4 Security Discussion

The benefit of obfuscation at function level is that the protection where function is called secures the function body indirectly. The sensitive information within a function is secured by the basic block based obfuscation. The control flow is secured by the function

Table 5.3: Algorithm performance comparison

Comparison Item	CFF based obfuscation	BBL based and function level obfuscation	Improvement
Instr disass error	54.39 %	73.77 %	19.38 %
Edge disass error	64.57 %	51.77 %	-12.8 %
Storage size	3.15	2.15	31.75 %

Table 5.4: Algorithm performance comparison

Comparison Item	Signal based obfuscation	BBL based and function level obfuscation	Improvement
Instr disass error	57.28 %	73.77 %	16.49 %
Edge disass error	41.18 %	51.77 %	10.59 %
Storage size	2.39	2.15	10.04 %

level obfuscation and basic block based obfuscation together. We discuss the resistance of the algorithm to static attack and dynamic attack.

5.4.1 Resistance to Static Attack

Static disassembly is to obtain static information by examining the software code, reasoning about possible behaviors without actually executing the program.

Generally, control flow analysis is the basis of static attack, followed by data flow analysis. Data flow analysis is limited to function within basic block without information support from control flow analysis, and so becomes ineffective in static analysis. The goal of control flow analysis is to construct the basic blocks, and edges that express control transfers between blocks. The proposal thwarts control flow analysis mainly in two functions. The first one is to protect basic block from being identified. It is realized partly by the basic block based obfuscation, in which basic block is separated and new basic block containing sensitive information is created. So to restore the basic block becomes the first difficult point to access the control flow of program.

The second one is to confuse the edge connection by self-modifying code. Edge connections include function call and control flow between blocks within function. Function call obfuscation and basic block based obfuscation secure them respectively. The incorporation of two proposals give us the advantage that it provides a seamless protection to edge connection analysis.

Static analysis is classified into two categories, namely flow-sensitive and flow insensitive, in accordance to whether it is either sensitive or insensitive to program control

flow[74]. Since flow-insensitive analysis can not provide sufficient information for purpose, flow-sensitive analysis is preferred in most situations. Our proposals focus on improving program's property in resistance to flow-sensitive analysis. Both the analysis and the experimental results show that static analysis is ineffective to our proposal based on self-modifying code.

The function call obfuscation also enable liveness analysis to help strengthen the protection of program. In order to remove obfuscation, assuming the attacker compiles and links the file disassembled from original program, the liveness analysis will identify code that serve for the obfuscated function call shown in figure 5.7 as dead code and remove them during compile and link time optimization.

5.4.2 Resistance to Dynamic Attack

Dynamic attack is to observe the behavior of program at run time. One technique of dynamic attack is instrumentation. Instrumentation records software execution state at the execution time. Instrumentation differs from profiling in that,

- The goal of the two techniques are different. Profiling is used to find the hot part that influence the performance of program, especially the execution speed. Instrumentation is introduced to monitor the behavior of program, or debug trace in other words.
- They are applied to program at different stages. Profiling is used in program optimization stage, usually at link time. And instrumentation is used on image of program for analysis, test and control of the program behavior.

To instrument the edge connection in program is the basis of dynamic attack. So an effective method to prevent program from dynamic attack is to prevent the program from instrumentation. The proposed algorithms are resistant to dynamic attack comes from the reason that it is hard for the attacker to find the edge if the control flow instructions

are obfuscated to normal instructions, or vice versa, including scenarios in chapter 4 and that in this chapter. Self-modifying code prevents the edge connection from exposing, thus prevent them from being instrumented and from dynamic attack.

5.5 Conclusion

In this chapter, we propose an algorithm at function level. We choose to apply the obfuscation algorithm to caller function after an analysis on the possibility of obfuscating in caller function or in callee function. Function call instruction is obfuscated to normal instruction in more security since the abnormal stack analysis will expose the protection to a static disassembler. The algorithm integrates with algorithm based on basic block in chapter 4 to protect program control flow from being reverse engineered. We implement the two algorithms together to evaluate the total performance. It brings in a little loss of performance. The benefit is that the disassembler will fail to produce a report since it is confused by the obfuscated control flow, and thus confuses the attacker on the precision of the disassembly code. We also discuss that the algorithm is resistant to both static and dynamic attack.

