Arab American University – Palestine

Faculty of Graduate Studies

Software Protection Framework based on Code Obfuscation Techniques

By

Ihab "Mohammad Zuhdi" Abed Nassra

Supervisor

Prof. Adwan Yasin

This thesis was submitted in partial fulfillment of the requirements for

the Master's degree in

Computer science

Feb, 2018

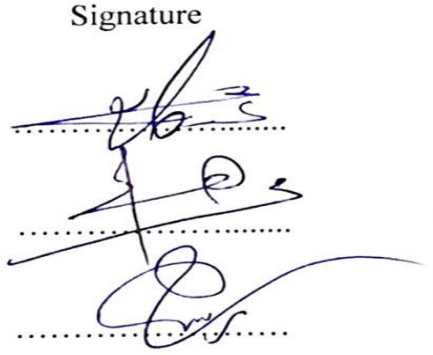
Software Protection Framework based on Code Obfuscation Techniques

By

Ihab "Mohammad Zuhdi" Abed Nassra

This thesis was defended successfully on 24/2/2018 and approved by:

Committee members	Signature
1. Prof. Adwan Yasin (Supervisor)	.....
2. Dr. Mujahed Eleyat (Internal Examiner)	.....
3. Dr. Rushdi Hamamreh (External Examiner)	.....



DECLARATION

The work provided in this thesis, unless otherwise referenced, is the researcher's own work, and has not been submitted elsewhere for any other degree or qualification.

DEDICATION

To whom who have entrusted me to God and taught me life ... Her great heart and generous tenderness

My dear mother

To the partner of life and the leader of the tender

My beloved wife

To those who waited for my success and strengthened my dreams

Dear brothers and sisters and friends

To his great credit

My supervisor Prof. Adwan Yassin

To my distinguished teachers and colleagues

Dedicate this research

ACKNOWLEDGMENT

This thesis would not be possible without the kindly support of my supervisor Prof. Adwan Yasin, who was abundantly helpful and offered invaluable assistance, guidance and support.

I wish to express my gratitude also to the committee members for their effort and time in discussing and reviewing this thesis, where without their assistance and knowledge this research would not be successful.

I wish to express my sincere gratitude to my family whom extremely encourage and support me all the time.

Special respect and appreciation to my colleagues at work for their support. I am also grateful to all people who assisted in achieving my work successfully.

Finally, I extended my thanks to Arab American University, especially faculty of graduate studies and faculty of engineering and information technology for their support.

Abstract

Protecting intellectual property against tampering and reverse analysis is an urgent issue to many software designers, where illegal access to sensitive data is a form of copyright infringement. Software owners apply various protection techniques in order to address this issue. Many of used techniques are weak, since they are vulnerable to both dynamic and static analysis, where the other are very costly since they impose considerable performance penalties. Moreover, these techniques are often not good as they rely on "security through obscurity" which may deter some impatient adversaries, but against a dedicated adversary they offer little to no security. Thus, if an adversary succeeds in extracting and reusing a proprietary algorithm, the consequences will be significant. Moreover, reverse engineering remains a considerable threat to software developers and security experts.

In this thesis, we proposed a software protection framework based on code obfuscation techniques in order to protect software against reverse analysis and unwanted modifications.

First, we presented an obfuscation technique for java programs in order to protect software against static reverse analysis. The proposed technique integrates three levels of obfuscation; source code, data transformation, and bytecode transformation level. By combining these levels, we achieved a high level of code confusion, which makes the understanding or decompiling the obfuscated programs very complex or infeasible.

Second, we proposed an obfuscating technique based on integrating encryption mechanism within recurrent neural network (RNN) in order to enhance the software protection level against dynamic analysis. Neural network provides a robust security characteristic in software protection, due to its ability of representing nonlinear algorithms with a powerful computational capability. The system is designed to enable the neural network generating of different encryptions for the same protected data. This creates a many to one relationship

between the keys and the encryption. In order to complicate the reverse analysis of the software and hindering the Concolic testing attack, we train the neural network to simulate conditional behaviors of a program. Consequently, we replace the critical points of program's data and control flow with a semantically equivalent neural network. Our method is designed to enable the neural network to execute conditional control transfers where the complexity of neural network ensures that the protected behavior is turned to a complicated and Incomprehensible form, making it impossible to extract its rules or locating the accurate inputs which lead to the execution paths behind the network.

Third, we proposed a tamper resistance mechanism based on obfuscation and diversification. The proposed mechanism combined call graph obfuscating, stack obfuscating, diversification, memory layout obfuscating, randomization, and basic blocks reordering in order to thwart tampering and increase the difficulties of static reverse analysis and dynamic stack tracing analysis. A random mapping table is used for mapping the addresses of call and return instructions during the runtime of program. Moreover, a complex call graph of functions is generated to make the obfuscated program harder to attacker analyses and understanding due to a complex dependency of the obfuscated graph. Additionally, a hash mapping table are applied for encoding and decoding of the data stack frames during the runtime of program.

The protection presented by our techniques is immune against static analysis, dynamic analysis, and tampering. Most tampering and revers analysis tools cannot easily undo the obfuscation effects of our techniques, as the attacker will consume a lot of time removing the bugs of the decompiled buggy program. Furthermore, our evaluations confirm that obfuscation effects in our system significantly increase the difficulties in revealing the obfuscated software. On the other hand, the performance evaluation confirms that our techniques protect software efficiently with an acceptable excess in execution time and memory usage.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER ONE
INTRODUCTION

Chapter 1 : Introduction

1.1 Introduction

This chapter presents an overview of the research title, problem statement, significance of the research, research aims and objectives, research questions and research hypotheses. Moreover; the research contribution , research organization, and publications are presented in this chapter.

1.2 Overview

Over the last years, a lot of software and programs have been suffering from copyright violations, as well as these software required a hard work, a lot of time , intelligence, and a lot of money. The costs of software protection against piracy is estimated billions of dollars, where a lot amount of copyright and intellectual property are included and protected within the software.

Software piracy is not the only mechanism of copyright violations, since there are many tools that can provide an access control to software's data and makes it easier for the adversaries and reverse engineers to anlyse the software and steal the intellectual property. (Rasch & Wenzel, 2013). In which, an illegal access could be obtained when the software is compromised. Furthermore, such stealing is difficult to reveal or tracking easily, which increase the challenges of software protection process.

The major problem of software protection is the distribution of software over the client devices, in which the owners lose the control on their software. Over the last years, client devices became more powerful (Gu, et al., 2011), where an attacker with a malicious intent can violate the copyrights and tampering the software via applying many analysis and reverse

engineering tools such as de-compilers, disassemblers, dynamic tracing and dynamic debugging. An illegal access could be obtained when the software is being cracked, where illegal copying and distributing of cracked software is a form of copyright infringement.

Attacking the client software by malicious users is called a white-box attack model, where the attacker has a full access to the software (De Mulder, et al.,2010). Furthermore, the malicious users can run the program, as well as, observe the memory, and change bytes during execution (Bos, et al., 2016).

Many Efforts have been introduced to thwart software analysis and tampering, but most of them are failed due to the prevalence of a healthy software monoculture and the inherently open architecture of current computer systems. The ideal software protection technique is the one that achieve the concept of "one machine, one code" (Khan, et al., 2015). The earlier proposed techniques include: physical tamper-resistant devices such as dongles and cryptographic techniques (Schrittwieser, et al., 2016). Software cryptographic techniques involve running encrypted code while the program instructions are being decrypted on the fly prior to their execution (Gautam & Saini, 2017). Software tamper-resistance such as code obfuscation techniques attempt to make the code more difficult to analyze and understand (Schrittwieser, et al., 2016).

This research aims to develop a framework for protecting software against tampering and reverse engineering analysis by integrating multi-levels of protections with the construction of many to one protection.

1.3 Problem Statement

Many studies have investigated one to one protection, where there is a clear lack of studies that are constructing the many to one protection, in which most of these approaches protect the intellectual property and seen as trade secrets. Therefore, the need for robust software protection techniques against many form of tampering, analysis and other means of exploitation is highly recommended nowadays, in which these techniques should address the lack of trustworthy software in an untrusted environment.

Software protection techniques serve as a binding to glue source codes into one monolithic software, in which without these protections the software become susceptible to attack, analyze and identify.

1.4 Significance of the Research

Software protection becomes more and more crucial and an urgent requirement to many software designers. In this context, we design a software protection framework based on code obfuscation techniques as a main contribution of this thesis. The proposed framework provides a robust security characteristic in software protection against tampering and reverse engineering analysis that attempts to analyze the embedded logic of the obfuscated software routines.

Furthermore, this study will fill a knowledge gap in one of the significant constituents of software security, since it provides an empirical model that can be implemented to protect software against analysis and unwanted modifications. The proposed framework will provide the researchers and practitioners with a new perspective of software protection. Thus, this research will help to improve the security level of software obfuscation without affecting the

performance of obfuscated software. The authors think that an interesting and new approaches can be opened from this research; therefore, researchers can utilize this research as a starting point for further researches.

1.5 Research Aims and Objectives

The first aim of this research is to explore the current state of software protection techniques in order to highlight the major limitations and deficiencies of these techniques.

Second, it aims to improve the level of software protection by integrates many levels of obfuscations and combines different protection techniques in order to complicate the process of reverse engineering analysis and make decompiling of the programs infeasible.

Third, it aims to indicate that employing the neural network with software protection provides a robust security characteristic, due to its ability of representing nonlinear algorithms with powerful computational capability.

Finally, the research aims to develop a model that satisfies all levels of obfuscations and provides a robust protection against many forms of tampering and reverse engineering analysis that attempts to analyze the embedded logic of the obfuscated software routines.

1.6 Research Questions

This researches aims to answer the following research questions:

- What is the best techniques that can be employed to obtain a robust software protection against tampering and reverse engineering analysis?
- To what extent the using of neural network can provide a robust security characteristic in software protection?

- To what extent the construction of many to one protection can protect the intellectual property and seen as trade secrets?
- To what extent the using of call graph and stack obfuscation can deter the tampering of software?

1.7 Research Hypotheses

The study addresses the relationship between the level of obfuscation and the potency of obfuscated software against tampering and reverse engineering analysis. The following hypotheses will be tested to address the research objectives:

- **H1:** Integrating multi-levels of obfuscations have a significant impact in achieving a high level of software protection and making the decompiling of software infeasible.
- **H2:** Introducing the neural network with software protection provides a robust security characteristic in software protection.
- **H3:** The construction of many to one protection protects the intellectual property and seen as trade secrets, as well as increasing the difficulties in revealing the obfuscated software.
- **H4:** Embedding the encryption and decryption functions inside the structure of the neural network increases the potency of the obfuscated software against reverse engineering analysis.
- **H5:** Combining of call graph obfuscating, stack obfuscating, diversification, memory layout obfuscating, randomization and basic blocks reordering thwart call stack tracing and analysis and also deter the tampering of software.

1.8 Research Contributions

The findings of this research are important to researchers since it resolves the main problems that other approaches have been suffering from by introducing the following contributions:

1. Developing a framework that integration multi-levels of obfuscations since depending on one level will not be sufficient to deter reverse engineering from analyzing the software.
2. The proposed framework protects software against static analysis, dynamic analysis, tampering, and call stack tracing and analysis. Therefore, the proposed obfuscator will protect the software at all levels, which consider as advantage over other proposed approaches.
3. Using advanced programming techniques such as compile time reflection and metaprogramming that give us the ability to inspect classes, interferes fields and methods at runtime, which enable us to develop and design encryption/decryption algorithms that can access and modify the obfuscated program during the runtime.
4. Introducing the neural network with software protection in order to provide a robust security characteristic in software protection due to its complexity and powerful computation capability.
5. Embedding the encryption and decryption functions inside the structure of the neural network to increase the potency of the protected software against reverse engineers.
6. The proposed obfuscator embeds the neural network function inside the software instructions, thus the neural network function merged with other program operations which make it harder to be located. Furthermore, it cannot easily separate the

network form the software correctly, because it is embedded in complex dynamic data dependencies.

7. Introducing the neural network to execute the conditional control transfers, where the complexity of neural network ensures that the protected behavior is turned to a complicated and Incomprehensible form, making it impossible to extract its rules or locating the accurate inputs which lead to the execution paths behind the network.
8. The proposed obfuscator makes the software parts depend on each other in order to force the adversary to investigate a larger part of the software to analyze a specific fragment of code.
9. A call graph and stack tracing obfuscator is proposed to protect the software from any tampering, and prevent attacker form detecting the behavior of obfuscated program.

1.9 Research Organization

The rest of this thesis is organized as follows:

Chapter two explains the research design, data sources, data collection strategies, quantitative and qualitative analysis, issues of reliability and validity methodology, and measurements and evaluation metrics.

Chapter there provides a literature review of software protection techniques. First we provide a background of software protections and obfuscations. Second, we explore the threats to software applications. Third, we clarify the software protection techniques against piracy. After that we present the tampering resistance techniques.

In chapter four, we explore the code obfuscation models & techniques that are used against static and dynamic analysis. At the end of this chapter, we provide a summarization and critical discussion.

Chapter five provides a detailed description of the proposed software obfuscation model. Chapter six presented the evaluation and experimental results of the proposed model. Moreover, this chapter presents a comparative evaluation with related models in terms of execution time and memory usage.

Last chapter is about conclusions and recommendations. Moreover, it presents the future works.

1.10 Publications

1. Yasin, A., & Nassra, I. (2016). Dynamic Multi Levels Java Code Obfuscation Technique (DMLJCOT). International Journal of Computer Science and Security (IJCSS), 10(4), 140.
2. Yasin, A., & Nassra, I. (2018). Software Obfuscation Technique based on Recurrent Neural Network". International Journal of Intelligent Systems and Applications (IJISA).
3. Nassra, I., & Yasin, A. (2018). Software Tamper Resistance Mechanism Based on Obfuscation and Diversification. Journal of Computer Security.

CHAPTER TWO
RESEARCH METHODOLOGY

Chapter 2 : Research Methodology

2 Introduction

The aim of this chapter is to present the research methodology that is used in this research. In this chapter we will explore the research design, data sources, data collection strategies, quantitative and qualitative analysis, issues of reliability and validity, and measurements and evaluation metrics.

2.1 Overview

Methodology is the method or style where the researchers follow when they conduct their research. Researchers choose the methodology of their research according to the research nature. Each research has its properties and uniqueness (Christensen et. al., 2011).

2.2 Research Design

Selecting the research design is a very important decision, where it depends on the research problem, objectives and assumptions. In this study, a combined qualitative and quantitative methodologies are used to analyze and evaluate the results that obtained from the experiments and evaluations. This research is an empirical research, where it aims to ensure that the proposed techniques provide a robust security characteristic in software protection. Furthermore, it aims to ensure that the security offered by the proposed techniques have a strong resistance against disassembling and de-compilation tools that attempts to analyze the embedded logic of the obfuscated software routines.

2.3 Data Sources

The main data sources of this research are: literature review, conducting of an experiments, and authors' experiences and skills.

2.4 Data Collection Strategies

Several data collection techniques are used in this research that describe how the research maintained a chain of evidence. First a general literature research is set off, and then contributes to a more focused literature review is conducted which contributed in conducting the experiments and analyzing the results. The following sections will describe the data collection strategies that are employed.

2.4.1 Literature Review Strategy

A continuous literature review is conducted through this research based on publications, articles and E-books. First, a review of software protection techniques is conducted, then the authors focus their review in code obfuscation techniques. At an early stage, the information gathered is used to study the shortages and limitations of the current software protection techniques. After that, a concentrated literature search is conducted in order to carry out the experiments, and compare the obtained results with others. The Information that gathered through the literature review and the experiments is used to develop the proposed code obfuscation techniques.

2.4.2 Experimental Results and Experience Strategy

The authors experience about the software security and their skills in programming, especially programming with Java and .NET Languages is used to describe and interpret the results of experiments. In addition, authors experience and the experiential results assistant in developing the software protection techniques.

2.5 Quantitative and Qualitative Analysis

The authors use both quantitative and qualitative analytical techniques as a mixed data analysis, where these techniques are used sequentially at different times. For instance, initial qualitative data might be interpreted, analyzed, and used to inform a quantitative phase of the study, after which quantitative data are analyzed.

This study followed a sequential and concurrent experimentally strategy. The steps toward finding the research results are: determining the level of protection, conducting the experiments; evaluating the proposed techniques, conducting a qualitative comparison, and conducting a quantitative comparison with other related techniques. Furthermore, the researchers evaluate the proposed techniques at each level of software protection.

2.6 Issues of Reliability and Validity

Credibility of any research is relying on the validity of their finding, not only on the reliability of their data. Therefore, this research systematically and consistently defines the subject of the study; and also measures and identifies the trueness of the data sources that relevant to the study subject. The authors are adopted a significant measure in order to avoid this study from any bias either when evaluating the proposed techniques or when selecting the test sample.

CHAPTER THREE
LITERATURE REVIEW

Chapter 3 : Literature Review

3 Literature Review

This Chapter aims to discuss the research conceptual framework and the previous literature studies in software protection. It firstly reviews the definitions and concepts related to software security and protection. Secondly, it discusses the threats to software applications. Thirdly it presents an overview of software protection techniques. Fourthly, it presents an overview of code obfuscation techniques. Fifthly, it reviews the software piracy protection schemes. Sixthly, it reviews the schemes that have been proposed to thwart software analysis. Seventhly, it reviews the tamper resistance schemes.

3.1 Background

Software protection plays a significant role in protecting copyright and intellectual property that is embedded within the software (Sasirekha, et al., 2012). Hence, the issue of software security is the major challenge that disturbed software designers for many years and still continue doing so, where the attackers struggle every new technique via adapting their methods. The potential threats of piracy, tampering and reverse engineering analysis become a matter of prime concern.

The term software protection means protect the software against piracy, tampering, analysis, unauthorized use, and other ways of exploitation (Hosseinzadeh, et al., 2016). It aims to address the lack of trustworthy software in an untrusted client environment. Software protection falls between the gaps of security, cryptography, and engineering.

Several mechanisms are applied to protect software property. As instance, watermarking can be used to protect ownership via embedding the copyright into the software (Hamilton &

Danicic, 2011). On the other hand, software fingerprinting considers as a relevant mechanism, in which it can facilitates the tracking of copyright infringers via embedding a message into each copy of the software (Chroni & Nikolopoulos, 2013). Code obfuscation seems to be a promising one of them, where it is an attempt to transform the application to an equivalent one which is harder to analysis by reverse engineers and difficult to understand by human (Hosseinzadeh, et al., 2016). However, most of the available code obfuscation techniques just parsing the source code according to the compiler's language lexical and syntax rules (Popa, 2011). Therefore, it is easily for the De-compilation tools to de-compiling the software back to the source code. Tamper resistance according to Junod et al., (2015) is a technique that is aimed to make the program unmodifiable. As well as, tamper resistance and code obfuscation can be used to reinforce other mechanisms.

On the other hand, advances in reverse engineering mechanisms with the help of dynamic code analysis make the software tampering and analysis more powerful (Moser, et al., 2007). Furthermore, it helps the attacker to track the software execution and monitor its information along the trail. According to Sivadasan et al., (2009), reverse engineering can be defined as the process of analyzing and extracting the proprietary structure elements from the software.

The main goal of reverse engineering analysis is to search for security breaches or loopholes in the software, either to steal the embedded logic or algorithm behind the functionality of the software (Yasin & Nasra, 2016).