Chapter 6

Conclusions and Future Work

We reviewed the existing obfuscation techniques in chapter 2 and it can be seen that lots of obfuscation techniques had been proposed in the past. Most of these proposals focus on control flow obfuscation since the protection from such techniques is nonreversible in the sense of program execution. Also, control flow analysis is the basis of data flow analysis and so there are double effects to apply obfuscation on control flow. Conventional obfuscation techniques, such as the double-process obfuscation[25] and signal-based obfuscation[54], focus on hiding the real control flow by making the program control flow complex for static analysis. Software reverse engineering technique consists of static and dynamic attacks. Instrumentation is a dynamic analysis technique that records software execution state at execution time. However, such protection is limited since dynamic attacks will reveal the original control flow and the data flow.

The challenge from dynamic attacks, such as instrumentation, will be weakened if program software is protected by self-modifying code. Self-modifying code advances in that the control flow of static analysis could be completely different from that of dynamic execution. Since the control flow is camouflaged with fake control flow under self-modifying code protection, it is time and energy consuming for the attacker to identify the fake control flow and instrument the program appropriately. The protection from self-modifying code is efficient with respect to both static and dynamic attacks.

6.1 Our Contributions

Based on self-modifying code, we proposed three proposals to thwart static and dynamic attacks.

a) Self-Modifying Code Obfuscation Based on Control Flow Flattening

In this algorithm, self-modifying code incorporates Control Flow Flattening technique. Control Flow Flattening has the effect that control flow and data flow are intertwined to increase the disassembly difficulty. This is realized by the dispatcher block and the statement that assigns the offset of address to register "eax" in each flattened blocks. Here, the statement instructions govern the control flow by data flow. We select such statement instructions as candidate instructions for obfuscation. We also implement an improved algorithm that obfuscates single instruction to multiple instructions. On one hand, it introduces more control flow disassembly errors. The experiment result shows that the mistakenly disassembled instruction errors are large. On the other hand, the randomness in obfuscated multiple instructions increases the difficulty for an attacker of restoring them to single instruction.

b) Self-Modifying Code Obfuscation Based on Basic Blocks

Then, we explore the feasibility of applying self-modifying code on the basic blocks without Control Flow Flattening. We introduce more randomness and better performance in control flow protection with a lower cost of storage size. The randomness is introduced by the split of basic blocks at random positions. Experiment results and the comparison with competitive signal-based obfuscation prove that the proposal is effective in defeating the disassembler to re-construct control flow.

c) Self-Modifying Code Obfuscation At Function Level

At function level obfuscation, we propose to protect the function call control flow with self-modifying code in the caller function where the stack analysis and liveness analysis are separate. Thus the stack disorder that originates from obfuscation is not a problem

for static disassembler and static disassembler fails to report the abnormal situation in stack analysis. We propose this algorithm to make exact disassembling and decompiling of function call hard. We then integrate the algorithm with obfuscation based on basic blocks. The advantage is that the disassembler, such as IDA Pro, fails to produce a final report, it keeps on calculating the control flow in most of benchmark suites.

All these algorithms have been implemented and evaluated under the *Linux* environment. We are also working to translate the PLTO platform to the Windows system to evaluate the performance. The translation consists of two parts, to port PLTO to Windows system, and to implement the front end as well as back end. Front end is to read in object files, construct basic block and edge connection and finally to generate control flow graph of program. Back end is to obfuscate the control flow and emit machine code. So far, we have successfully finished the first part and the debugging of front end and back end will be finished soon.

6.2 List of Publications

- Shan Liang, Sabu Emmanuel, "Protection of DRM Agent Codes", In Proceedings of the 2009 IEEE Pacific-Rim Conference On Multimedia, LNCS 5879, pages 743–754, Springer, 2009.
- Shan Liang, Sabu Emmanuel, "Intellectual Property Rights Protection with Self-Modifying Code", in the Proceedings of IEEE Transactions on Information Forensics and Security(TIFS), Submitted.
- Shan Liang, Sabu Emmanuel, "Mobile Agent Protection with Function Level Self-Modifying Code Obfuscation", To be submitted to IEEE Transactions on Information Forensics and Security(TIFS).

6.3 Future Work

We have explored software protection with self-modifying code. The proposed algorithms provide a platform for wide application, such as database source code and data protection[18][27][28][29]. Database reverse engineering is to recover the schema of the database of an application from DMS-DDL text and program source code that use the data in order to understand their exact structure and meaning[30][31][32][55]. We plan to explore how to apply self-modifying code in database obfuscation. The obfuscation applies to both source code and data. Our future work will also focus on the protection of parallel and distributed system applications.

References

[1] http://diablo.elis.ugent.be/.

[2] http://msdn.microsoft.com/en-us/library/xdkz3x12(VS.71).aspx.

[3] http://www.cs.arizona.edu/ debray/binary-obfuscation/.

[4] A.V. Aho, R. Sethi, and J.D. Ullman. Compilers: principles, techniques, and tools. 1986.

[5] B. Anckaert, M. Madou, and K. De Bosschere. A Model for Self-Modifying Code. *Lecture Notes in Computer Science*, 4437:232, 2007.

[6] B. Anckaert, M. Madou, B. De Sutter, B. De Bus, K. De Bosschere, and B. Preneel. Program obfuscation: a quantitative approach. In *Proceedings of the 2007 ACM workshop on Quality of protection*, pages 15–20. ACM New York, NY, USA, 2007.

[7] D. Aucsmith. Tamper resistant software: An implementation. In *Proceedings of the First International Workshop on Information Hiding*, pages 317–333. Springer-Verlag London, UK, 1996.

[8] L. BLUM, M. BLUM, and M. SHUB. A simple unpredictable pseudo-random number generator. *SIAM journal on computing(Print)*, 15(2):364–383, 1986.

[9] N. Borselius. Mobile agent security. *Electronics and Communication Engineering Journal*, 14(5):211–218, 2002.

[10] M. Brzozowski and VN Yarmolik. Obfuscation as Intellectual Rights Protection in VHDL Language. In *Computer Information Systems and Industrial Management Applications, 2007. CISIM'07. 6th International Conference on*, pages 337–340, 2007.

[11] Hongxu Cai, Zhong Shao, and Alexander Vaynberg. Certified self-modifying code. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 66–77, New York, NY, USA, 2007. ACM.

[12] J. Cappaert, N. Kisserli, D. Schellekens, B. Preneel, and K. Arenberg. Self-encrypting code to protect against analysis and tampering. In *1st Benelux Workshop on Information and System Security (WISSec 2006)*, 2006.

[13] W.K. Chen, S. Lerner, R. Chaiken, and D.M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, pages 81–90, 2000.

[14] S. Chow, Y. Gu, H. Johnson, and V.A. Zakharov. An Approach to the Obfuscation of Control-Flow of Sequential Computer Programs. *Proc. 4th. Information Security Conference (ISC 2001)*, pages 144–155, 2001.

[15] C. Collberg, GR Myles, and A. Huntwork. Sandmark-a tool for software protection research. *IEEE Security and Privacy*, 1(4):40–49, 2003.

[16] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. *University of Auckland Technical Report*, 170, 1997, http://researchspace.auckland.ac.nz/bitstream/handle/2292/3491/TR148.pdf.

[17] C. Collberg, C. Thomborson, and D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM New York, NY, USA, 1998.

[18] JL Context Hainaut, V. Englebert, J. Henrard, JM Hick, and D. Roland. Database Reverse Engineering: From Requirements to CARE Tools. *Automated Software Engineering*, 3:9–45, 1996.