There are several tools either commercial or free ones that can be used to perform software reverse engineering process. These tools are generally classified as de-compilers, de-obfuscators, disassembler, debuggers, hex editors, un-packers, and program executables (PE) editors (Amankwah, et al., 2017).

De-compiler is a technique that used to retrieve the source code of a software from its machine code or intermediate bytecode. (Khan, et al., 2015; Buzatu, 2012). However, if a de-compiler fails to retrieve the source code, it produces its equivalent assembly code. De-compilers which can retrieve a better readable source code for the binaries files are considered as reality de-compilers. On the other hand, De-obfuscators is a relevant technique that is design to reverse or remove the obfuscation effects that are applied on source code as an attempt to regenerate the original source code (Uppal, et al., 2014). Furthermore, it can operate on either bytecode files or binary files. A disassembler is a technique that generates an assembly language code from executable or binary code, while the debuggers work as disassemblers with the ability of providing a view of the registers and stacks current state. Moreover, advanced debuggers allow to illustrate the runtime state of the software by setting breakpoints into the assembly code in order to help adversary in editing the software (Amankwah, et al., 2017). Disassemblers and debuggers can be used to unpacking software, decoding password, revealing software structure, and identifying faults in a program.

Hex editors is also another technique that can be used to edit and view the binary files in hexadecimal format. Hence, the adversary can easily edit instructions of a given executable file using a basic hex editor. Some hex editors provide file comparison utility that can be used to search for specific instructions that are need to be modified. However, if search facility in a hex editor is not available, the adversary can use a disassembler or debugger in order to locate the wanted instruction position in the binary file. Furthermore, some advanced hex editors are able to edit the memory, carry out hash calculations, and manipulate logical and physical drives (Sasirekha, et al., 2012).

The unpacker is a tool that can be used to convert a packed file into its original source code, where the packed file is a file that is compressed to occupied a low storage region. Additionally, it can be used to reverse commercial protection schemes via removing the obfuscation affections that are applied on it. On the other hand, program executables (PE) editors can be used to extract the binary files' headers in order to change or remove any hidden secret code (Khan, et al., 2015). Whereas the programs that are designed to modify themselves in the memory can be debugged using memory dumpers.

3.2 Threats to Software Applications

Software have been suffering from three major threats: piracy, reverse engineering analysis, and tampering. Piracy concerns unauthorized copy and use of software, reverse engineering analysis involved techniques to inspect the internal structure of software, while tampering represents techniques to tamper the software. Tampering attacks aim to modify the functionality of the software while reverse engineering techniques attempts to analyze the embedded logic of the software.

In this thesis, we don't focus on piracy prevention. We focused our work to protect software against reverse engineering analysis and tampering attacks, because piracy protection relies on the same techniques that are used to protect software against tampering and analysis. The following sections elaborate the software threads in details.

3.2.1 Piracy

It involves unauthorized use or copying of software instances either by individuals for use for themselves or by companies whom then sell the illegal copies to users (Kulkarni & Lodha,

2012). Moreover, crackers may pirate a software to steal its features and include these features in their own products.

Software piracy take several forms as follows (Gomes, et al., 2015):

- **Softlifting**: this form of piracy is considered as the most common type, in which someone purchasing a single licensed copy, then installing it on other colleagues' computers in a violation of licensing terms.
- **Client-server overuse**: Violate the number of copies that are licenses for.
- **Hard-disk loading**: typically involves installing an unauthorized copy of software onto a computer being sold to buyer. This makes the deal more attractive to the buyer, and without any additional costs to the dealer. The dealer commonly doesn't provide the buyer with the original disks or manuals. An example of this form is the piracy of operating systems as Windows.
- **Counterfeiting**: this form of piracy involves generating fake copies of a software, making it look authentic. The dealer provides manuals and dikes to the buyer in order to make the product looks as much the original product. The copies of software are made using a CD-burner.
- **Online piracy**: it involves downloading pirated or illegal software from the Internet, auction or blog, or peer-peer network. Currently, there are thousands of websites that providing unlimited downloads of pirated software to any user.

3.2.2 Reverse Engineering Analysis

As mentioned previously that the reverse engineering analysis can be used to inspect the inner workings of software, where it can extract the secret keys, hidden

algorithms, and other information embedded in the software. Moreover, it can be applied on non-executable code, assembly code, and executable code. The reverse engineering process is shown in figure 3.1.

This type of attack take two forms as follows:

- **Static analysis**: These techniques is applied on static code or non-executable code. In involves two stages: disassembling and debugging (Cappaert, 2012). Disassembling is usually preformed using either recursive traversal or linear sweep. Linear sweep scans the software's code, then disassembling its instructions one by one, assuming that every instruction is followed by another instruction (Debray, et al., 2010). On the other hand, recursive traversal derives and disassembles the control flow. De-compilation step could return source code from low-level code. In some programming languages such as Java or .NET, it is easy to decompile bytecode to source code.
- **Dynamic analysis**: these techniques is implemented on executable code. In which it traces the executed instructions, data values, and register contents. This form of analysis has more powerful than a static analysis; however, it requires more analyzing time and more complex work (Canfora, et al., 2011). Furthermore, it requires a platform similar to the target code's platform. In some cases, a program may be equipped with anti-debugging techniques which may inhibit the dynamic analysis process

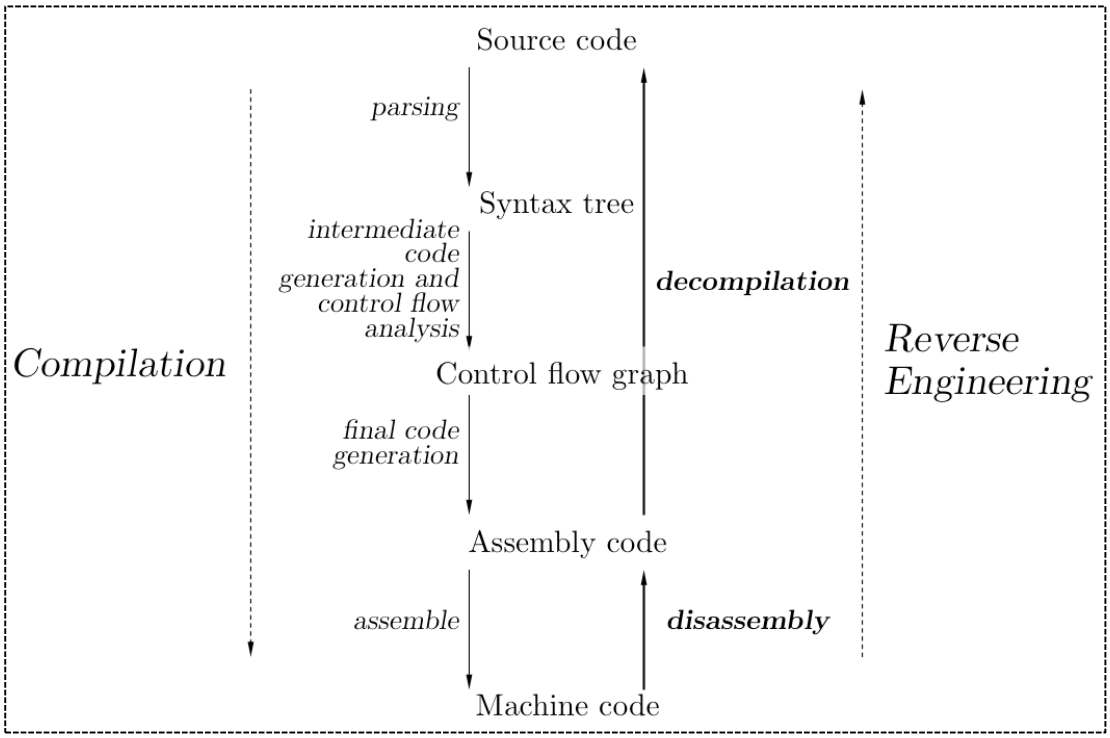


Figure 3.1. *Reverse engineering analysis stages (Cappaert, 2012)*

3.2.3 Tampering

Tampering attacks typically analyses the binary file. The adversary in such attack needs information about the program internals before he can tamper the software successfully (Uppal, et al., 2014). Therefore, tampering attacks usually preceded by applying several reverse engineering techniques. Tampering techniques can be classified as follows (Cappaert, 2012):

- **Static tampering techniques**: these techniques modify a static binary file such image. It assumed that the code is not loaded into memory and modified there. Furthermore, downloading a crack and applying it to open and read binary file is also called a static tampering attack.

- **Dynamic tampering techniques**: these techniques alter the software at runtime. First, debuggers load the code into the memory. After that it traced the software instructions one by one, which enable the adversely to monitor and modify the code of the loaded program. A dynamic tampering attack is commonly implemented by hand similar to software debugging attack.

3.3 Protection Techniques Against Piracy

Over the last years, many protection techniques are proposed to battle software piracy such as watermarking and fingerprinting. Software watermarking is a property or value that is embedded into the software in order to prove ownership (Imran, et al., 2015). The owner can extract this hidden message from the software to obtain an evidence of piracy. Watermarks can be categorized into static and dynamic, where the static watermarking techniques work by embedding a watermark into the program's code, while the dynamic watermarking works by embedding a watermark into the program's execution state (Hamilton & Danicic, 2011). Software fingerprinting is another technique that embeds a unique identifier into each instance of the software which belongs to a specific end user or company (Masoumi, et al., 2014). Attackers can apply code transformations to break or remove the software watermark and fingerprinting messages.

Collberg proposed the first dynamic watermarking scheme in 1999, he suggested that watermarking techniques should be difficult to discover and resilience against the crackers whom trying to remove the watermarks (Collberge & Thomborson, 2002). Many studies are conducted to protect software against piracy. As instance, Shi et al. (2010) enhanced the watermarking protection technique that was proposed by Monden et al. (2000). The idea is

to insert a dummy method into a java program, in which the watermark bits are embedded through altering the operands or encoding the instructions' opcodes into the dummy method. On similar study, Kapi and Ibrahim (2011) enhanced the dummy method algorithm by adding an encoding scheme that generates a fixed size dummy method. Chan et al. (2013), proposed a software piracy technique using a software birthmark to detect any tampering of JavaScript programs code. A birthmark is a unique characteristic that can be used to identify the program. The proposed technique is based on heap graph, where the birthmark is formed via extracting objects from the heap and constructing a heap graph. On similar study, Patel et al. (2014) presented a dynamic birthmark system for JavaScript programs, where frequent subgraph mining and agglomerative clustering are used. Tian et al. (2015) presented a new type of software birthmark that is based on dynamic key instruction sequences. Chen et al. (2017) presented a software watermarking approach for Java application. The proposed scheme divides the watermark bits into pieces according to the number of the method names; in which, each piece is encoded with a method name in order to embed the watermark into the program.

3.4 Protection Techniques Against Reverse Engineering Analysis

This section presents a number of techniques that are used to protect software against reverse engineering analysis. Most of these techniques aims to obscure the inner routine of the software to protect it against analysis.

Over the last years, number of software protection techniques are proposed. Commercial and freely available protection techniques are often not good as they rely on "security through obscurity" (Yasin & Nasra, 2016), which include renaming variables, string literals and adding

nonsense instructions. These techniques may deter some impatient adversaries, but against a dedicated adversary they offer little to no security. There are two main forms of reverse engineering: static and dynamic analysis.

Static analysis is a broad term that refers to analyses the software without actually executing it. Therefore, static analysis techniques look at programs in a non-runtime environment. In the past, these techniques were required a source code in order to analyses the software, however obtained the source code is unpractical and sometimes is unavailable. Consequently, static analysis tools nowadays are worked by assessed the binary code which called bytecode or compiled code instead of source code (Schrittwieser, et al., 2016). These techniques enable the attacker to analyses the software effectively and comprehensively.

In dynamic analysis, attacker runs the software using a set of inputs while monitoring and tracing the generated software's output. Moreover, attacker can profile the software in order to detect the actual paths chosen for program execution (Kulkarni, 2012). Furthermore, dynamic analysis provides the attacker with a considerable power in locating the secret information such as secret keys that are embedded into the software. Dynamic analysis is more difficult than static analysis since it requires to run the software on different inputs.

A reverse engineer usually begins by inspecting the software using disassembling tools, and then finding patterns, composing software parts as an attempt to understanding it, bit by bit. First a binary file is disassembled, then, the reverse engineer decompiles the disassembled code into the original source code. Finally, the source code will be obtained (Cappaert, 2012). These steps are the main steps of static reverse engineering, while the dynamic reverse engineering involves monitoring and tracking the execution of the program as an a tempt to analysis the behavior of the program.

According to many researchers (Frederiksen & Courtney,2011; Cappaert, 2012; Kulkarni, 2012; Scrinzi,2015; Sebastian, et al., 2016; Schrittwieser, et al., 2016); code encryption, white box cryptography, Self-modifying code, and code obfuscation are presented as the main techniques used to thwart the reverse engineering analysis.

3.4.1 Code Encryption

Code encryption encrypts the code of software to prevent the adversary from gaining access and then analyzing the source code. This technique protects software against static reverse engineering and static tampering. During the runtime of the software the encrypted parts of code will be decrypted using a secret key (Braga & Dahab, 2016).

The original software code is encrypted in an encrypted executable file, while a decryption routine is added to the original software. Hence, code encryption is a form of self-modifying code (Mavrogiannopoulos, et al., 2011). Actually, the entire software is treated as a data, where the decryption routine remains code.

Encrypted and polymorphic viruses implement code encryption techniques. Hence, the encrypted virus is encrypted at every new generation of virus body using a unique key to avoid detecting by anti-virus engines (Sharma & Sahay, 2014). Moreover, the decryption routine is added to verify that the virus body is decrypted during the execution time of the program. However, even if the encryption routine is unchanged, the encrypted viruses is evolving and inserting a mutation engine in order to ensure that the changes of the decryption routine are the changes for each new generation of the virus body, which called polymorphic viruses (Natani & Vidyarthi, 2014; Radkani, et al., 2017). Furthermore, when a virus is decrypted

and stored in the memory, a new key is being selected to encrypt the new variant of virus and adds a modified decryption routine (Khalilian, et al., 2016).

Advantages and Disadvantages

The major advantage of code encryption is the low cost and the flexibility of this technique. The most important advantages of code encryption are:

- The secret key is spread over the entire program which force the adversary to analyze the complete software code.
- Attacker should extract the encryption key and feeds it into a decryption routine in order to decrypt the source code of the software, which is a hard task if the decryption routine is very complicated.

However, code encryption has been suffering from many weaknesses which are:

- If the decryption routine is easy to analyze, the adversary can break the decryption routine, and decrypts the software.
- Furthermore, the de-assemblers and de-compilers help the reverse engineers to analyze the software code and returned the source code without any encryption.
- Another weakness of this technique, is the disclosure of the code and data in the memory which can be intercepted, decompiled and debugged. Even if the code remains encrypted, the adversary can still monitor what happens during the execution if bits in data or the encrypted code are being flipped. This technique is also known as fault analysis (Lazar, et al., 2014).

State of the Art

Cappaert et al. (2008) proposed a partial encryption scheme based on a code encryption. The proposed scheme divides the binary code into small segments and encrypted them in order to utilize the partial encryption approach. The encrypted binary code is being decrypted during the execution of program. Therefore, only the essential parts of the code are decrypted during the execution time without the need to decrypt the whole code at once. However, a segment of code may invoke by more than two preceding segments which consider as a major problem of this scheme.

Jung et al. (2008) proposed a software encryption scheme based on key chain. The proposed scheme uses a fixed size segments of code rather than a variable size segment. The idea is to encrypt the basic blocks that are partitioning via control operations, such as branch and jump commands in assembly code. The proposed scheme tries to solve the problem of Cappaert's scheme by duplicating the invoked block of code when the block is invoked by more than two preceding blocks.

Wu et al. (2010), proposed a scheme which replaced the original code with an encrypted code based on a specified instruction distribution. Hence, the encrypted code mimics other code statically, while during the execution time, the original code is reconstructed. The idea is to implement a block cipher via key chaining to be inadequate for encrypting the executable code using a duplication and block transformation. In spite of the proposed technique can hide the keys in blocks and make the keys varying from block to block. However, it is suffered from the disclosure of the original code in the memory during the runtime of the program, therefore the de-assembler's tools can easily retrieve the source code and traced its execution.

The study conducted by Sasirekha and Hemalatha. (2012), presented a software encryption approach based on index table. The proposed approach employs other encryption techniques such as quasi group encryption, number theoretic transformation, and hadamard transformation in order to encrypt the data of the index table. The main encryption technique that is implemented in this scheme is the quasi encryption technique. This encryption technique is used for scrambling the data to maximize the entropy at the output. Despite of the proposed scheme may complicate the task of attacker since it increases the confusion along the data. However, it is not sufficient in diffusion and confusion the tools of cipher text statistical analysis. This drawback is considered as a big limitation of this scheme.

Protsenko et al. (2015), proposed a dynamic self-protection and Tamper proofing approach for Android applications based on native code. The software consists of three dynamic encryption techniques which called re-encryption, tamper-proofing, and dynamic code loading. The idea is to protect Android applications based on bytecode manipulation and inspection implemented from the native code. The proposed technique applied the bytecode tamper proofing, self-protection, and dynamic code loading within the encryption process in order to protect the code of the original application from static analysis. Although the proposed approach may complicate the process of reverse engineering, but it is suffered from a high execution time and larger program size. Furthermore, the proposed technique is not compatible with the applications that used Java reflection techniques.

In similar study, Kim et al. (2016), proposed a technique that protects the Android applications against static reverse engineering. The proposed technique is used to encrypt and decrypt multiple DEX files inside the APK files and loads them dynamically. When the application is launched, the encrypted DEX files are decrypted and loaded dynamically.

Despite of the proposed technique can protect multi DEX APK files against static reverse engineering. However, when the application is launch all files will be decrypted and loaded normally without any encryption, which enable the adversary to obtained the whole files without any encryptions.

3.4.2 White-Box Cryptography

In the late 90s, a new threat in security that attempts to extract the key information out of DES and RSA implementations was presented (Cappaert, 2012). These type of attacks focus their work in extracting the secret key which are embedded into the cryptographic implementation. In similar study, Chow et al. (2002) described a new threat model called white-box attack that aimed to extract the cryptographic key to obtained a full access to the implementation of DES and RSA algorithms. According to Chow, obfuscating the cryptographic only is not sufficient to protect against this threat since a parts of the secret key is stored in the malicious host's memory that can be readable by an attacker. On the other hand, Chow et al. presented a new mechanism called white box cryptography that is used to protect cryptographic algorithms against white-box attacks. This technique transforms a cipher with a fixed key to a chain of lookup tables in order to protect the secret key from extraction. Furthermore, white box cryptography transforms the code of program and make it harder to analyze and extract the secret key from it. Cryptographic algorithm with a fixed key could be transformed into a chain of lookup tables via applying a partial evaluation. Hence, this technique guarantees that the information of the secret key is included in the generating lookup tables. Furthermore, these information is spread over the lookup tables via adding mixing bijections transformation which maximizes the dependence of the output

regarding the input (Yang, 2013). This means that a small change in input, even one bit, causes a maximum change in output. The general idea is to spread the information of the secret key over the whole implementation which force the adversary to understand a huge part of the implementation code. The current available techniques are applicable only to XOR functions, permutations, and cryptographic algorithms that are constructed with the lookup tables. This limitation may not consider aa a big drawback since most symmetric cryptographic algorithms such as AES and DES are constructed with these functions (Chow, et al., 2002).

Advantages and Disadvantages

The most important advantages of white-box transformations can be summarized as follows (Michiels, 2010; Cappaert, 2012):

- The embedded secret key information is spread over the whole implementation of the cipher, which force the adversary to understand and analyze the complete implementation code of white box.
- Multiple diverse instances of one software can be created due the randomness that are inserted into the implementation code of the software.
- The white box cipher can be used as a public key, while its embedded secret kay can be used as private key. Attackers should invert the lookup tables of the entire white box implementation one by one, or they should extract the secret key and includes it into the decryption routine.

In spite of the advantages of white box cryptography, it still suffering from serious disadvantages and limitations mainly in performance (Cappaert, 2012; Avoine, et al., 2017):

- Implementing the white box cryptography reduces the overall performance of the software, due to the significant increasing in execution time.
- Transforming the cryptographic algorithms to a chain of lookup tables cases a significant increase in code size.
- The security of white box techniques is still debatable as an efficient way against adversaries.

State of the Art

Researches in white box cryptography begin in 2002 with the research of a white box DES implementation that is conducted by Chow et al. (2002, November), then this research is followed by the white box AES implementation at the same year (Chow, et al., 2002, August).

White box DES implementation was attacked by Jacob et al via performing a differential cryptographic attack. (2002). On the other hand, performance improvements were proposed by Wyseur and Preneel (2005) and Link and Neumann (2005). In their studies, they improved the work of Chow via presented a modified white-box DES cryptography approach which is more resilient to both the statistical bucketing attack that described by Chow and the differential cryptographic attack that described by Jacob. Furthermore, they proposed a target function that requires fewer encryptions than the original one and needs to access only the encryption function of white-box DES. The proposed technique improves the performance of the white-box DES and provides more resilient against Jacob's attack. However, it is still slower than the typical DES encryption and requires more space and time.

On the other hand, the implementation of white box AES has been broken by Billet et al. (2004). Bringer et al. (2006) presented a modified implementation of white-box AES, in

which the S-boxes become part of the embedded secret key. This scheme had been cryptanalyzed via De Mulder et al. (2010).

Karroumi (2010), proposed a new version of white box AES based on the work of Chow. The proposed model aimed to provide a better resistance against the attack of Billet et al. It enhanced the resistance of the white box AES implementation via modifying the algebraic structure of each AES round and applying a different method that is worked with the structure of the AES building block. Furthermore, elements of the states and the sub keys are transformed to fit the modified structure of each AES round. Although, the proposed model may increase the security of white box implementation, but the location and the structure of each AES round can be easily recovered from its binary file.

The study conducted by Lepoint et al. (2013), described a new attack which exploits the collisions that can be occurred on internal variables of the white box implementation. The Results show that the implementations of Chow et al. and Karroumi are vulnerable to the BGE attack.

Cho et al. (2015), analyzed and defined the practical requirements that should be applied in order to provide a better resistance against white box attack. Their study proposed a secure and effective cryptographic constructions that combined the white box cryptography with primitive and standard block cipher. Furthermore, the proposed design transforms the existing secured cryptographic libraries in the black box model to secured cryptographic libraries in white box model. The main disadvantage of the proposed design is the requirement of providing a detailed information on how the white box implementation can be transformed and constructed. Furthermore, a knowledge about the round transitions and

the location of the S-boxes might be required with the applied encodings format to be included into the look up tables.

Sasdrich et al. (2016) proposed a white-box AES implementation based on reconfigurable hardware. Their work show that the white box implementation can be mapped to existing reconfigurable hardware architectures. The goal of using the reconfigurable hardware devices is to provide sufficient amounts of resources to cope with the massive memory requirements of white box implementations. For this hardware implementation they examined the vulnerabilities against side channel attack (SCA), differential computation analysis (DCA) and Differential power analysis (DPA). The results show that the secrets in hardware implementations can be reveal when performing a SCA, DCA and DPA attacks under gray-box settings. The proposed study explained and verified the reason behind the success of such attack via providing a better understanding of the mathematical foundations of those attacks in order to improve the future implementations of white-box.

Bos et al. (2017), conducted a study to assess the security of white box implementations. Their study introduced the DCA attack and differential fault analysis (DFA) attack in order to examine the resistance of software against such types of attacks after being encrypted using the white box cryptography. Their results show how DCA and DFA attacks without administration privileges can extract the embedded secret key from many public non-commercial white box implementations of standardized cryptographic algorithms.

3.4.3 Self-modifying Code

Self-modifying code is a technique that generates or modifies codes during the running time of a program (Xianya, et al., 2015). The nature of self-modifying code is depending on

the stored program architecture, which means that both the data and code are stored in the same memory space (Ghosh, et al., 2013). Consequently, some program's instructions can be modified and read as data via other instructions. There are two types of modifications, one is to generate new instructions via rewriting the memory address itself; while the other is to modify the existing instructions via rewriting the contents of the memory address.

Self-modifying code mechanisms can protect software against static analysis since it can hide the internal information of the software; however, it can't cope with dynamic analysis properly. Furthermore, it has the characteristics of easy implementation, high productivity and low overheads (Xianya, et al., 2015). According to Mavrogiannopoulos et al., (2011) and Xianya et al., (2015) self-modification techniques can be classified based on the adversary tools capabilities as follows:

- a disassembler.
- a debugger that can't handle the self-modifying code.
- a debugger that can handle the self-modifying code.
- specialized tools.

Many commercial protection software uses self-modifying code techniques to protect program against piracy. On the other hand, some malwares use the self-modifying code to avoid detection by antivirus software.

Advantages and Disadvantages

The major important advantage of self-modifying code is its robust protection intensity against static analysis, in which the adversary spends a lot of time and resources to break this protection. However, it can't cope properly with the dynamic analysis. Furthermore, better protection will cause more performance overheads. Other major problems of this technique

is huge additional of software size, high execution time, and complexity of its implementation.

State of the art

Cappaert et al., (2006) proposed a self-modifying code technique based on dynamic code mutation. The proposed technique used functions as encrypting units. The idea is to calculate the calling function's hashing value in order to protect the called function. When a function is about to be implemented, it is decrypted within its calling function's hashing value, then the function will be re encrypted again after the execution is completed.

Kanzaki et al., (2006) proposed a self-modifying protecting mechanism which replaces the instructions of the protected program automatically. In another study, the authors improve their previous works via implementing their instruction replacement mechanism on source code level. Their new mechanism generates a fake copy of the original program's code, then the variances among the assembly code of the two copies are compared. The proposed mechanisms may prevent attackers from normal dynamic analysis, however, all protected instructions can be easily detected by the attackers at runtime of program (Kanzaki, et al., 2008).

In another study of Kanzaki et al., (2010), they proposed a self-checksum technique based on time sensitive code. The proposed technique checks whether the software is under dynamic debugging or not via adding the sequences of specific instructions that calculates the execution circles of processer inside the software. Furthermore, the proposed technique checks the running time before restoring the instructions, if the running time is in the predetermined range, the instructions are restored correctly, otherwise they will be modified

into fake ones. This technique may prevent adversaries from normal dynamic debugging. However, all protecting instructions such as MOV instructions can be easily detected by adversaries. Furthermore, it takes a lot of time to implement the instructions when they are under debugging.

Anckaert et al., (2009) proposed a self-modifying technique to hide the actual data locations during the program's runtime. The proposed technique applies pointer scrambling, mitigating variables from stack to heap, and periodic reordering of the heap. Their technique supposes a trusted software-based memory management unit that spreads in the memory pages during the execution time. The main trait of this technique is its ability to transfer the whole protection mechanism to other platform, because they require to re programming the part of virtual machine only not the whole protection mechanism. However, this technique has been suffering from the large cost of overheads in both space and execution time because the cost of implementing the protection method of virtual machine is much higher than other methods. In similar studies, Ghosh described how to protect methods based on virtual machine (Ghosh, 2013; Ghosh, 2010), and also conducted some researches on cracking the protection mechanism of the virtual machine (Ghosh, 2012).

Das (2014), proposed a preventive design approach to hide the proprietary code part via adding a self-modifying code at binary-level. The proposed approach divides the original program in two main programs which are server program and client program, in which the two separated programs communicate with each other through a shared memory. The user can interact with only the server program while the proprietary code part will be in the client program. The proposed technique prevents the client program from being executed under the debugger, therefore the obfuscated proprietary code cannot be debugged dynamically.

However, this technique can't support parallel processing where data or code are being shared among multiple threads.

3.5 Tampering Resistance Techniques

Tampering resistant techniques requires high programming skills to embed a "booby traps" into a binary or source code level in order to detect any tampering with the program. Tampering resistance can be defied formally as preventing any tampering of a software via detecting undesired modifications, and reacting in case of tampering (Cappaert, 2012). Reacting is important to return the crashed programs after detecting modification attempts.

3.5.1 Software Guards

Software guards can verify the program code based on a complex, nested network. Tampering a software requires attacking the whole guard network. This means, localizing, identifying, and eliminating the complete guard network and then tampering with the actual software code itself. A guard's graph and its placement in a control flow graph is shown in Figure 3.2.

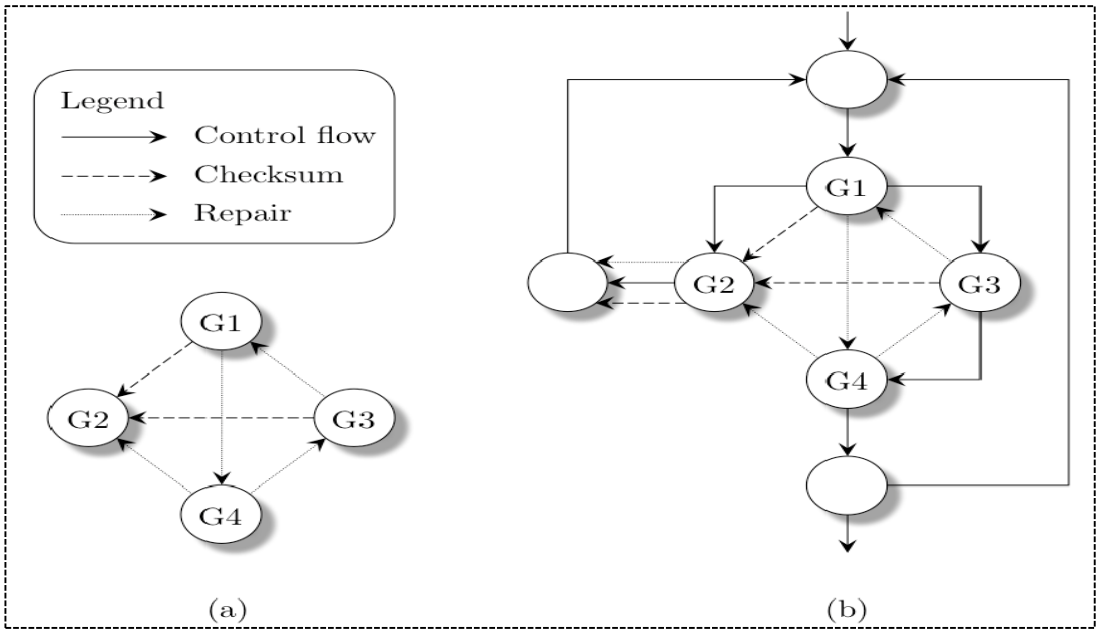


Figure 3.2 *(a) A guard's graph (b) placement of guard's graph in a control flow (Cappaert, 2112).*

Advantages and Disadvantages

The main Advantages of software guards is its ability to repair the modified (damaged) program code during the execution time. Unfortunately, it is difficult to automate their construction, where the strength of protection relies mainly on owner programming skills. Furthermore, the maintenance cost will be very high.

State of the art

The study carried by Erlingsson et al. (2006) proposed a software guard's system to protect the kernel and user mode address spaces. The proposed scheme suggests to run the native plug in a safely code via isolating the untrusted code and interposition the system calls. The idea is to separate all kernel extensions into separated protection areas in order to prevent any chances of faults to occur. Despite that the proposed scheme executes the code in safe execution environment, however its overheads are high because it requires to monitor both the kernel and user mode address spaces. Furthermore, it requires administrative privileges to execute the code.

In another study, Cappaert et al., (2006), presents a software guards based on encryption technique to protect a software from static analysis and tampering attacks. The proposed technique uses the concept of code encryption to generate code dependencies which implicitly protect integrity. Furthermore, they proposed several dependency techniques based on a static call graph that allow code decryption simultaneous with code verification during the runtime of the program. If the code is modified dynamically or statically, it produces an incorrect decryption of other code which generates a corrupted executable code.

In similar study, Cappaert et al., (2008), presents a new software guard technique that are able to encipher code during the execution time, based on other code as key information. The proposed technique uses a code as a key to decrypt other code, which it creates code dependencies and makes the program more tamper resistant. The proposed technique provides a property and confidentiality of code, where other previously proposed software guards didn't provide yet.

Ghosh et al., (2010), presented a software guard approach based on process level virtualization. The proposed approach involves using of encryption and software check summing guards in order to protect the program. The idea is to assemble the virtual machine (VM) within the program during the build time. By this way the program can't be executed without the VM. The VM provides just-in-time decryption of the program. Despite of the strength of this approach; however, it can't prevent the adversary from obtaining an analyzable snapshot of the code.

In another study of Ghosh et al., (2013), they proposed a software guard mechanism that provides a tamper detection during the execution time of the program. The proposed mechanism creates software knots which are an instructions sequence that checksums portion of the code to detect tampering. These knots are used to check the integrity of cached code. Moreover, the proposed mechanism inserts code into a program to generate polymorphic software knots dynamically. The proposed mechanism provides a suitable platform for extending guard protection, however, its overheads is high in term of performance and memory usage.

3.5.2 Code Signing Techniques

Some programming languages such as C didn't have any security mechanisms that check code before execution. Consequently, these languages are susceptible to tampering attacks which modify the program in such way that its computations cannot be trusted.

In order to prevent tampering with a program, its code requires to be protected during storage and transmission. Each time the program executes, it should verify and check its integrity to reveal tampering (Kiehtreiber & Brouwer, 2006). Code signing techniques are most suitable for this type of checking. The owner signs the software where the end user verifies the signature that is appended to the software. This model is shown in Figure 3.3. This model is similar to Windows drivers which are signed by Microsoft and verified by the operating system during the installation time (Microsoft Corporation, 2002).

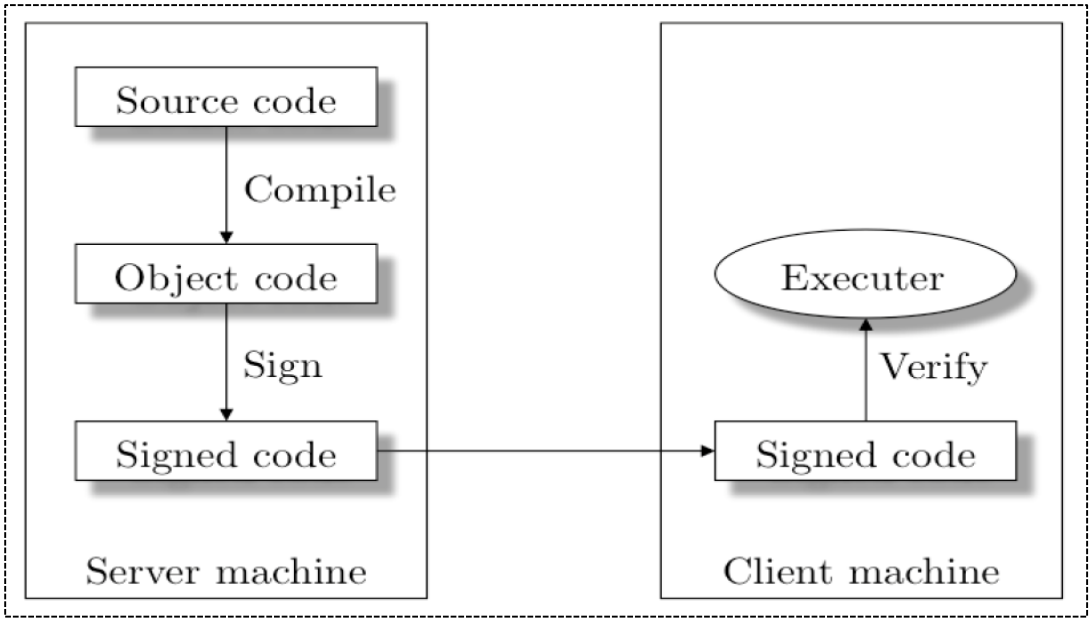


Figure 3.3 *Code Signing Model*

The main disadvantage of these techniques is that the signature's verification process relies on a public key to verify the authenticity of the public key. If not the case, an attacker can generate a signature.

3.5.3 Oblivious Hashing

Chen et al. (2002) proposed an oblivious hashing, which is a technique that allows implicit computation of an actual execution's hash value. The main idea of this technique is to hashing the execution trace of a code's segment, which enable to verify the execution behavior of the program. Hashing instructions are embedded within the original code, in which it takes the results of previous instructions and apply them to hash values that are stored in the memory, as shown in Figure 3.4.

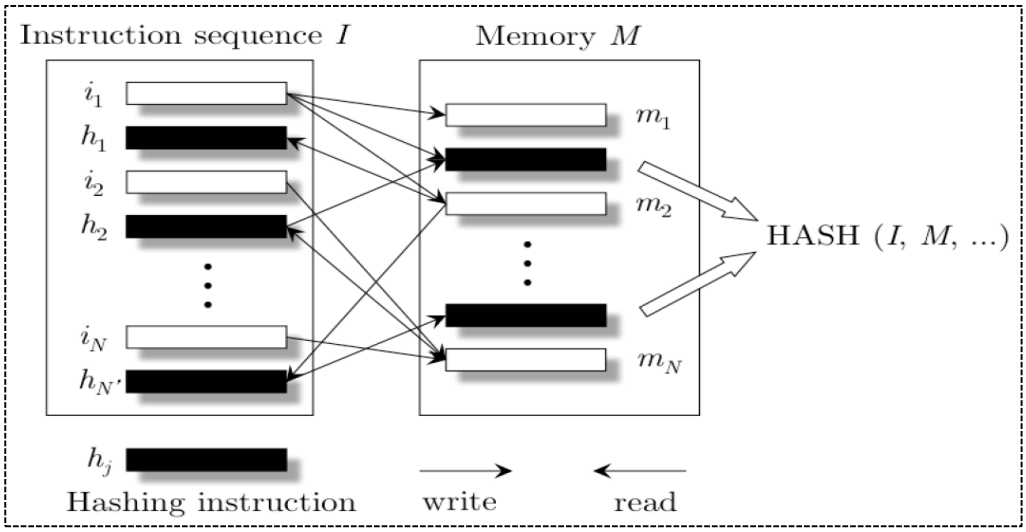


Figure 3.4 *Oblivious hashing are interweaved with original code (Cappaert,2012)*

State of the art

Oblivious hashing can be used for remote code authentication or to provides a local software tamper resistance. For instance, Chen et al., (2007) apply oblivious hashing techniques to Java bytecode in order to protect the call stack. Their idea verifies whether

the executing function is legitimate via inspecting the top stack using interweaved hashing instructions. However, in a white-box cryptography, the remote code authentication isn't an option because local software should provide its own security.

In another study, Jacob et al., (2007) proposed an oblivious hashing to provide local software tamper resistance. The proposed approach combines the oblivious hashing with many other techniques such as opaque predicates, overlapping of instructions, interleaving of instructions, code outlining, branch functions, and junk instructions. Jakubowski et al., (2007) proposed a program predicates to check the integrity of a software. The proposed predicates check dynamically whether a program is in a valid state, in which they act as a generalization of oblivious hashing techniques. They turn a program segment PS into a table of learned Fourier coefficients, where these coefficients can be used to computes the PS.