[19] L. D'Anna, B. Matt, A. Reisse, T. Van Vleck, S. Schwab, and P. LeBlanc. Self-protecting mobile agents obfuscation report. *Final report, Network Associates Laboratories Report*, pages 03–015, http://www.isso.sparta.com/documents/spma.pdf.

[20] B. De Sutter, B. De Bus, and K. De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):882–945, 2005.

[21] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Link-time binary rewriting techniques for program compaction. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(5):882–945, 9 2005.

[22] S. Drape. Generalising the array split obfuscation. *Information Sciences*, 177(1):202–219, 2007.

[23] L. Ertaul and S. Venkatesh. JHide–A Tool Kit for Code Obfuscation. *Proceedings of IASTED International Conferrence on Software Engineering and Applications*, 2004.

[24] W.M. Farmer, J.D. Guttman, and V. Swarup. Security for mobile agents: Issues and requirements. In *Proceedings of the 19th national information systems security conference*, volume 2, pages 591–597. Citeseer, 1996.

[25] J. Ge, S. Chaudhuri, and A. Tyagi. Control flow based obfuscation. In *Proceedings of the 5th ACM workshop on Digital rights management*, pages 83–92. ACM New York, NY, USA, 2005.

[26] D. Grawrock. The Intel Safer Computing Initiative, 2007, http://www.intel.com/intelpress/toc-secc.pdf.

[27] J.L. Hainaut. Introduction to database reverse engineering. *LIBD Lecture Notes*, 2002.

[28] J.L. Hainaut and J. Henrard. A general meta-model for data-centered application reengineering. In *Proc. of the Dagstuhl Seminar: Interoperability of Reverse Engineering Tools*, 2001.

[29] J.L. Hainaut, J.M. Hick, J. Henrard, D. Roland, and V. Englebert. Knowledge transfer in database reverse engineering: a supportingcase study. In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 194–203, 1997.

[30] J. Henrard, V. Englebert, J.M. Hick, D. Roland, and J.L. Hainaut. Program Understanding in Databases Reverse Engineering. In *Proceedings of the 9th International Conference on Database and Expert Systems Applications*, pages 70–79. Springer-Verlag London, UK, 1998.

[31] J. Henrard and J.L. Hainaut. Data dependency elicitation in database reverse engineering. In *Fifth European Conference on Software Maintenance and Reengineering, March 2001 Lisbon, Portugal*, page 11. IEEE Computer Society, 2001.

[32] J. Henrard, J.L. Hainaut, J.M. Hick, D. Roland, and V. Englebert. Data Structure Extraction in Database Reverse Engineering. In *Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, pages 149–160. Springer-Verlag London, UK, 1999.

[33] G. Hoglund and G. McGraw. *Exploiting Software: How to break code.* Pearson Higher Education, 2004.

[34] IDAPro. DataRescue. In *http://www.datarescue.com/.*

[35] P. Jakubik. Callback implementations in c++. In *The 1997 Conference on Technology of Object-Oriented Languages and Systems, TOOLS 23; Santa Barbara, CA; USA*, pages 377–405. IEEE COMP SOC, Los Alamitos, CA,(USA), 1997.

[36] Hongxia Jin and Ginger Myles. A technique for self-certifying tamper resistant software. In *QoP '07: Proceedings of the 2007 ACM workshop on Quality of protection*, pages 12–14, New York, NY, USA, 2007. ACM.

[37] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Exploiting self-modification mechanism for program protection. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 170–179, 2003.

[38] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th conference on USENIX Security Symposium-Volume 13*, page 18. USENIX Association, 2004.

[39] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[40] D.J. Lie. *Architectural Support For Copy And Tamper-resistant Software*. PhD thesis, Stanford University, 2003.

[41] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM Press New York, NY, USA, 2003.

[42] C. Linn, S. Debray, and J. Kececioglu. Enhancing software tamper-resistance via stealthy address computations. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, 2003.