3.5.4 Software Diversification

Many of code obfuscation techniques offer protection against reverse engineering and program analysis, with varying degrees of strength and complexity. However, many of these techniques do not protect software against Break Once Run Everywhere (BORE) attacks, where this type of attack if successfully crack one instance of a software can be applied similarly to crack all other instances of the same software. Typically, all copies of a software have the same binary code image, which enable an attacker to design a generic reverse engineering and tampering scheme. Moreover, an attacker may generate and use the same attack payload or same attack vector to compromise simultaneously as many software as possible (Davi, et al., 2012). To mitigate this attack, Cohen proposed to diversify the software

into multiple and various instances, where each instance appears differently and in the same time preserves the entire semantics of the original program (Cohen, 1993). Software diversification forces an attacker to tailor a specific payload or attack vector for each instance of software, which make the attack very expensive and reduce its impact (Wang, et al., 2017). Furthermore, it makes the design of a universal cracking scheme for all software instances very hard since each instance should be cracked individually.

Software diversification considers as a leading protection technique against BORE attack. It's significantly increases the effort and the time of attacking an installed base of protected software. Consequently, it is a strong mechanism to protect software that are distributed in large numbers of the customer's devices such as cloud applications, mobile applications, games, and some desktop applications. Furthermore, it can effectively apply to proactively prevent an adversary via regularly upgrading the security software on installed hardware, thus frustrating attacker attempts to crack the system (Xie, et al., 2015). Additionally, replacing security software with new diversified instances will force the adversary to abandon his existing analysis. Ultimately, the effort of breaking code exceeds the value gained.

State of the art

Diversifying mechanisms are proposed with various scopes from a single instruction to the whole software. Typical diversifications include basic block reordering, instruction substitution, and dead code insertion.

Anckaert et al. (2007) proposed to diversified the program during the execution time in order to make a trade-off between unique executions of program and distributing of identical

copies. The proposed scheme makes it difficult to focus on in a point of target code and may fool an adversary from understanding the diversified program. Roeder and Schneider (2010) proposed an obfuscation technique which periodically restart servers with diverse instances of the program. The proposed technique restarts periodically number of compromised replicas which are concurrently executed to complicate the attacker's job. Wang et al. (2012) designed a branch obfuscation scheme which replaced explicit jump instructions with implicit trap codes, then deploys these jump conditions on the remote trusted entity.

Davi et al. (2012) proposed a code obfuscation technique which impede code reuse attacks via implementing software diversity to the binary during the execution time of a program. The proposed technique diversifies the program's code randomly over the entire memory for each invocation. It transforms the control flow graph of a program and allows splitting and injecting of nodes. Despite this technique may thwart code reuse attacks, however it can't prevent any runtime attacks such as return oriented programming (ROP) attack. Additionally, it can't be efficient if the attacker could determine the memory layout.

Kisserli et al. (2007) proposed a technique to protect a program against global tampering via applying diversity per program's instance. The proposed technique creates snippets based on genetic programming ideas to thwart patches depend on low overhead and mass distribution of pirated software. They proposed several kinds of snippets that are targeting distinct patching schemes, where they automated their implementation by applying genetic programming mechanisms. The proposed technique can be useful to protect against probabilistic attacks. However, Adversaries are still feasible to attack unpredictable memory addresses spaces due to limited randomization space of this technique. Pappas et al. (2013) presented a code transformations method which can be implemented statically, without

modifying the basic block's location. The proposed mechanism enables a safe randomization of stripped binaries with partial disassembly coverage. They suggest to introduce in place code randomization to prevent the utilization of vulnerable Windows applications such as Adobe Reader. Friedman et al. (2015) proposed an obfuscation approach that transforms a binary code into chronomorphic binaries which diversify themselves during the program's execution time. The idea is to modify the binary code via moving or changing critical potentially gadgets repeatedly to prevent the attackers from obtained enough information about the software's memory layout. The proposed approach enables the chronomorphing code to rewrite executable code; however, if an adversary can locate the chronomorphing code and exploit it, he can rewrite the code to do whatever he wants.

Xie et al. (2016) presented an obfuscation mechanism based on control flow randomization and instruction fragment diversification. The idea is to generate diversified instruction fragments using different transformation rules, where a random generator functions is used to choose different branches path from the multiple way branches of programs. The transformation rules include rules of junk instructions insertion, instructions expansion, registers transformation, equivalent instructions replacement, and instructions position exchanging. Despite of the proposed mechanism complicates the structure of control flow graph. However, it suffers from high computational time, because it adds more complexity to the actual execution of the program especially when mapping between different program's fragments and their corresponding actual flows.

Sullivan et al. (2017) presented an Instruction set randomization (ISR) obfuscation scheme based on software diversity to protect against code reuse attacks. The proposed scheme applies in place code encryption to hide the code layout of a randomized binary. The idea is

to implicitly restrict control flow targets to basic block entries, then hides the underlying code layout under malicious read access to the program. Despite of the proposed scheme prevents just in time return oriented programming (JITROP) attack, however, it is suffering from a high performance overhead. Furthermore, an adversary could analyse and tamper the software's data via observing each program's execution, then gain the call function's detailed information such as local variables, input parameters, and the return and entry addresses.

CHAPTER FOUR

CODE OBFUSCATION MODELS & TECHNIQUES

Chapter 4 : Code Obfuscation Models & Techniques

4 Introduction

This chapter discuss and review the code obfuscation models and techniques that have been used to protect software against tampering and reverse engineering. First, we presented code obfuscation techniques used to thwart static analysis. After that we discussed the code obfuscation techniques that have been used to thwart dynamic analysis. Finally, we review the hybrid code obfuscation techniques.

4.1 Code Obfuscation Techniques to Thwart Static Analysis

Object oriented programming has been applied everywhere since it provides many features to extend, adapt, and read the code. Unfortunately, this way of programming keeps many traces into the executable file, which can be exploit by reverse engineers and help them in reconstructing the original source code.

The first obfuscation was proposed by Diffie and Hellman (1979), where Collberg et al. (1997) introduced the technique to protect Java programs. On other hand, the first formal definition of obfuscation was given by Barak et al. (2001), where an obfuscator was defined in terms of a compiler which takes a program as input and generates an obfuscated program as an output. Barak et al. defined an obfuscation method as “a failure if there exists at least one program that can’t be completely obfuscated by this method”.

According to Collberg (2002) and Hosseinzadehet al. (2016), code obfuscation can be defined as an attempts to transform a program into an equivalent one which is harder to analyze by reverse engineers and difficult to understand by human. Code obfuscation can apply one or more code transformations which make the source code harder to be analyzed

and become more resistance against tampering with preserve its functionalities. Consequently, the obfuscated code can be distributed to untrusted hosts without risking to be reverse engineering process.

Code obfuscation was initially implemented for programming languages such as Java since Java bytecode is very susceptible to be analysis by attackers, therefore a lot of Java obfuscators and de-obfuscators have been designed (Gautam & Saini, 2017). Furthermore, the .NET obfuscators have becoming popular over the Internet. However, obfuscators of C and C++ are difficult to be found, even if those programming languages are very popular and widely used (Banescu, et al., 2016).

Many commercial code obfuscation techniques just scrambling the identifier names and removing the redundant information form the code such as comments and debugging information. These techniques are quite trivial since obfuscation should offer a lot more possibilities (Viticchié, et al., 2016).

According to Gautam and Saini (2017) and Sebastian, et al. (2016), there are three main categories of code obfuscation and transformations against static analysis:

- Lexical transformation: it replaces the names of variables with meaningless names or names without any semantic value, which will reduce the understanding of the source code by human (Gautam & Saini, 2017).
- Layout transformation: it transforms the layout of a program via deleting the comments, changing the format of the source code, and removing the debug information (Sebastian, et al., 2016).

- Data transformation: it transforms the data and data structures of a program into an obfuscated form (Gautam & Saini ,2017). It involves updating inheritance relations, variable splitting, changing the scope and lifetime of data, and changing the structure of arrays such as splitting, folding, flattening an array, and merging two or more arrays (Sebastian, et al., 2016). Complexity of a program can be increase via implementing a dummy class or creating partitions among classes.
- Preventive transformations: This type of transformation involves inserting junk bytes between program's instructions in order to fool linear sweep disassemblers. The inserted junk bytes are interpreted by disassembly as the beginning of program's instructions, which led to disassemble the obfuscated program incorrectly.

Many commercial techniques concerned on lexical transformations, where the academic researches concerned on other categories.

Advantages and Disadvantages

The flexibility and the low implementation costs is the main advantages of code obfuscation. Moreover, the software can be obfuscated according to the level of security that are needed. Consequently, the extra computation time and cost that may introduce by obfuscation can be traded off with the performance.

The main advantages of code obfuscation and transformation are:

- It is possible to create various instances of the original software in order to obtained a sufficient prevention against global static attacks.
- Code transformations requires a low maintenance cost due to compatibility with other systems and the automation of the transformation process.

- Obfuscation is a platform independent since it can be applied on high-level code.
- Despite of this technique makes static analysis of program harder, However, it does not provide a perfect protection of software against dynamic analysis and tampering.

Related Works

Researchers proposed many techniques to protect the source code from static analysis and also obscure the internal structure of the program. For Instance, Memon, et al (2006) proposed to remove the name of the variables and methods from Java source code in order to hide the completion of code statements. Despite of this technique can fool some de-compilers, but unfortunately, most of recent smart de-compilers can substitute these names with sequentially names and exceed this trick easily. Furthermore, the techniques cannot be applicable to all methods such as the instance method that implements an abstract method of a superclass or the instance method that overrides an inherited method of a superclass.

Sivadasan, et al (2008), proposed a tool for restructuring arrays of java code, they first spit the array into two arrays then merge and folding the arrays, finally they flatting the array. Their tool generates a class that encapsulates the array object where the instantiated objects of those classes used for source code writing.

Sivadasan, et al (2009) proposed a framework for hiding integers of java code using Y-factors; they improved the constant hiding techniques proposed by Ertaul et al (2005). The proposed framework uses the Y_factors to transform the non-negative numbers into simple expression which followed the form of "2*d + r". They used an array of prime numbers where the sum of the numbers in any pair should be a prime number. After that, the pairs of numbers are stored in the array in an increasing order of their sum values. Wu et al. (2010) proposed

a polymorphism obfuscation approach that encodes data into a representation from which looks like a program code. This approach was implemented by Mavrogiannopoulos et al. (2011).

Samawi and Sulaiman, (2013) proposed a multi-client technique to protect the client's programs against static analysis. The proposed technique encrypts the coefficients of the same service using different encryption modulus for each user in order to prevent the same client from revealing the coefficients using different sessions. It charges the clients on a per usage basis, where the coefficients are splatted and obfuscated in different ways. Despite that only the authorized users are able to obtain the original output of the program, however, when the user gets the original code he can modify it easily.

Han et al. (2014) proposed an obfuscation approach to prevent the unauthorized parties from redistributing and reusing the HTML contents. The proposed approach transforms the internal representation of the original HTML text to an unreadable arrangement. The idea is to separate the text of the original content from a single layer into several transparent layers. These transparent layers overlap each other in order to preserve the same visual representation. The user finds it difficult to read and update the obfuscated content of the HTML since it is separated into overlapped layers. However, the proposed approach is suffered from a major problem which are the encryption and decryption functions are build using JavaScript language; therefore, if the user disabled the JavaScript on the browser the proposed approach can't be work. Furthermore, the JavaScript is a client side language, where the end user has a full control on it.

Kulkarni and Metta (2014) proposed an obfuscation approach to protect critical segments of software such as data masking and license checking. The proposed approach constructs a

non-trivial code clones and adds it to the original code in order to provide a better resistance to static attacks. Despite that these clones make the source code more hard to read and analyze by attackers, however they require to be constructed manually which is a time costly and thus they require additional effort for development.

Blazy and Trieu (2016) proposed an obfuscation transformation scheme that relies on a simulation proof. It involves a relationship among semantic states and operating over a realistic programming language such as C. The proposed approach performs a CFG flattening over the arrays and the data of C programs. The Proposed approach should preserve the semantics of the original C program.

4.2 Code Obfuscation Techniques to Thwart Dynamic Analysis

This section presents a number of code obfuscation techniques that are used to obscure the software during the execution time in order to thwart the dynamic analysis. Some techniques obfuscate code offline, while others transform a program during the execution time.

In some cases, the program is not only analyzed by adversaries, but it also tampered using many tampering techniques such as branch jamming attack. An adversary in this kind of attack replaces a conditional jump by an unconditional one in order to force a particular branch to be taken even when it is not being under the expected conditions.

Control flow obfuscation, intermediate code (bytecode) obfuscation, binary obfuscation, and hybrid obfuscation have been presented as the main techniques used to thwart the dynamic analysis.

4.2.1 Control Flow Obfuscation

Control flow obfuscation is the process of obscure the program's control flows via introducing bogus control flows, replacing the control flow instructions with jump instructions, or employing dispatcher-based controls (Junod, et al., 2015). This technique aims to prevent the de-compilers from generating a valid well-structured program (Yasin & Nasra, 2016). Bogus control flows refer to the control flows which are purposely added to a program to increase its complexity but will never be executed at all (Xu, et al., 2017).

The attackers trace a program dynamically in order to collect information that donate where and how controls can be flow from one block to another.

To guarantee that the attacker cannot reach the bogus control flows, Collberg et al. (1997) presented the idea of opaque predicates. According to Mohan, et al., (2015) the opaque predict can be defined as the predicate which its outcome is known at obfuscation time but it is hard to deduce through static program analysis. In general, an opaque predicate can be constantly true, constantly false, or context dependent. There are three ways to generate the opaque predicates: contextual schemes, programming schemes, and numerical schemes (Xu, et al., 2017)

Many obfuscating mechanisms apply program transformations which rely on opaque predicates to obfuscate the control flow transfers (Mohan, et al., 2015); and then obfuscate the data flow via introducing a bogus code in untaken paths. Typical program transformations involved branching the functions, function pointers, and control flow flattening; depend on the fact that pointer analysis and inter procedural alias are "non-deterministic polynomial-time hardness" (Chen, 2009).

To enhance the strength of control flow obfuscation, some researchers proposed to use a signal handling as a mechanism of obfuscation (Zhang, et al., 2013). This mechanism works via artificially generates an exceptions and uses the mechanisms of exception signal handling to hide the control flow. However, these mechanisms normally cause a notable performance degradation when implemented to the whole program level.

Advantages and Disadvantages

Employing control flow obfuscation in software protection makes the disassembly process harder, therefore the de-compiler may fail to return the original code. Moreover, this technique provides a strong protection against both static and dynamic analysis attacks, therefore the adversary requires more time and resources to break this protection.

However, this technique has its drawback in performance because every transformation introduces an extra cost in terms of memory usage and execution time of the obfuscated program.

Related Works

Popov and Andrews (2007) presented a control flow obfuscation approach based on signal handler. The proposed approach replaces the control flow instructions, such as call, return and jump instructions with a trap instruction. When the signal is raised by a trap instruction during the execution time it will be trigger the signal handler of the program. Consequently, the system control is transferred to the original target address again. This approach may provide a good protection against dynamic analysis, however it incurs a high performance overhead due to the high cost of signal handling.

Sharif et al. (2008) proposed a conditional code obfuscation approach which implements the hash function to protect the program's control logics. The proposed approach implements a hashing function algorithm to protect equal logics branched through obfuscating the branches that are triggered via inputs from a continuous interval. In spite of the cryptographic algorithms such as the hash functions are pseudo random permutations, which may complicate the obfuscated branches; however, it has its drawback in performance due to overheads of implementing hashing functions. Moreover, it is a random mapping algorithm which have problems like collision.

Chen et al (2009) proposed a control flow approach to obfuscate the whole branches of a program and insert bogus code. The proposed approach uses the tags like opaque predicates to defeat the software piracy, prevent malicious code injection, and hinder reverse engineering analysis. Their work focus on applying two features, the architectural for automatic propagation of tags and the violation handling of tag misuses. Moreover, a prototype based on Itanium processors has been implemented which exploit the user level exception and exception propagation handling. However, this approach is suffered from a major drawback which is the level of obscurity is rely directly on the number of exceptions that are used for obfuscation since each exception is a flow insensitive and standalone.

Laszlo et al., (2009), discuss the adaptation of a control flow transformation technique of C++ language, they proposed an algorithm which performed a control flow flattening based on control flow information. The algorithm transforms the general control structures and also shows how to deal with unstructured control transfers.

Schrittwieser and Katzenbeisser (2011) proposed a control flow scheme based on software divarication concept in order to prevent dynamic attack. The idea is to splits the code into

small segments before diversification. Moreover, they suggest to reconstructs the control flow of the software before implementing the code. The proposed scheme utilizes the branching function concept via inserting indirect jumps into a program in which their real jump target cannot be detected until execution time. Consequently, the adversary needs to gather all information to get a complete view of the program. One advantage of this scheme is that the control flow information has been stripped from the code area, in which the attackers cannot be able to detect the control flow information when analyzing the code section. However, it is also easy to detect the control flow information through analyzing the data segment since they are initialized and defined to the ordinary variables.

In a similar study, Balachandran and Emmanuel (2013) suggest to remove the information of a control flow from the code segment and hide them into a data segment. During the run time, these control flows are reconstructed in order to preserve the semantics of the software. The proposed approach performs well against static and dynamic analysis. However, the obfuscated program can be easily traced using just-in-time (JIT) compilation, in which the program can be translated to machine code, and then executed directly without any obfuscation.

Balachandran et al. (2014), presented an algorithm that obfuscates the control flow across functions. The proposed algorithm strips the code fragments from the original function and stores it in another function. Each function contains the code fragments from various functions, therefore a function level shuffled version of the original program is created.

Mohan et al. (2015), proposed an opaque-control flow integrity (O-CFI) technique in order to detect the attackers who have steal a full access to the randomized binary code. The O-CFI is a defense technique that prevents control-flow hacking attacks (Carlini, et al., 2015). The

proposed technique checks whether the attacker steal the edges of the control-flow graph from the randomized binary code, heap, or the stack of the victim processes. Despite of the of the strength of this technique, but it can't detect the edges of the control-flow that remained unprotected by the coarse-grained CFI implementation. Moreover, the artificial diversification should be applied to vary the set of unprotected edges among all instances of the program, with preserving the probabilistic guarantees of fine grained diversity.

Peng et al. (2016) presented a control flow obfuscation mechanism for Android applications. The proposed mechanism combines between flattening control flow obfuscation and inserting redundant control flow. Moreover, they suggest to further improve the strength of obfuscation via building a control access policy. The basic idea is to transfer the instructions of control flow from the original code and keep them in another module, function, signal handler, or stack. During the execution time these modules are invoked and the control flow has been reestablished. This mechanism provides a strong protection against automated attacks. However, the adversary can retrieve the address of the control flow that is stored in extra modules. Furthermore, the adversary can build his own de-obfuscation custom script and makes a conjunction with a reverse engineering tools such as IDApro.

4.2.2 Bytecode and Intermediate Code Obfuscation

Traditionally, a program is compiled into native code or semi-compiled code. For instance, java programs are compiled down to bytecode, where the Microsoft .NET programs are compiled into Microsoft Intermediate Code (MSIL).

Most of the symbolic information is stripped off when the program is compiled, where the identifiers which denote functions and variables in source program become addresses in the

compiled program. Consequently, the names of methods, types, and fields are stored in a constant pool within the bytecode file. The compiled form of .Net and Java programs reveals type information through field declarations, method signatures, casts and encoded type hierarchies. This issue facilitates bytecode verification, however it makes them more susceptible to analysis by reverse engineers and decompiled by the attackers.

The Java programming language become more popular since its first release in 1994 (Chan, et al., 2004). Java is a platform independent where the compiled program (bytecode) can run on most platforms. It uses symbolic references to link entities from various libraries in order to achieve a platform independent. On the other hand, the Java Virtual Machine (JVM) works as an interpreter of bytecode, in which it translates and executes them to machine code. Moreover, the dependencies are resolved when the classes are loaded by JVM during the execution time of program (Memon, et al., 2006; Vasudevan, et al., 2015).

Obfuscation techniques are one of the grate defenses against the de-compilers. Obfuscation transforms clear bytecode to more obscure one through encrypt the identifiers and class names in the bytecode files (Ogheneovo, et al., 2014). The obfuscation aims to make the decompiled program harder to understand, so that the attackers have to spend much time and effort on the obfuscated bytecode. Most of the existing obfuscation techniques simply scramble the identifiers and symbolic information in the constant pool of bytecode files.

On the other hand, de-compilers rely on the information that are stored in the bytecode during the de-compilation process, in which the decompiled program is almost identical to the original source program (Buzatu, 2012). There are many commercial and freely de-compilers, where those de-compilers become the fatal weapons of intellectual property violations.

On the other side, Microsoft generates a bytecode file which is very clear, so that it is so easy to decompile this file. Furthermore, Programs which written for the .NET Framework are executed in an environment that controls the requirements program's runtime. This runtime environment is known as the Common Language Runtime (CLR), which is also a part of the .NET Framework (Mei, et al, 2016). The CLR provides the virtual machine's appearance of a program, therefore the developers didn't need to consider the capabilities of the specific processor that will execute the program (Geoffray, et al, 2010). Moreover, CLR provides other important services such as exception handling, memory management, and security guarantees.

CLR and the class library comprise the .NET Framework which is aimed to make it easier to develop computer programs and to reduce the vulnerabilities of security threats of computers and programs (Santos, et al, 2014). Consequently, for these languages that use a symbolic linking mechanism similar to Java such as .NET common language runtime and runtime model of the C# language can be candidate to apply the code obfuscation techniques. Many of .NET obfuscators come into play where these obfuscators strip as much metadata and obfuscate the bytecode to make it more difficult to decompile, but still produces the same result.

Advantages and Disadvantages

Bytecode obfuscation has an advantage over other techniques that the obfuscated bytecode can be distributed and compiled on most machines' architectures, therefore it doesn't require to create binaries for all architectures. Furthermore, this technique strips as much metadata and modify the bytecode in a such way which is more difficult to decompile, but still

generates the same results. Moreover, this technique provides a strong protection against both static and dynamic analysis attacks, therefore the adversary requires more time and resources to break this protection.

However, this technique has its drawback in performance because every transformation introduces an extra cost in terms of programs size, memory usage, and execution time.

Related Works

Obfuscation is a very useful tool for protecting bytecode. Although there are many commercial or ferly products available; however, few researches focus on this type of obfuscation.

Chan et al., (2004), proposed an approach that scrambles the identifiers in the java bytecode by adding an additional information to the identifiers, and stored them into the bytecode file. However, this additional information increases the size of the bytecode file and require additional computation time which reduce the efficiency of program.

In another study, Batchelder and Hendren (2007) proposed a bytecode obfuscation technique that exploits the semantic gap among what is legal in source code and what is legal in bytecode. The proposed technique aims to obscure the operational level via complicated the control flow and the structure of the object oriented design of the program. This technique may have complicated the life of reverse engineer, but it significantly degraded the performance of program because exploiting the semantic gap is not an easy task, which also require much time to be implemented in practice.

The study conducted by Tang et al., (2009), enhanced the work of Chan, et al. (2004) through generates a scrambled bytecode with good obfuscation effects while reducing the

effort spent on manually code development. Hou and Chen (2010) proposed a mechanism to protect the data of a dominant path in a method of a Java bytecode via integrating control flow obfuscation, guards network and oblivious hashing. First, they suggest to build a dominator tree relies on the basic blocks of the target method. Then they select the dominant path of the dominator tree. The bytecode of the dominant path is transformed through the control flow obfuscation, guards network, and the oblivious hashing. Oblivious hashing is used to monitor the stack, where the guards network generates copies of the hash values which are produced by oblivious hashing. As a next step, they hide these copies inside the target method, and then inserts codes to check the values of those copies in various basic blocks. Finally, the target method is transformed randomly using the control flow obfuscations in order to increases the complexity of the guard's network. Despite of the proposed technique may complicate the de-compilation process, however, it adds more complexity to programs which will degrade its performance.

Neves and Araujo (2012), proposed an obfuscation technique which is integrated with the development of C++ programs via employing the compiler itself to implement the obfuscated code generation. The proposed technique uses advanced C++ techniques, such as expression templates, template metaprogramming, and operator overloading. Furthermore, their obfuscator uses the C++ compiler in order to generate a randomized obfuscated code via applying standard techniques, such as dead code generation and opaque predicates.

Andrivet (2014), proposed a technique that implement and enhanced the work of Neves and Araujo (2012) via using only advanced C++ techniques without modifying the compiler and without using any external tool. The proposed technique obfuscates function calls and

string literals using C++ template metaprogramming such as code generation and opaque predicates.

Zhang et al., (2012) proposed a tool for obfuscating the Android applications bytecode. The proposed tool analyzes the bytecode for a given Android application in order to discover the parts worth offloading. After that, it rewrites the bytecode to perform a special application structure supporting on demand offloading. Finally, it generates two artifacts to be deployed respectively onto both the server and the Android smartphone. This method requires a stable and permanent internet connection between smartphone and the server, therefore if a connection is lost, this method fails.

Wu et al., (2016) proposed an obfuscation method for Java bytecode based on Java Virtual Machine (JVM) with a unique cryptographic puzzle. The proposed method introduces randomness elements as a dummy operand inside the methods of bytecode in order to produce multiple obfuscated versions of bytecode. It obfuscates sequence of instructions instead of individual instruction using an encryption key, which allow one to many transformations of the software puzzle. In spite of the proposed method overcomes the existing bytecode obfuscations techniques due to its randomness generation, however, the attacker can perform A Denial-of-Service (DoS) attack by sending a huge number of bogus requests to the server while the client requests a connection with a unique cryptographic puzzle from the server.

4.2.3 Binary Code Obfuscation Techniques

Executable binary code contains a reliable information about the content and the behavior of a program. The link, compile, and optimize steps make the detailed execution behavior of a program differ substantially from its source code (Luo et al,2014). Binary code analysis

can be used to provide information about the content and structure of a program. Furthermore, binary analysis generates an information about the content of the program's code (functions, modules, instructions, and the basic blocks), data structures (stack and global variables), and the program's structure (data and control flow). On the other hand, software reverse engineering techniques are used to analysis the binary programs automatically. The goal of these techniques is to generate a high level representation of the program in order to enable the attacker from understanding and modifying the program's structure.

Storing binaries in encrypted form can provide theoretically perfect protection against attackers; however, it requires to decrypt the binaries during the execution time or having an execute-only memory that can treat with the decrypting binaries when they are loading into that memory. An alternative approach is to leave the binaries in executable form, but to use code obfuscation techniques to make reverse engineering hard. The goal here is to deter attackers by making the cost of reconstructing the high-level structure of the program prohibitively high. Binary obfuscation is a technique used to shadow the real program code to make it hard for an attacker to obtained access to its source code, and make it difficult to understand what the program has to do (Xie, et al, 2010). Binary obfuscation techniques play an important role in evading malware static analysis and detection, where these techniques focus on evading syntax based detection (Wu, et al, 2010). However, semantic analysis techniques and statistical analysis techniques are proposed to thwart their evasion attempts. Most of recent binary obfuscation techniques are used to prevent either semantic or statistical analysis, but not both.

Advantages and disadvantages

Binary obfuscation techniques provide a robust protection to thwart both tampering attacks and reverse engineering analysis, therefore the adversary requires more time and resources to break this protection. However, these techniques have a high performance overhead and can work only in dedicated platforms.

Related Works

Many techniques have been proposed to obfuscate binary code, most of them provide a strong protection, regardless of their overheads in performance.

Wu et al., (2010) proposed a binary obfuscation technique with the potential of evading both semantic and statistical detections. The proposed technique uses a mimic function to obfuscate the binary code into mimicry executables via building a collection of Huffman trees and produces a mimimorphic engine. The mimimorphic engine is added to the obfuscated program in order to restore the original code during the execution time. The Huffman tree have been created for every instruction relies on their parameters occurrence frequency. and evaluate its capability of detection techniques. They implement the mimimorphic engine's prototype on the Intel x86 platform. However, the generated binary code can evade the semantic analysis and statistical anomaly detection. However, it contains the mimimorphic engine and the decoder with the Huffman trees which are not obfuscated, therefore the attacker may reveal the original code using dynamic analysis tools.

Fang et al., (2011), proposed a mechanism to obfuscate binary code in multiple stages. The proposed mechanism used a block obfuscation to hide the binary details into bytecodes, while the multiple stage obfuscation hides the control flow of program in a more complex level via applying a polymorphism tree. The proposed mechanism can be useful to thwart dynamic

analysis; however, it still suffered from exponential computational time since adding the binary details to the bytecode will increase the complexity of bytecode.

Wartell et al., (2012), presented a binary stirring technique which implement a native code with the ability of self-randomization to its instruction's addresses each time it is being executed. The proposed technique generates a new binary code which basic block's addresses are determined dynamically at loading time. Therefore, if an attacker can reveal the code gadgets in one binary code's instance, the instruction addresses in other instances are unpredictable. In spite of the proposed technique transforms the binary code to such way that cannot be easily disassembled by attacker. However, it suffers from very a high computational time because it adds more complexity to the actual execution of the program.

Zhe et al., (2015), proposed a control flow obfuscation approach to protect the control flow of binary code based on code mobility. The proposed approach transforms the most critical control flow logic into a remote trusted entity in order to make the binary code behavior unpredictable either using static or dynamic analysis. The idea is to replace some critical conditional jumps instructions with non-conditional jump instructions in order to hide the original branch conditions and the jump target memory addresses. This approach is not efficient to deter the dynamic analysis since all the original control flow logics should be restore during the runtime of the program.

4.3 Hybrid Obfuscation Techniques

Many obfuscation techniques provide one to one protection, while few techniques have been constructing the many to one protection, where most of these techniques protect the intellectual property and seen as trade secrets. Many to one protection techniques rely on the

fact that combining the encryption with external tools or hardware will increase the software resistance against tampering.

According to Mana and Pimentel (2011) the best way to increase the software security is to utilize the smart card technology with the help of tampering techniques in order to design a robust software protection scheme. The proposed scheme used asymmetric cryptography where the private key is embedded on the smart card. A public key is used to encrypt messages, where the corresponding private key that has been embedded inside the smart card can only decrypt these messages. Furthermore, the proposed scheme requires burning a unique certificate for each user on the smart card. A major trait of the proposed scheme is the potency against adversaries, because it is bypass the threats and code substitution to an authorized management protocol. However, this scheme is requiring a high computational processing due to the using of asymmetric cryptosystem which will degraded the performance of software.

Ghosh et al. (2010) suggest to develop a software that should run only within a virtual machine environment, in which the Just in Time (JIT) compiler can only decrypts and executes the source code. The proposed scheme removes the decrypted code periodically in order to avoid any attempts of analyzing the code or tracing its memory dump. Although the proposed scheme may protect the software from unauthorized users, however, whenever the code is removed from the memory, it is required to decrypt the original code many times for a single program execution, which may degrade the software performance. In addition, the virtual machine should be included within the software, where the authors didn't suggest any mechanism to protect the virtual machine itself.

In another study, Kimball and Baldwin (2012) proposed a software protection technique that applied an encrypted code execution and page granularity code signing to be implemented within trusted emulators. The idea is to combine obfuscation methods, anti-disassembly and anti-debugging within the encrypted source code. The proposed technique aims to prevent reverse engineering from decompiling the code as they run in trusted emulator. However, the encrypted source code should be decrypted before it can be executed, which may consider as limitation of this scheme because the original source code will be disclosed to the adversaries.

Introducing the neural network with a code obfuscation is raised nowadays, where many researches finds that the complexity and powerful computation capability of neural network will increase the robustness of code obfuscation and make the life of reverse engineers more hard. For instance, Ma, Haoyu, et al. (2014), proposed a control flow obfuscation approach, where the execution of the conditional logics is replaced with a neural network that simulates their functionalities. The idea relies on the fact that the conditional logics operations are similar to some kind of binomial classification tasks. Neural network can be used to simulate such classification task, in which it can be trained to respond with the offset of conditional branches and remember any predefined output value that assigned to each group in classification. Consequently, the neural network function can be turned into a conditional dispatcher, where the dispatcher controls the execution path of the program via handling its return address according to the neural networks output. Despite of the proposed approach may confuse reverse engineering from analyzing the conditional branches; however, it couldn’t be applied in complex and nested conditional logics, since there is more than one parameter that controls the output of neural network. Furthermore, the decryption process

will be disclosed to the attackers when they analysis or trace the memory during the execution time of the program.

In another study of Ma, Haoyu, et al. (2016), they proposed a model of dynamic fingerprinting based on neural network to ensure that all of the fingerprinted instances of software have the same behaviors at the semantic level. The authors suggest that the integrated fingerprinting should preserve its semantic transformation and prevent the collusive attack from any execution kind of differential analysis on the fingerprinted software. They implement the integrated fingerprinting on the top of their previously proposed neural network approach. The neural network in the proposed model has been trained to remember both the control information and the fingerprinted message, where the embedded fingerprinting is a part of the neural network outputs. Consequently, the neural network will serve both purposes of fingerprinting and obfuscation.

Lungu and Potolea (2012), presented a locking mechanism based on neural network to protect the software copyrighted material. The proposed algorithm defines two functions, one used to protect the data and the other one to unlock it, where the two functions share the same encryption key. The idea is to replace the decryption function with an equivalent neural network function. The proposed mechanism implements reactionary key generations for the same data that needed to be protected, via applying many to one relationship among the keys and the encryption. The neural network has been used to obscure the description of the decryption function, therefore the neural network was trained to encapsulate the decryption algorithm. In spite of the proposed mechanism suggested that the decryption function used to validate and decrypt a given key should be embedded into the neural network structure, which may complicate the reverse engineering process. However, if the adversary uses brute

force attach against such neural network, he can obtain the key and steal the intellectual property, in which an illegal access could be obtained.

4.4 Other Code Obfuscation Models & Techniques

This subsection presents the obfuscation schemes which apply a combination of existing obfuscation techniques and additional mechanisms, such as use of self-modifying code, time sensitive codes, memory management, and use of distributed system.

Madou et al., (2005), proposed an obfuscation technique which based on dynamic code mutation. The main of their work is to mutate the program by running edit scripts. Therefore, some parts of the procedures in the original program are removed and then a stub is placed at the entry point of the procedure. During the run time of the program, these parts will be restored. In addition, the routine will go into the stub to execute the editing engine and then the stub will be removed.

Dedic et al. (2007), proposed a transformation algorithm that protects software from reverse analysis by emulating the steps follows by attackers during the reverse engineering process. The idea is to tracking out the walks that the attackers made on the software and representing them in a graph diagram. This methodology can be useful to locate precisely the most important parts of the code that needed to be protected. Moreover, proposed scheme involves adding number of tamper detection checks at different locations of program. Each tamper check contains a predefined part of program that needed to be monitored. The proposed mechanism can be useful in some situations to thwart analysis, but it still suffers from exponential computational time.

Anckaert et al., (2009), proposed a scheme to hide the actual data locations during execution using pointer scrambling and periodic reordering of the heap. Furthermore, the proposed scheme mitigates the variables from stack to heap. Their system assumes permutation of memory pages during the runtime using a trusted software-based memory management unit.

In another study, Kanzaki and Monden (2010) presented a scheme to protect the software against dynamic analysis. The proposed scheme overwrites the sensitive codes by inserting of fake codes using self-modifying techniques. The main idea is to return the original code when the execution time of a guard code block become within a predetermined range, otherwise the sensitive code is replaced with a fake code. This scheme may resist the dynamic analysis, but it requires accurate profiling information of original code before obfuscating it such as estimation of the time that taken by guard code.

Falcarin et al., (2011), proposed an obfuscation approach based on deployment of incomplete application. Code of application arrives as a flow of mobile code blocks from a trusted network entity. These blocks are arranged in the memory with various customized layout. This approach may deter static and dynamic analysis due to deployment of incomplete application and code mobility respectively.

Foket et al., (2014) proposed an obfuscation approach that combined three types of transformations: Class hierarchy flattening, object factory, and Interface merging. Class hierarchy flattening removes many of type hierarchy from the programs. Object factory and interface merging remove type information from object creation sites, method signatures, and casts.

Vasudevan et al., (2015) proposed an obfuscation technique which involves an overlaid architecture to handle a class loader system. The idea of this technique is to set variable and methods names as empty or null and also to set the string literals as empty, in order to prevent the de-compilers from identifying the literals and variables of the obfuscated program. The proposed technique enables a method calling and a class loader system in which each class is loaded and called by the class loader architecture. By this way, its enables the using of string literals to call a method or load a class. Although this technique can fool some de-compilers, but unfortunately, most of recent smart de-compilers can substitute these names with sequentially names and exceeds this trick easily.

4.5 Summarization and Critical Discussion

Code obfuscation techniques can be classified into the following: Source code obfuscation, Layout obfuscation, data obfuscation, debug info obfuscation, control flow obfuscation, bytecode obfuscation, and binary obfuscation.

Source code or lexical obfuscation involves renaming the identifiers and variables with meaningless names, removes comments, changes the formatting of the source code and removes the debugging information. This process will lead to reduce size of the software and reduces the understanding of the source code. However, source code obfuscation has its shortage and limitations as follows: first, names of the standard java API classes, which are a part of the JRE, cannot be obfuscated. Second, it cannot rename the entities that are accessed via reflection at run time since the particular method or class might be dynamically accessed, especially if it is belonged to a third party, framework, or it is a part of another application. Third, Serializable classes names cannot be obfuscated.

Layout obfuscation simply transforms the layout of the program via deleting the comments, changing the format of the source code, and removing the debug information

Data Obfuscation transforms the data and data structures of a program into an obfuscated form. It involves updating inheritance relations, variable splitting, changing the scope and lifetime of data, and changing the structure of arrays such as splitting, folding, flattening an array, and merging two or more arrays. Nevertheless, this method faces a major problem that is the encrypted data should be decrypted during the runtime, so a particular decryption code should be included within the software. On the other hand, Array restructuring working only in transforming integer arrays where no string encryption added to the program.

Debug info obfuscation works by hiding debug information generated by java compiler, this information will be need to get meaningful stack traces such as line number information and source file names to the resulting class files.

Control flow obfuscation simply restructures the control flows of the program via introducing bogus control flows, replacing the control flow instructions with jump instructions, or employing dispatcher-based controls. However, altering the control flow may increase the runtime to such a drastic level that could affect the efficiency of obfuscation and degrades the performance of the obfuscated program. Furthermore, the de-compiler may fail to return the original code of obfuscate control flow; However, not all de-compilers are that dumb. The criteria used in evaluating the quality of control flow obfuscation depend on how much obscurity are added to the program. However, the combination of control flow obfuscation with data obfuscations techniques might be a good way for an obfuscator to defy against de-compilers.