[43] F. Liu, B. Lu, and X. Luo. A Chaos-Based Robust Software Watermarking. In *Information security practice and experience: second international conference, ISPEC 2006, Hangzhou, China, April 11-14, 2006: proceedings*, page 355. Springer-Verlag New York Inc, 2006.

[44] M. Madou, B. Anckaert, B. De, B. Koen, D. Bosschere, and B. Preneel. On the Effectiveness of Source Code Transformations for Binary Obfuscation. In *Proc. of the International Conference on Software Engineering Research and Practice (SERP06*, 2006.

[45] M. Madou, B. Anckaert, B. De Sutter, and K. De Bosschere. Hybrid static-dynamic attacks against software protection mechanisms. In *Proceedings of the 5th ACM workshop on Digital rights management*, pages 75–82. ACM New York, NY, USA, 2005.

[46] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software Protection Through Dynamic Code Mutation. *Lecture Notes in Computer Science*, 3786:194, 2006.

[47] M. Madou, L. Van Put, and K. De Bosschere. Loco: An interactive code (de) obfuscation tool. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 140–144. ACM New York, NY, USA, 2006.

[48] A. Majumdar and C. Thomborson. Interpreting Opacity in the Context of Information-hiding and Obfuscation in Distributed Systems. In *2006 IEEE Region 10 Conference TENCON 2006*, pages 1–4, 2006.

[49] A. Majumdar and C. Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In *Proceedings of the 29th Australasian Computer Science Conference-Volume 48*, pages 187–196. Australian Computer Society, Inc. Darlinghurst, Australia, Australia, 2006.

[50] H.; Lokan C.; Cornforth D. Marhusin, M.F.; Larkin. Implementation of Program Behavior Anomaly Detection and Protection Using Hook Technology. In *Computational Intelligence and Security, 2008. CIS '08. International Conference on*, 2008.

[51] S. Michiels, K. Verslype, W. Joosen, and B. De Decker. Towards a software architecture for DRM. In *Proceedings of the 5th ACM workshop on Digital rights management*, pages 65–74. ACM New York, NY, USA, 2005.

[52] W. Michiels and P. Gorissen. Mechanism for software tamper resistance: an application of white-box cryptography. In *DRM '07: Proceedings of the 2007 ACM workshop on Digital Rights Management*, pages 82–89, New York, NY, USA, 2007. ACM.

[53] David M. Nicol and Hamed Okhravi. Performance analysis of binary code protection. In *WSC '05: Proceedings of the 37th conference on Winter simulation*, pages 601–610. Winter Simulation Conference, 2005.

[54] I.V. Popov, S.K. Debray, and G.R. Andrews. Binary obfuscation using signals. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium table of contents*. USENIX Association Berkeley, CA, USA, 2007.

[55] W.J. Premerlani and M.R. Blaha. An approach for reverse engineering of relational databases. volume 37. Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA, 1994.

[56] V. Rosset, CV Filippin, and CM Westphall. A DRM architecture to distribute and protect digital contents using digital licenses. In *Telecommunications, 2005. Advanced Industrial Conference on Telecommunications/Service Assurance with Partial and Intermittent Resources Conference/E-Learning on Telecommunications Workshop. AICT/SAPIR/ELETE 2005. Proceedings*, pages 422–427, 2005.

[57] A. Sachan, S. Emmanuel, A. Das, and M. S. Kankanhalli. Privacy preserving multiparty multilevel drm architecture. In *Workshop on Digital Rights Management, 6th IEEE Consumer Communications and Networking Conference*, pages 1–5, 2009.

[58] C. Sashidhar and S.M. Shatz. Design and Implementation Issues for Supporting Callback Procedures in RPC-Based Distributed Software. In *Proceedings of the 21st International Computer Software and Applications Conference*, pages 460–466. IEEE Computer Society Washington, DC, USA, 1997.

[59] Karsten Scheibler. Using self modifying code under Linux. In *http://asm.sourceforge.net/articles/smc.html*.