Bytecode obfuscation encrypts the identifiers and class names in bytecode files, the obfuscation aims to make the decompiled program harder to understand, so that the attackers have to spend much time and effort on the obfuscated bytecode.

Binary obfuscation is a technique used to shadow the real program code to make it hard for the attacker to obtained access to its source code, and make it difficult to understand what the program has to do. Binary obfuscators work quite well, but in general it is limited to the standard simple binary formats. Moreover, these techniques have a high performance overhead and can work only in dedicated platforms.

Despite of data and string obfuscation techniques can sometimes work well, however it often fails because the programming languages have a complex name resolution rules and different formatting; therefore, processing such techniques usually requires a complete language parsing, not only a string hacking. In addition, layout and semantic obfuscation techniques can work quite well to thwart the static analysis because it encodes constants in an inconveniently readable way. However, when nested commands, multiple statements per line, comments placed around incomplete blocks of codes, unusual and peculiar conventions of identifiers and function names are encountered, as they often exist in complex and large systems; it will case failure in obfuscating the program correctly.

4.6 Conclusion

Code obfuscation is a promising technique used to protect software from being analyzed by reverse engineering. Software owners apply various obfuscation techniques in order to address this issue. Many of these techniques are weak, since they are vulnerable to both dynamic and static analysis. On the other side, other mechanisms are very costly since they

impose considerable performance penalties. Furthermore, most of the available obfuscation techniques are often not good as they rely on “security through obscurity”, where these techniques may deter some impatient adversaries, but against a dedicated adversary they offer little to no security. Consequently, depending only on one technique is not sufficient to deter reverse engineers from analyzing the program; Where the combination of many obfuscation techniques produces a robust protection against many forms of reverse engineering analysis and attacks.

CHAPTER FIVE
PROPOSED SOFTWARE OBFUSCATION MODEL

Chapter 5 : Proposed Software Obfuscation Model

5 Introduction

One of the main concerns for software owners is protecting their software from reverse engineering. If an adversary succeeds in extracting and reusing a proprietary algorithm, the consequences may be significant. Moreover, reverse engineering remains a considerable threat to software developers and security experts. Attackers crack software by reverse analysis methods such as static disassembling, static decompiling, dynamic debugging and dynamic tracing.

In this chapter, we proposed a model to protect software against tampering and both static and dynamic reverse engineering analysis. We begin by describing the threat model, then we described the architecture of the proposed model. Furthermore, we briefly reviewing the concept of static analysis and explain the steps of reverse engineering. In addition, we discuss how the disassembling tools can inverse the assembly stage in compilers to return the source code. In Section 5.3, we present a detailed description of the proposed Static Analysis Prevention (SAP) Module. The proposed protection techniques make it too difficult for an attacker to analyze the java programs statically. Section 5.4 contains a detailed description of the proposed Dynamic Analysis Prevention (DAP) module, while Section 5.5 presents Tampering Resistance (TR) module.

5.1 Threat Model

Threat models identify the threats to a system. They model both the attacker and the system to specify all possible attacks to a system. The threat model of our system is illustrated in figure 5.1.

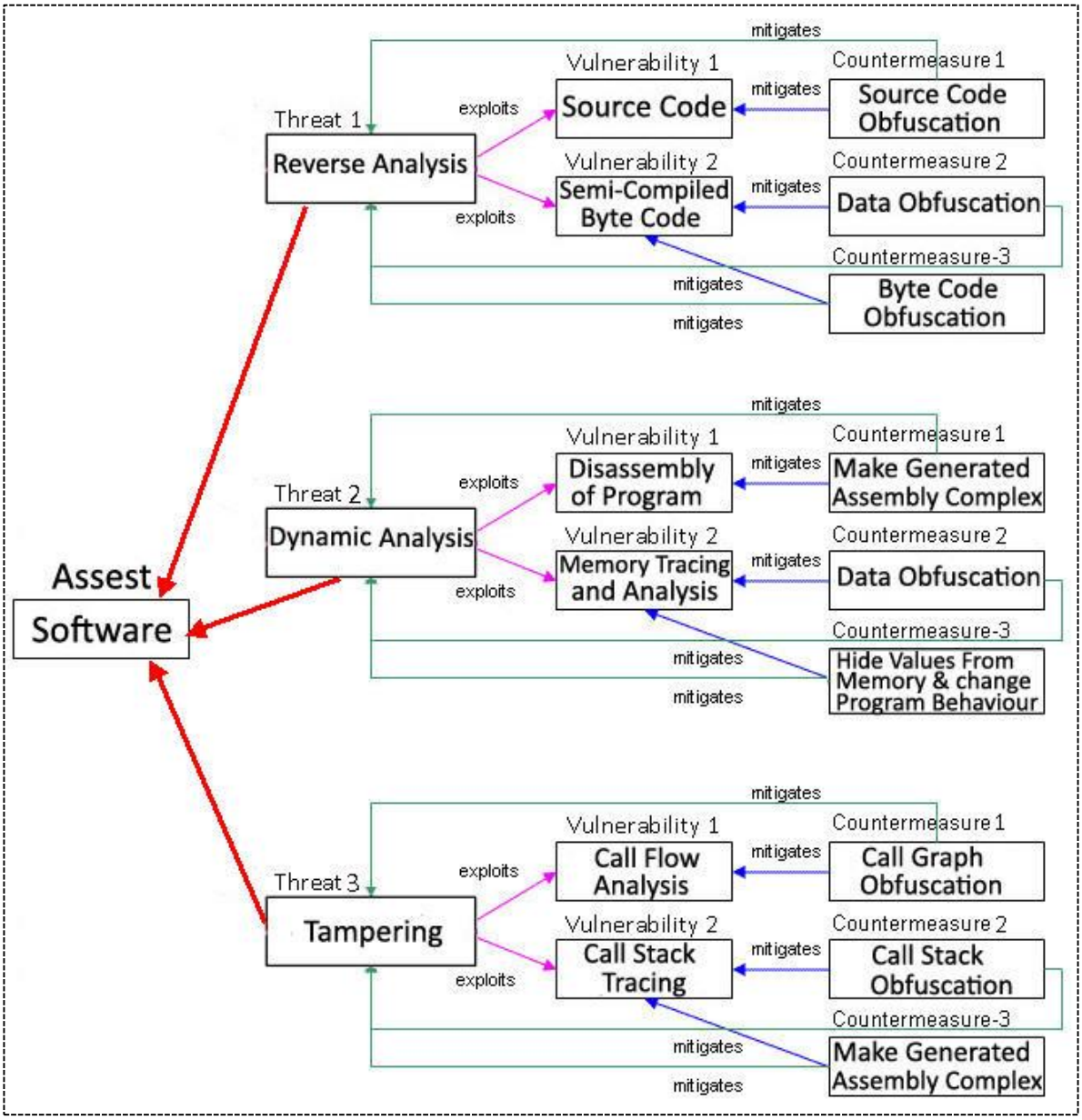


Figure 5.1 *: Threat Model*

After developing the software, it is assumed that the executable will run on an untrusted host machine where the attacker has a full access over the host machine; therefore, he has an

access to the executable code of the software, where he can use any reverse engineering tool such disassembler or debugger to analyze the code. After analyzing the code, he can reuse, modify, or extract proprietary algorithm or data structure of the software. Moreover, in some cases he can insert an extra code to get customers' information by violating the trust factor between vendor and customer. This threat model is commonly called Untrusted Host Threat Model, also it is widely known as the white box model where the attacker has full privileges on the system. The work in this thesis focuses on this threat model.

In this type of threat model, the actual host itself is not trusted, nor is the user. Especially in private systems, but also in corporate environments, end users and their computers cannot be trusted, where the user himself might have malicious intent, such as intent to violate the license agreement that comes with a software application, or extracted and stolen the intellectual property (data and/or code) from the software. Critical systems enforcing certain policies such access control, digital rights management, etc. in order to prevent attackers from tampering their software. White-box attacks are hard to prevent on open systems, such as the PC, but software protection can raise the bar for the attacker.

5.2 Proposed Model Architecture

The proposed model consists of three main modules as follows (The proposed model architecture is shown in figure 5.2):

- Static Analysis Prevention (SAP) module: this model contains three sub modules: source code obfuscation, data obfuscation, and byte code obfuscation. It is aimed to prevent reverse analysis from analyzing and modifying the program’s source code.
- Dynamic Analysis Prevention (DAP) module: this module contains two main sub modules: neural network data obfuscation and neural network control flow obfuscation. It is aimed to prevent attacker form disassembling the code correctly. Moreover, it aimed to prevent them from tracing and analyzing the memory during the run time of the program.

- Tampering Resistance (TR) module: this module contains two main sub modules: call graph obfuscation and call stack obfuscation. It is aimed to prevent attacker form tampering the software and monitoring its behavior.

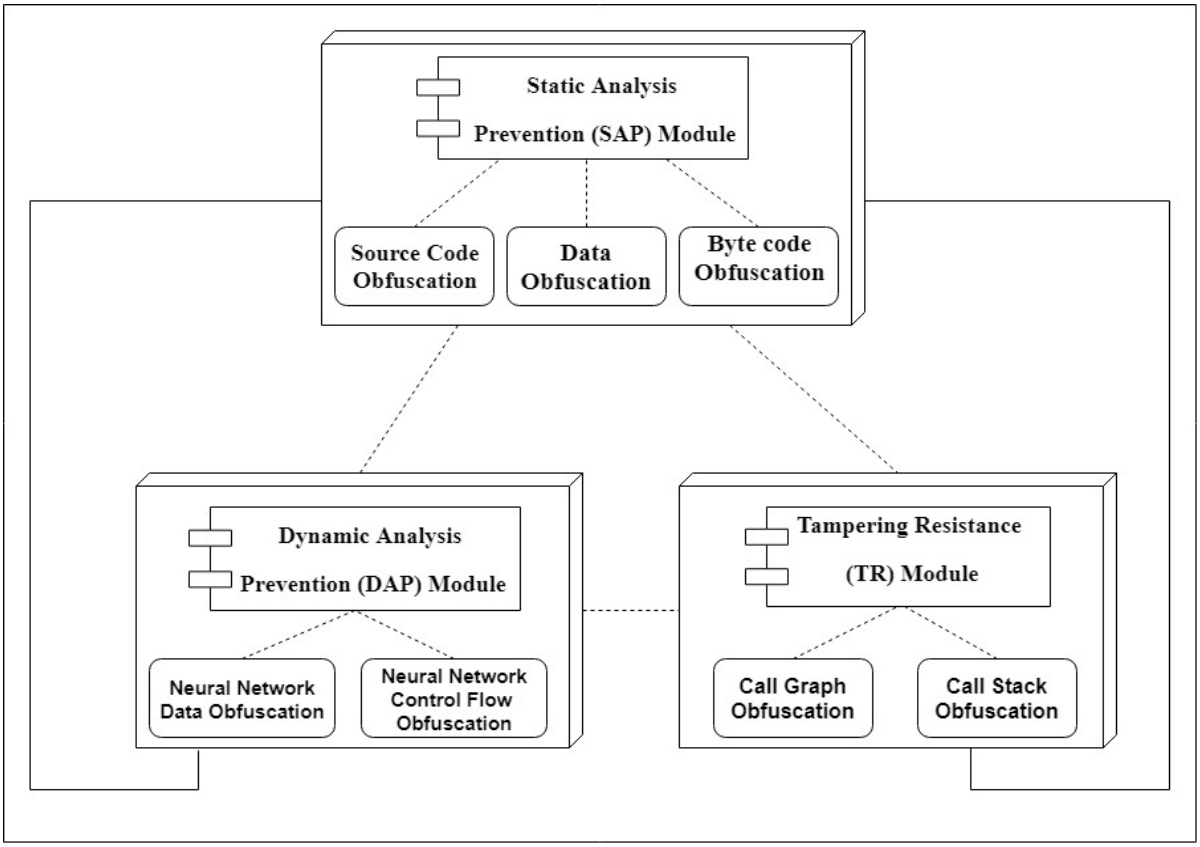


Figure 5.2: *Proposed Model Architecture*

5.3 Static Analysis Prevention (SAP) Module

The proposed SAP module makes it too difficult for an attacker to analyze the java programs statically. It integrates three levels of obfuscation; source code, data obfuscation, and bytecode obfuscation level. By combining these levels, we achieved a high level of code confusion, which makes the understanding or decompiling the java programs very complex or infeasible.

This section is based on work described in (Yasin & Nassra, 2016). The author of this thesis is the principal author of this publication.

5.3.1 Static Analysis

Static analysis is a broad term that refers to analyze the code without actually executing it. Compilers implement static analysis techniques to optimize the code. Reverse engineer typically begin inspecting an object via disassembling it, then they attempt to understand it bit by bit, finding patterns, composing parts, etc. In software, a similar process happens, in which a binary file is disassembled first. After that the reverse engineer decompiles the disassembled code into source code. Finally, the source code is being inspected.

5.3.2 Disassembling

The disassembling process is the inverse of the assembly step in compilers. It translates the binary code to assembly instructions which conform a specific processor's architecture. The disassemblers try to differentiate between data and code when they inspecting the software unless something indicates whether a specific byte is data or code, however a byte could be either data, code, or both (Michiels, et al., 2007). Moreover, in several architectures code consists of multi byte instructions. Consequently, instructions can overlap with other instructions and data. Most disassemblers assume that data and code are not overlapped, and that a program consists of a sequence of non-overlapping instructions (Pang, et al., 2013). Linn and Debray (2003) illustrate that the disassemblers can be fooled by inserting data between instructions which called "junk bytes", because it will disassemble those bytes as instructions.

5.3.3 De-compilation

De-compilation is another technique that can be applied by reverse engineers to statically analyze the software. It translates the bytecode to source code in order to enable the attacker

to understand the entire program. Many attackers prefer to use the de-compilers instead of disassemblers, to avoid themselves from confronting with millions lines of assembly code. A de-compiler typically looks for patterns which can be translated to source code. Moreover, the high level code is more compact and it is often easier to understand by human.

5.3.4 Overview of Proposed Module

Protecting software against static analysis is the first line of defense since it obfuscates the program to an equivalent one, which is harder to analyze statically by reverse engineers and difficult to understand by human.

Proposed module integrates the following levels of obfuscation:

- Source code obfuscation level: At this level, proposed technique obfuscates the source code of the program by replacing the identifiers such as variables, functions and classes names with nonsense names that convey no information.
- Data obfuscation level: at this level, the proposed technique encrypts the values of constants, local and global static program variables in order to make the de-compilation process more complex.
- Bytecode obfuscation level: at this level, the proposed encryption algorithm will substitute the identifiers names of bytecode with Illegal obfuscated identifiers, which will generate a syntax and compilation errors when it decompiled by de-compilers.

Furthermore, the proposed obfuscation algorithms are based on java metaprogramming concepts, which can be defined as the process of writing programs that manipulate other programs or itself depend on metadata with the ability to treat programs as their data (Tanter,

et al., 2008). It means that a program could developed and designed to access, modify analyses, read and transform other programs, and even modify itself at running time (Bosboom, et al., 2012). Working on Meta level allows us to customize the java object model during the runtime environment. Therefore, the proposed techniques work at a level much closer to the java virtual machine rather than dealing with the higher-level language, where most compile time obfuscation tools and approaches worked.

Moreover, we can determine the scope of the customization by which methods and bytecode instructions are wrapped at load time; however, the real nature of the customization is modified at runtime environment.

Rewriting bytecode also can be applied at load time rather than runtime; using an application level class loader or prior to load time by directly rewriting class files. In some cases, it gives the programs more flexibility and efficiently handling new situations without the need of recompilation. These techniques give us the ability to use java objects & classes later at runtime, and also allowing us to manipulate and modify the startup of the program and its behavior.

The proposed obfuscation techniques have many features over the other protection techniques:

1. The proposed techniques integrate multi-levels of obfuscations since depending on one level will not be sufficient to prevent static reverse engineering analysis.
2. Hides the decryption algorithms and embedded them within the obfuscated program's instructions.

3. We do not need to decrypt the obfuscated source code at the first level of obfuscation, because we encrypt the source code without violating the java language specifications.
4. We use advanced programming techniques such as Compile time Reflection and Metaprogramming, which give us the ability to inspect classes, interferes fields and methods at runtime, and also enable us to develop and design encryption and decryption algorithms that can access, modify other programs and the program itself at runtime.

5.3.5 Source Code Obfuscation Sub Module

At this sub module, the proposed technique obfuscates the source code of the program by removing whitespaces and indentation , strip comments, encode the constants in inconveniently readable ways, and replaced identifiers such as variables, functions and classes names with nonsense names that convey no information. It aims to prevent the human understanding of the code, which complicates the statistics analysis of the source code. In addition, this mechanism will reduce the size of program as a result of replacing long names of identifiers with short nonsense names.

The proposed obfuscation algorithms generate a nonsense names in a such way that should not violate the java language naming specifications or causing any compilation or syntax errors. There are some methods in java that could not be obfuscated such as instance methods which implements an abstract method of a superclass, instance methods which overrides an inherited method of superclass, and instance methods which used as a callback function. These methods should be excluded from the obfuscation by including them in an exception

list. As shown in figure 5.3, the proposed source code obfuscation algorithm has the flowing general steps:

1. Traverse the java package and class structure from top to down.
2. If the method is in exception list, keep it without obfuscation; otherwise apply the "stringShuffle" algorithm which will replace the original identifiers with randomly generated nonsense names.
3. Apply the cleaning and optimizing process which removes nice indentation and whitespace, strips comments, removes annotation, and hides debug information.
4. Save the updated file and continue to another file on the program.