[60] K. Schelderup and J. Olnes. Mobile agent security-issues and directions. *Lecture Notes in Computer Science*, 1597:155–167, 1999.

[61] B. Schwarz, S. K. Debray, and G. R. Andrews. Disassembly of executable code revisited. *Proc. IEEE 2002 Working Conference on Reverse Engineering (WCRE), pages 45-54*, October 2002.

[62] B. Schwarz, S.K. Debray, GR Andrews, and M. Legendre. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[63] L. Shan and S. Emmanuel. Protection of DRM Agent Codes. In *Proceedings of the 2009 IEEE Pacific-Rim Conference On Multimedia*, pages 743–754. Springer LNCS vol.5879, 2009.

[64] J. Shen, L. Cheng, and X. Fu. Implementation of Program Behavior Anomaly Detection and Protection Using Hook Technology. In *Communications and Mobile Computing, 2009. CMC'09. WRI International Conference on*, volume 3, 2009.

[65] S.W. Smith. Secure coprocessing applications and research issues. *Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory*, 1996.

[66] M. Sosonkin, G. Naumovich, and N. Memon. Obfuscation of design intent in object-oriented applications. In *Proceedings of the 3rd ACM workshop on Digital rights management*, pages 142–153. ACM New York, NY, USA, 2003.

[67] M. Sosonkin, G. Naumovich, and N. Memon. Obfuscation of design intent in object-oriented applications. In *Proceedings of the 3rd ACM workshop on Digital rights management*, pages 142–153. ACM New York, NY, USA, 2003.

[68] David Lie Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 168–177, New York, NY, USA, 2000. ACM.

[69] T. Thomas, S. Emmanuel, A. Das, and M.S. Kankanhalli. Secure multimedia content delivery with multiparty multilevel DRM architecture. In *Proceedings of the 18th international workshop on Network and operating systems support for digital audio and video*, pages 85–90. ACM New York, NY, USA, 2009.

[70] W. Thompson, A. Yasinsac, and J. McDonald. Semantic Encryption Transformation Scheme. In *Proc. of 2004 International Workshop on Security in Parallel and Distributed Systems, San Francisco, CA*. Citeseer, 2004.

[71] S.K. Udupa, S.K. Debray, and M. Madou. Deobfuscation: Reverse Engineering Obfuscated Code. In *Proceedings of the 12th Working Conference on Reverse Engineering*, pages 45–54. IEEE Computer Society Washington, DC, USA, 2005.

[72] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of Software-based Survivability Mechanisms. In *Proc. International Conference of Dependable Systems and Networks*, July 2001.

[73] C. Wang, J. Hill, J. Knight, and J. Davidson. Software Tamper Resistance: Obstructing Static Analysis of Programs. 2000.

[74] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. University of Virginia, Charlottesville, VA, 2000, University of Virginia, Charlottesville, VA.

[75] Chenxi Wang. A Security Architecture for survivability Mechanisms. In *Phd thesis, Department of Computer Science, University of Virginia*, October 2000.

[76] S. Watterson and S. Debray. Goal-Directed Value Profiling. In *Proc. 2001 International Conference on Compiler Construction (CC 2001*, pages 319–333. Springer LNCS, 2001.

[77] K. Wilson. *An introduction to software protection concepts.* Intellectual Property Today, pages 36-41, 2007.

[78] K.S. Wilson and J.D. Sattler. Software control flow watermarking, Aug 2004.

[79] G. Wroblewski. General method of program code obfuscation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, pages 153–159, 2002.

[80] J. Zhang, N.S. Wu, J.G. Luo, and S.Q. Yang. A Scalable Digital Rights Management Framework for Large-Scale Content Distribution. In *Proceeding of International Symposium on Intelligent Signal Processing and Communication Systems, Hongkong*, 2005.

[81] Tao Zhang, Santosh Pande, and Antonio Valverde. Tamper-resistant whole program partitioning. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 209–219, New York, NY, USA, 2003. ACM.