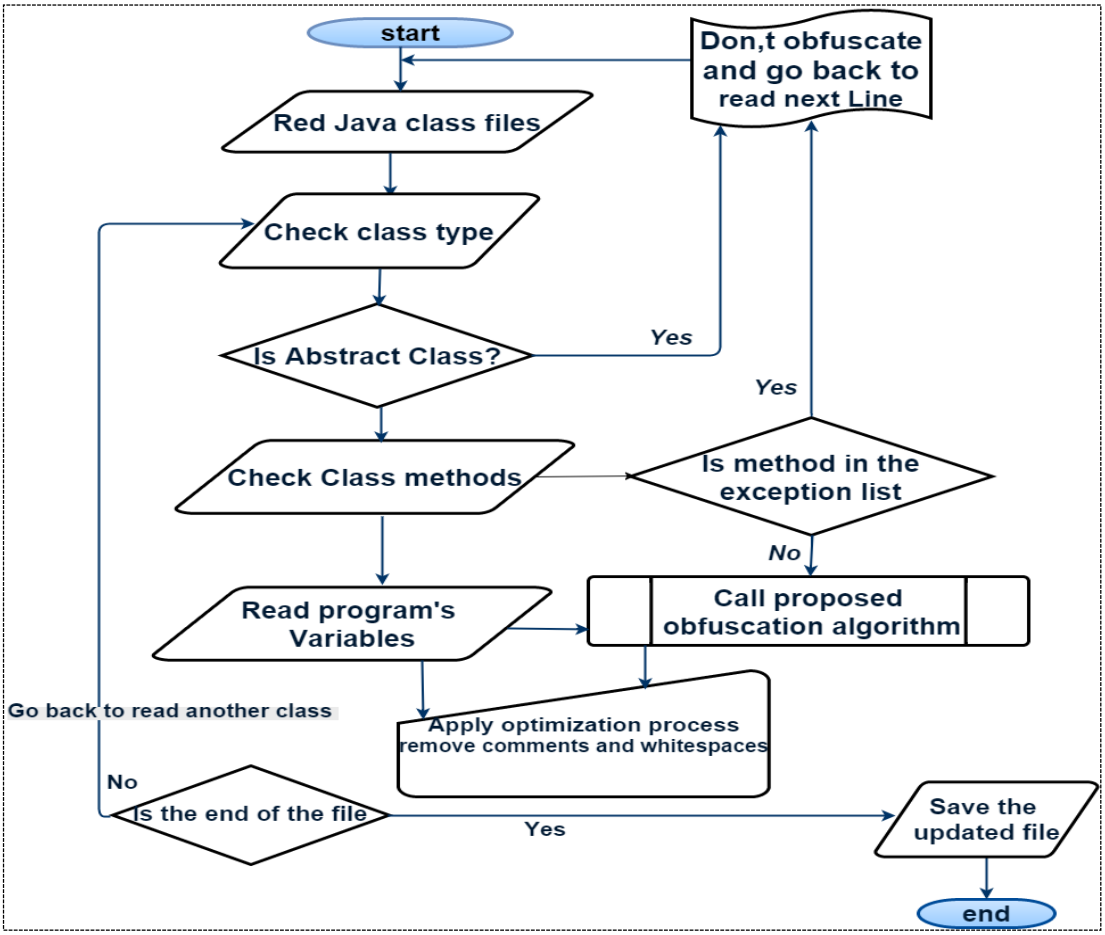


Figure 5.3: *Source code obfuscation algorithm flow chart (Yasin & Nassra, 2016).*

Proposed generator algorithm generates a random nonsense names by applying a combination of letters, numbers, and specific allowed special characters which are the dollar sign and underscore ($, _) characters. The generated nonsense names should follow the java naming specifications and acceptable as an identifiers names. The way of generating the nonsense names shown in algorithm 5.1, which it involved the following steps:

1. Generates random characters in the ranges of (a-z, A-Z).
2. Generates random numbers from (0 -10).
3. Combines the generated value randomly with allowable special characters ($, _) symbols.
4. Shuffled the generated nonsense names by applying the string shuffle algorithm.
5. Check if the shuffled nonsense follow the java naming specifications, if so accept it as a nonsense; otherwise reject it and generates another one.

Algorithm 5.1: Random Seed Nonsense Generator Algorithm (Yasin & Nassra, 2016)

```
1.  FUNCTION RAND_NUM_INT ()
2.  RANDOM RANDOM = NEW RANDOM ();
3.  INT  RANDOMNUM = GENERATE_RANDOM_INTEGER (SEAD) ;  // GENERATE RANDOM INTEGER FROM (0-9)
4.  RETURN RANDOMNUM;
5.  END FUNCTION
6.  FUNCTION RANDOM_CHARACTER_GEN ()
7.  CHAR BASEVALUE GENERATE_RANDOM_CHARACTERS() ;
8.  RETURN (CHAR) RANDOM;
9.  END FUNCTION
10. FUNCTION RANDOM_SEED_GENERATOR ()
11. STRING SEED="";
12. STRING SPECIAL_CHARACTOR="$_";
13. CHAR CHARACTER;
14. INT RANDOM;
15. FOR (INT J = 1; J <= 3; J++) {
16. CHARACTER = RANDOM_CHARACTER_GEN ();
17. RANDOM = RAND_NUM_INT ();
18. SEED = SEED + CHARACTER + RANDOM;
19. END FOR LOOP
20. SEED = SEED + SPECIAL_CHARACTER;   // GENERATE RANDOM SEED
21. RETURN SEED;
22. END FUNCTION
```

In order to complicate the way of generating the nonsense names, and make it more confused we randomly permutated and shuffled the generated nonsense names. However, we should ensure that the generated shuffled nonsense should not violate the java language specification; Finally, we replaced all identifiers names with the generated shuffled nonsense names. The way of shuffled the nonsense names shown in algorithm 5.2 and figure 5.4.

Algorithm 5.2: String Shuffle Algorithm (Yasin & Nassra, 2016)

INPUT: *GENERATED NONSENSE NAME*
OUTPUT: *SHUFFLED NONSENSE NAME*

```
1. FUNCTION STRING_SHUFFLE (STRINGBUILDER SHUFFLESTR)
2. RANDOM NEWRAND = NEW RANDOM ();
3. FOR (INT K = SHUFFLESTR.LENGTH() - 1; K > 1; K--) {
4. INT SWAPWITH = NEWRAND.NEXTINT(K);
5. CHAR TMPCHAR = SHUFFLESTR.CHARAT(SWAPWITH);
6. SHUFFLESTR.SETCHARAT(SWAPWITH, SHUFFLESTR.CHARAT(K));
7. SHUFFLESTR.SETCHARAT(K, TMPCHAR);
8. END FOR LOOP
9. END FUNCTION
```

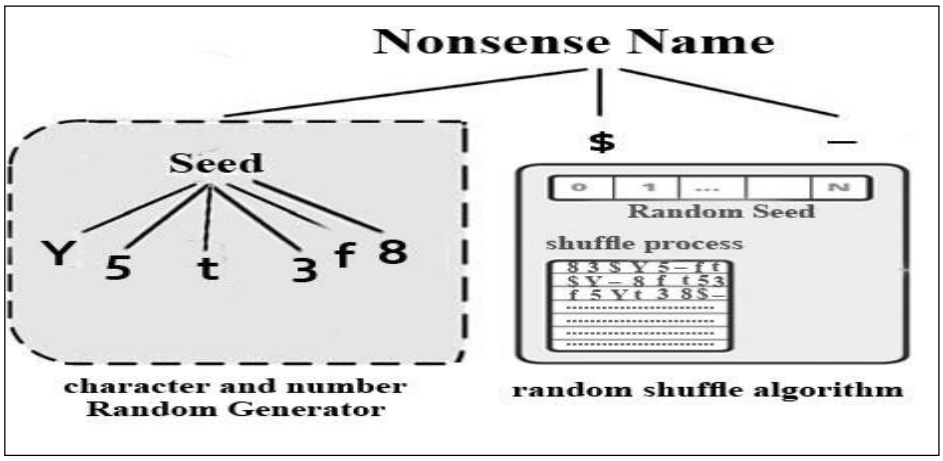


Figure 5.4 *: Random generation and shuffle nonsense names (Yasin & Nassra, 2016).*

As a next step, we optimized the source code by removing the whitespaces; comments removes annotation, hides the debug information. The results after applying the first level of obfuscation shown in figure 5.6, where the original source code before obfuscation shown in figure 5.5. As can see that all identifiers are transformed to nonsense names, which will make the code less clear and difficult to comprehend by human, as an example variable "*employeeName*" transformed to "*Y_1515$q0*" nonsense name.

```
public class Employee {
     private String employeeName;
      private int employeeAge;
      private double empSalary;
      private double salary;
      private double tax;
      private char gender;

   public Employee()
      {
           employeeName = "Employee Name";
           employeeAge = 33;
           tax =  12.5;
           salary = 854.77;
           empSalary = salary - tax;
           gender = 'M';
      }
      public String getEmployeeName()
      {
           return employeeName;
      }
      public double getSalary()
      {
           return salary;
      }
      public void setEmployeeName(String name)
      {
           employeeName = name;
      }
      public void setSalary(double salary)
      {
           salary = salary;
      }
}
```

Figure 5.5 : *Employee class original source code before obfuscation (Yasin & Nassra, 2016).*

```
public class $59_Z4CS {
    private String Y_1515$q0;
    private int _51Yq15$0;
    private double $q0Y_1551;
    private double Y0q515$1_;
    private double 1$50q51_Y;
    private char _0$5qY115;

  public $59_Z4CS()
    {
        Y_1515$q0 = "Employee Name";
        _51Yq15$0 = 33;
        1$50q51_Y =  12.5;
        Y0q515$1_ = 854.77;
        $q0Y_1551 = Y0q515$1_ - 1$50q51_Y;
        _0$5qY115 = 'M';
    }
    public String $_Y51q051()
    {
        return Y_1515$q0;
    }
    public double U8_$7rd2()
    {
        return Y0q515$1_;
    }
    public void _$2rdU78(String d7r_28U$)
    {
        Y_1515$q0 = d7r_28U$;
    }
    public void U78$5qYry(double _d7U$2r8)
    {
        Y0q515$1_ = _d7U$2r8;
    }
```

Figure 5.6: *Employee class source code after the first level of obfuscation (Yasin & Nassra, 2016).*

5.3.6 Data Obfuscation Sub Module

At this sub module, the proposed technique encrypts the values of constants, local and global static program variables to make the de-compilation process more complex. In order to complicate the process of data manipulation, we apply several encryption algorithms and each of them dedicated for one variable data type.

The proposed technique allow the selection of encryption algorithm for string, characters, integers and decimal numbers such as float and double in a random manner. The same

encryption key used in all encryption algorithms. The static data will be encrypted during the program initialization, while it is being decrypted using dedicated methods that manipulate the variables values. The encryption algorithm shown in algorithm 5.3, where the process of obfuscating the data is illustrated in figure 5.7.

Algorithm 5.3: Data encryption algorithm (Yasin & Nassra, 2016)

```
//GENERATE RANDOM ENCRYPTION KEY USING RANDOM KEY GENERATOR
FUNCTION DATAENCRYPTION ()
1.  ENCRPTIONKEY = KEY_RANDOM_GENERATOR ();
2.  //GENERATE THE PERMUTATION VECTOR
3.  FOR (KI IN ENCRYPTIONKEY)
4.  PERMUTATIONVECTOR[I] = KI MOD 8
5.  END FOR LOOP
6.  //TRAVERSE PROGRAM VARIABLES
7.  BEGININDEX = 0;
8.  FOR (ALL VARIABLES V IN PROGRAM)
9.  //DETERMINED VARIABLE DATA TYPE
10. VARDATATYPES =DETERMINED_VARIABLES_DATA_TYPES (VARIABLE V);
11. NOOFBITS = VARIABLE.LENGTH ();
12. ENDINDEX = NOOFBITS;
13. IF (VARDATATYPES EQUAL "INTEGER")
14. CALL INTEGER_ENCRYPTION_ALGORITHM(VARIABLE V);
15. ELSE IF (VARDATATYPES EQUAL "FLOAT" OR VARDATATYPES EQUAL "DOUBLE")
16. CALL DECIMAL_ENCRYPTION_ALGORITHM (VARIABLE V);
17. ELSE IF (VARDATATYPES EQUAL "STRING")
18. CALL STRIN_ENCRYPTION_ALGORITHM (VARIABLE V);
19. ELSE IF (VARDATATYPES EQUAL "CHAR")
20. CALL CHAR_ENCRYPTION_ALGORITHM(VARIABLE V);
21. END FOR LOOP
22. END FUNCTION
```

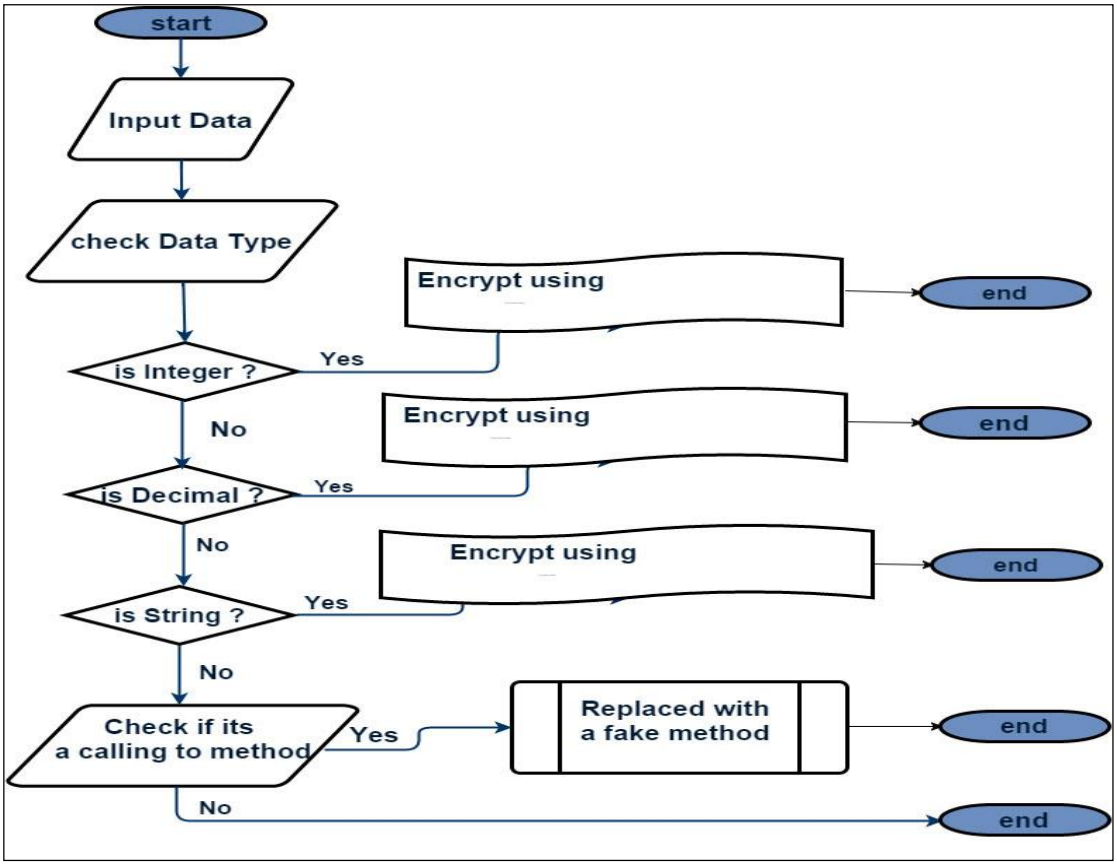


Figure 5.7: *Flow chart of data encryption process (Yasin & Nassra, 2016).*

5.3.6.1 Description of used encryption algorithms

In order to complicate the encryption process and obfuscate the way of generating the encrypted output values, we don't used the normal ASCII code of characters; alternatively, we construct a code table that contains all possible characters and their codes that may be used in the programming languages and it will be used by all encryption algorithms, see table 5.1 below.

Table 5.1 : *Transformation Table (Yasin & Nassra, 2016).*

a	b	c	d	…	z	A	B	…	Z	0	1	…	9	!	@	#	%	?	Space	…
0	1	2	3	…	25	26	27	…	51	52	53	…	61	62	63	64	65	66	67	…

The used encryption algorithms are based on the Permutation and Substitution encryption principles in order to enhance performance, confusion and diffusion characteristics of the Ciphertext.

5.3.6.2 Permutation Algorithm (PA)

The Permutation algorithm is a block cipher that handles each plaintext block independently and the length of each block equal the length of the encryption key in terms of positions number. The permutation process starts by initialization the vector of permutation (VP), which is a key dependent and computed by using the following formula (Yasin & Nassra, 2016):

$$Vpi = Ki*(i+1) \ Mod \ N \qquad i= 0, 1....N-1; \qquad (5.1)$$

Where, N = the length of block cipher;
Ki = code of ith Key Character;

For implementation issues related to data presentation we suggest that N=8 and consequently the encryption key size =8; the resulting VP will be as shown in table 4.2. The first element of VP dictates that the first element of plaintext should be permuted with fourth element (0←→4). From table 5.2, it is clear that the proposed algorithm swapped the position of plaintext, in order to enhance its confusion characteristics. The proposed algorithm uses sliding window technique to deal with the blocks that their size less than the key size by borrowing from the previous block the needed number of character to fill the last block.

Table 5.2 : *Example of Permutation Vector using 8 size encryption key (Yasin & Nassra, 2016).*

Position	0	1	2	3	4	5	6	7
Transposition	4	2	6	1	0	3	5	7

For more confusing and preventing the brute force attack from discovering the permutation pattern, we add extra random characters before and after the permutated string. The number

of these extra characters is equal to the half of the encryption key length. For instance, in our case these extra random characters are equal to 4 since the key length is 8. The process of obfuscating string and applying sliding window technique is illustrated in figure 5.8.

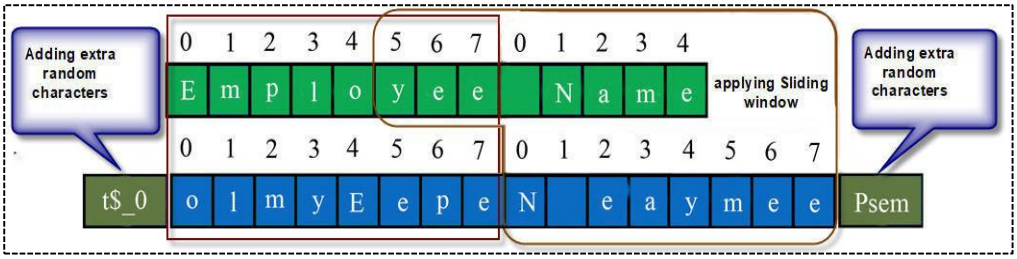


Figure 5.8: *String obfuscation process & applying sliding window technique.*

5.3.6.3 Substitution Algorithm (SA)

SA algorithm encrypts the integer by XORing the last element's position of the key with the integer, where the result value of XOR operation is XORed with the previous element's position of the key. This process repeated until we reach the first element of the key. The encryption function takes two parameters A and K, where A is the integer value and K is the encryption key. This process is described in algorithm 5.4.

Algorithm 5.4: Integer Encryption Algorithm (Yasin & Nassra, 2016)

INPUT: A: *INTEGER VALUE*, **K**: *ENCRYPTION KEY*
OUTPUT: *INTEGER ENCRYPTED*

```
1. FUNCTION ENC (INT A, STRING K)
2. // INITIAL VALUE OF ENCRYPTED INTEGER IS A
3. INT ENCRYPTED_INTEGER = A;
4. FOR (INT I =LENGTH[K] ; I<=0 ; I--)
5. //XORING VALUE OF INTEGER WITH THE KEY
6. ENCRYPTED_INTEGER = ENCRYPTED_INTEGER ⊕ I ;
7. END FOR LOOP
8. RETURN ENCRYPTED_INTEGER ;
9. END FUNCTION
```

We encrypt the float and double using the same technique described above but with small modification that we ignore the decimal point and take the entire number as an integer; for instance, a float number such as 10.5 treated as 105 ignoring the decimal point. In order to

retrieve the original values of numbers during the runtime of program we send the count the digits before the decimal point as another parameter to the decryption algorithm. On other hand, the string and characters are encrypted using PA described previously.

The next step that follows the encryption process of data is the replacement of data value by calling the decryption algorithm in order to obfuscate its real value. By this way, we complicate the disassembly of the code and prevent the Disassembling tools from detecting the actual variable's value. During the runtime the decryption algorithm will reverse the process and restore the original data values. Furthermore, the decryption algorithm will remove the borrowed characters in case of applying sliding window technique, where the number of these characters is send as another parameter to the decryption algorithm.

Moreover, we obfuscate the name of the decryption algorithm in order to hide its real name from attackers; for instance, the decryption algorithm of String is obfuscated to "*Ob_f57_oR*". The results of second level obfuscation shown in figure 5.9.

```
2
3  public class $59_Z4CS {
4      private String Y_15l5$q0;
5      private int _5lYq15$0;
6      private double $q0Y_155l;
7      private double Y0q515$l_;
8      private double l$50q51_Y;
9      private char _0$5qY1l5;
10
11    public $59_Z4CS()
12     {
13
14         Y_15l5$q0 = Ob_f57_oR("t$_0olmyEepeN eaymeePsem",3);
15         _5lYq15$0 = FI_x5(27);
16         l$50q51_Y =  FI_x5(71,2);
17         Y0q515$l_ = FI_x5(85471,3);
18         $q0Y_155l = Y0q515$l_ - l$50q51_Y;
19         _0$5qY1l5 = Ch_f55("28");
20     }
21     public String $_Y51q05l()
22     {
23         return Y_15l5$q0;
24     }
25     public double U8_$7rd2()
26     {
27         return Y0q515$l_;
28     }
29     public void _$2rdU78(String d7r_28U$)
30     {
31         Y_15l5$q0 = d7r_28U$;
32     }
33     public void U78$5qYry(double _d7U$2r8)
34     {
35         Y0q515$l_ = _d7U$2r8;
36     }
```

Figure 5.9: *Employee class source code after the second level of obfuscation (Yasin & Nassra, 2016).*

5.3.7 Bytecode Obfuscation (BO) Sub Module

At this sub module, the proposed encryption algorithm will substitute the identifiers in bytecode with Illegal obfuscated identifiers in order to generate a syntax and compilation errors when it decompiled and recompiled again by attackers.

Java language specification states that an identifier cannot be begin with number or contain some special character such as (;), (:), (/), (%), (!), (#) or space, etc. It should be start with a letter followed by a mixture of letters and digits. In addition, it cannot be similar to a reserved keyword such as null or Boolean literal.

Lexical analyzer can exploit these rules in order to parsing and analyzing a program. However, in the bytecode these rules need not be complied because Java Virtual Machine (JVM) loads the bytecode without verifying whether the names in the constant pool obey with the identifiers definition or follow the Java language naming specification. Consequently the constant pool of the bytecode can contain illegal characters, keywords, null or Boolean literals.

Proposed BO algorithm exploits these rules to obfuscate the identifiers names stored in constant pool of the bytecode with illegal names that does not follow the lexical of Java language specification in order to cause a compilation error when the obfuscated bytecode decompiled and recompiled again. Therefore, the attacker will spend a lot of time and effort debugging it, which is useless. In addition, the de-compiler tools will face a big problem treating such illegal symbols and names, and this will make it too difficult or impossible for any decompiling tools to obtain the original names or handle these illegal characters. However, there are some characters and symbols that cannot be used as an identifier as they have specific meanings for JVM such as ‘‘<init>’’ which used by JVM to call the constructors, ‘‘<clinit>’’ which used by JVM for static member initialization. In addition to the characters, ‘‘/’’, ‘‘:’’ and ‘‘n’’, as the JVM used these characters as a path separator in the file systems host. Furthermore the character ‘‘$’’ is used by JVM as a separator between type and its nested types (Chan & Yang, 2004). Proposed BO algorithm uses a combination of illegal special characters, which are chosen randomly by using the following steps:

- Generate random obfuscated name.
- Select randomly number or illegal special characters from the list (|! | # | % | @ | * | _ | . | ; |).

- Append the selected characters generated from step 2 at the beginning of the obfuscated names.
- Replace all identifiers names in constant pool with the generated obfuscated names.

Proposed BO algorithm uses a semi colon (;) in our encryption process, where the semi colon means end of statement, therefore decompiles will treat this name as two variables. For instance, if we have the flowing statement (x; y), de-compiles divides it into two variables x and y instead of treated it as a one variable. Furthermore, BO algorithm uses the dot character which treated as separators of tokens in a source program; so that, it will make the task of de-compiler more difficult because the Java compiler will consider ‘‘.’’ as a separator between a reference and its members object or a type. The experimental results in next chapter proves that all de-complies fooled by this illegal symbol.

5.4 Dynamic Analysis Prevention (DAP) Module

In this module, we proposed an obfuscating technique based on integrating encryption mechanism within recurrent neural network (RNN) in order to enhance the software protection level against dynamic analysis. We begin by briefly reviewing the concept of dynamic analysis. The proposed protection module is achieving a high level of code confusion, which makes the understanding or decompiling the programs very complex or infeasible. This work tries to fill this gap by presenting our experience with several encryption techniques. This section is based on work described in (Nassra & Yasin, 2018). The author of this thesis is the principal author of this publication.

5.4.1 Dynamic Analysis

Dynamic analysis techniques are implemented during the execution time of the program, where these techniques trace the executed instructions, data values, and register contents.

This form of analysis has more powerful than a static analysis; however, it requires more analyzing time and more hard work (Canfora, et al., 2011). Furthermore, it requires a platform similar to the target code's platform. In some cases, a program may be equipped with anti-debugging techniques which may inhibit the dynamic analysis process.

5.4.2 Overview of Proposed Module

In this section, we proposed a data and control flow obfuscating techniques based on integrating encryption mechanism within recurrent neural network (RNN) in order to enhance software obfuscation level. Neural network provides a robust security characteristic in software protection, due to its ability of representing nonlinear algorithms with a powerful computational capability. Furthermore, understanding the operations of a RNN is complicated as the internal knowledge is embedded in a complicated, self-contradictory, and distributed structure. In order to complicate the reverse engineering of the software and hindering the Concolic testing attack, we train the neural network to simulate conditional behaviors of a program. Consequently, we replace the critical points of program's data and control flow with a semantically equivalent neural network. The system designed to enable the neural network generation of different encryptions for the same protected data. This creates complex relationship between the keys and the encryption. The protection presented by our mechanism is robust against static and dynamic analysis. Furthermore, our evaluations confirm that employing the neural networks in our system significantly increase the difficulties in revealing the obfuscated software.

5.4.3 Data Obfuscation Using Neural Network

Neural network provides a robust security characteristic in software protection, due to its powerful computational capability in representing nonlinear algorithms. Many studies demonstrate that the neural network is a universal approximation function that can be simulate arbitrary functions (Bengio, et al., 2013) (Schmidhuber, 2015). Furthermore, researches show that neural networks with more than one hidden layer could be more robust (Lungu, 2012). Protection using the neural networks is more powerful than the traditional way of protection since the neural network is a non-human readable structure, where many neurons could be used to protect a single part of code.

The proposed technique encrypts the software's data based on a neural network via providing it with the data and the actual encryption key, where the encryption algorithm is based on permutation encryption principles. The same encryption key is used in both encryption and decryption algorithms. The following model is used to obfuscate the data of the software:

Let Kv = {k0, k1, k2 …, kn}, be the set of keys received from the user to encrypt/decrypt the data (D) of the software.

Let F(K) = F (Kv) = k0 ⊕ k1 ⊕ k2 ⊕ … kn, be the function that generates the actual key used to encrypt/ decrypt the data (D) of the software.

Let En: Enf (D, F(K)) be the function which encrypts the data using the actual key.

For the function En we should have the function

Dn: Dnf (E, F(K)), the function which decrypts the encrypted data (E), given the actual key.

We use the neural network to obscure and encapsulate the characteristics of both En, and Dn, in order to increase the complexity of detecting and analyzing the embedded logic of the obfuscated software routines.

The proposed neural network has three inputs: data, encryption key, and the permutation vector as shown in figure 5.10, where the encryption algorithm is based on the permutation cipher principles.

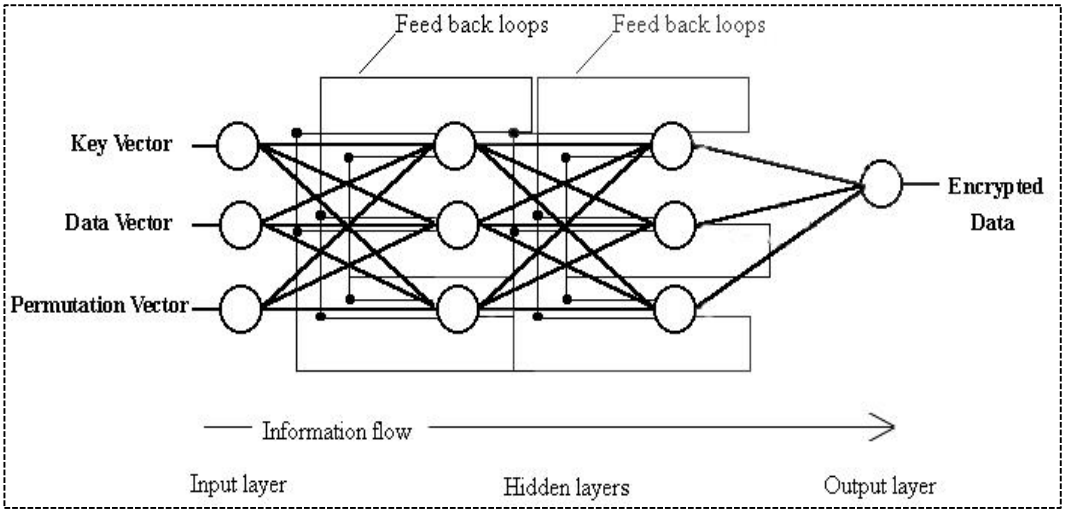


Figure 5.10*: Proposed Encryption based on RNN*

The Permutation cipher is another form of transposition cipher, in which it handles each block of plaintext independently, where the length of block cipher and the plaintext are equal in terms of positions number, an example of permutation vector is shown in table 5.3. The permutation process starts by initialization a permutation vector (PV) which can be computed via applying the following formula (Yasin & Nasra, 2016):

$$PVj = Kj*(j+1) \bmod M \qquad j = 0, 1 \ldots M-1; \qquad (5.2)$$

Where, M: is the block cipher length;
Kj = ASCII code of jth Key Character;

The proposed algorithm applies a sliding window technique when the data block has a size less than the size of the encryption key, via borrowing from the preceding block the needed number of characters in order to fill the last block (Yasin & Nasra, 2016).

Table 5.3*: Example of Permutation Vector using 8 size encryption key. (Yasin & Nasra, 2016)*

Position	0	1	2	3	4	5	6	7
Transposition	4	2	6	1	0	3	5	7

Due to approximate nature of the neural network, the proposed technique suggests to attach the obtained neural network with the obfuscated software, where the obfuscated data will be embedded into the structure of neural network. The neural network uses the encryption key (k) to encrypt /decrypt the data successfully as follows:

Given (k1, k2, …, Kn), the user's encryption keys
Given the PVi, the permutation vector.
For each En of the encrypted software:
Substitute En with NNE (D, F(Kj), PVi), where NNE is the neural network encryption function
Substitute Dn with NND (E, F(Ki), PVi), where NND is the neural network decryption function

The above formula encodes the data block using the actual encryption key based on permutation vector, where the results are padded with the neural network output. On the other hand, the decryption function recomposes the original block through decrypting the embedded encrypted block using the same encryption key.

Furthermore, the proposed technique encrypts the values of constants, global, static, dynamic and local variables in order to complicate the process of data manipulation. The static data are obfuscated in advance using the neural network, where its value is replaced with the neural network decryption function that restore their original values during the runtime of the program. On the other hand, the dynamic data are encrypted and decrypted using the neural network that manipulates their values during the runtime of the program.

As a next step of encryption process, the data value is replaced by the calling of the neural network's decryption function in order to obfuscate its real values, complicate the disassembly of the code, and preventing the tracing memory tools from recovering the real data.

In order to complicate the adversary task of locating and analyzing the weight matrix of the neural network, we recommend to use a heap allocation in order to store it with a pointer aliasing. Consequently, the neural network creates and updates the data dynamically, as shown in figure 5.11; where a memory space is allocated to store the weight matrix. We create the aliasing pointers that are referencing the locations in the allocated memory space in order to generate a complex aliasing effect. Until the neural network matrix is constructed, the allocated memory space is updated progressively with their operands by deploying the program's instructions into the code sections of the software. By this way, the neural network parameters split into various blocks inside the instructions that are distributed over the body code of software. Furthermore, updating the weight matrix is encapsulated with dynamic and complex data dependencies. Consequently, the adversary cannot reverse the neural network during the runtime of program without actually reaching the corresponding inputs.

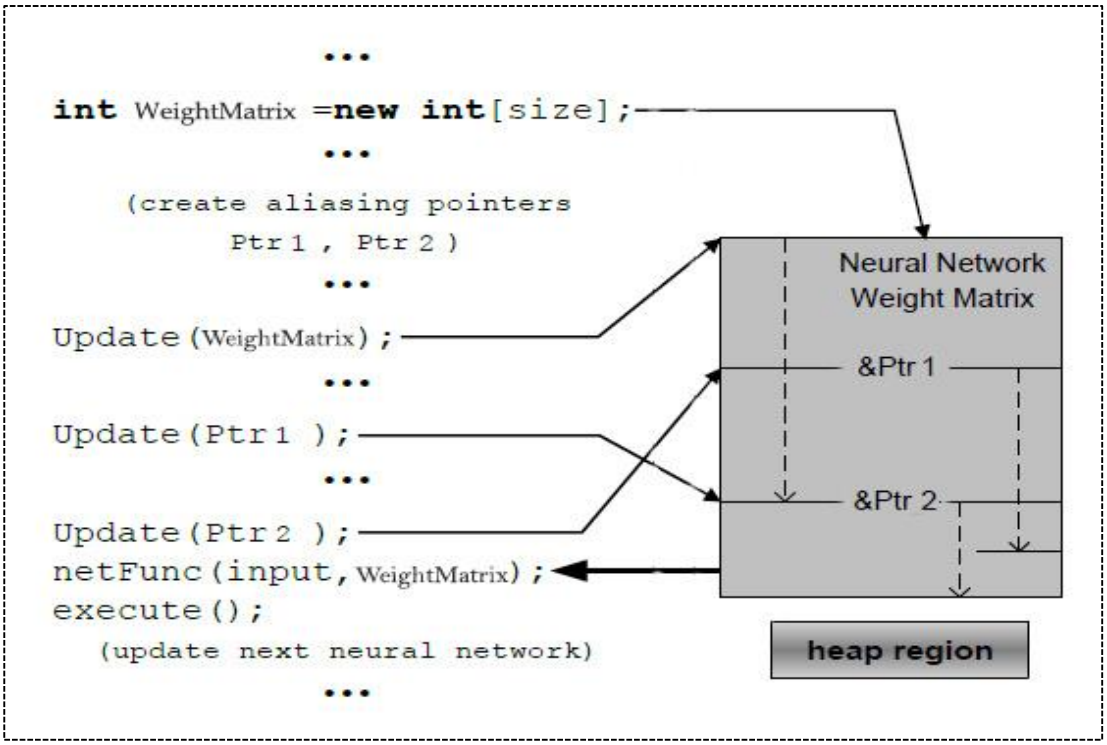


Figure 5.11: *Dynamic creating and updating the data used by the neural network*

5.4.4 Control Flow Obfuscation Using Neural Network

Conditional branching is commonly used to switch control to one of two execution paths, depend on whether the input's value satisfy given conditions or not. In some sense, such selective operation is similar to binomial classification tasks; in which, the input is examined where it could be assigned to one of two groups (true/false), where the control logic of program is switches to the corresponding execution path as shown in figure 5.12.

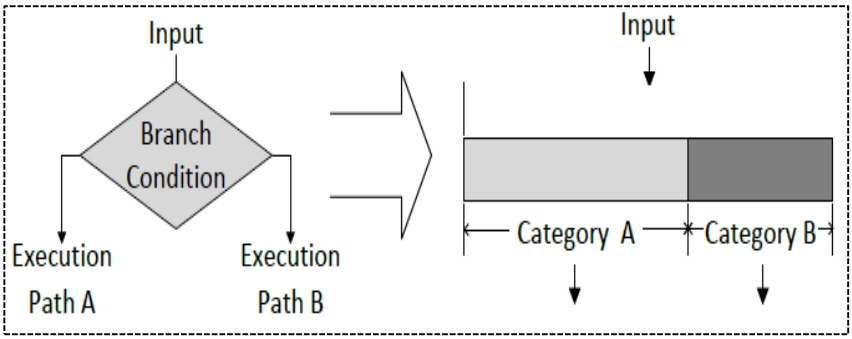


Figure 5.12: *Relationship between conditional branching and classification*

Consequently, we can build a control flow obfuscation model depend on classification algorithm's properties. The neural network is used to design our control flow obfuscation since it is a common understood classification tool, and also due to its incomprehensible way of reasoning. we employ the neural network to obfuscate the program's control flow via training it to simulate conditional behavior of a program. Our method designed to enable the neural network to execute conditional control transfers where the complexity of neural network ensure that the protected behavior is turned to a complicated and Incomprehensible form, making it impossible to extract its rules or locating the accurate inputs which lead to the execution paths behind the network. Thus, it can hinder attacker from detecting the control flow structure of the obfuscated program.

First step, the obfuscator locates the conditional branches' target of a program, and then it creates a training set via choosing a sequence of values which trigger both execution paths. Second step, we train the neural network to simulate the behaviour of the target conditional branches. Third Step, the obfuscator adds a function to the program in order to calculate the neural network's output. Finally, the instructions of the targeted conditional branches have been replaced with a call to the function of neural network. After set of trains, the neural network receives the inputs, and then switch the control flow towards the correct execution path. By this way our model converts the conditional branches into more complicated form and semantically equivalent one, similar to previous control flow obfuscation schemes.

Conditional branching decides whether to transfer control to a specific block of code or stay on the same code block (or in case of loop, it go back again to a preceding one), where the target instruction's address is commonly represented via a relative offset to the instruction pointer's value. Consequently, the neural network is capable to respond with branches' offset since it is strong enough to remember any predefined output value that assigned to each classification group. We implement our control flow obfuscation by replacing the conditional branching instructions with a calling to neural network dispatcher function which determines the execution path depend on whether the output of neural network is true, and turn program execution toward the correct path. Therefore, the neural network dispatcher will automatically push the branches addresses into stack for function returning, where the it can control the execution path of program via manipulating its return addresses according to the neural network's output.

By this way, we increase the confusing of reverse analysis tools because we turn the conditional branches into indirect control transfers.

5.4.5 Neural Network Training

The training usually begins by assigning the parameters of the network, such as the weights and bias factors, then adjusting the weights progressively until the difference between the actual output and the desired output of the training sample is reduced to a negligible degree.

The training set that has been designed to train the proposed neural network contains100,000 inputs/output pairs. We select the first 70,000 elements as a training sample while the final 30,000 elements are used as testing sample. We initiate the set of network parameters randomly, then we adjust the weights and the bias factor until the resulting network fits the desired output via applying the learning algorithm.

The training set that is used during the training process has the following structure:

- Input: Encryption key (*F(Ki)*), Data (D) Vector, and Permutation Vector (PV_i).
- Output: Encrypted data (E).

We also use the backpropagation algorithm as a learning algorithm, but with a little twist because the parameters of network are shared by all time steps. Moreover, the gradient at each output don't depend only on the calculations of the current time step, but it depends also on the previous time steps. As instance, in order to calculate the gradient at time step 5 *(t = 5)*; we would need to back propagate 4 steps and sum up the gradients. This type of backpropagation algorithm called Backpropagation Through Time (BPTT). Moreover, the output layer and the hidden layers are used a tangent-sigmoid function as a transfer function *(2/(1+exp(-2*n))-1)*. With these settings, a learning rate of 0.2 is granted the best results.

We implement the training process using the neural network's MATLAB tool. The training performance is measured using the Mean Squared Error (MSE), where the learning process stops either when the 4000th iteration is reached or when the generalization stops improving.

The training process of the proposed network took 3000 training iterations until a negligible MSE was obtained. MSE is degraded to 0.00001 as a best value since there are no any further improvements that can be generated after that, as shown in figure 5.13 below.

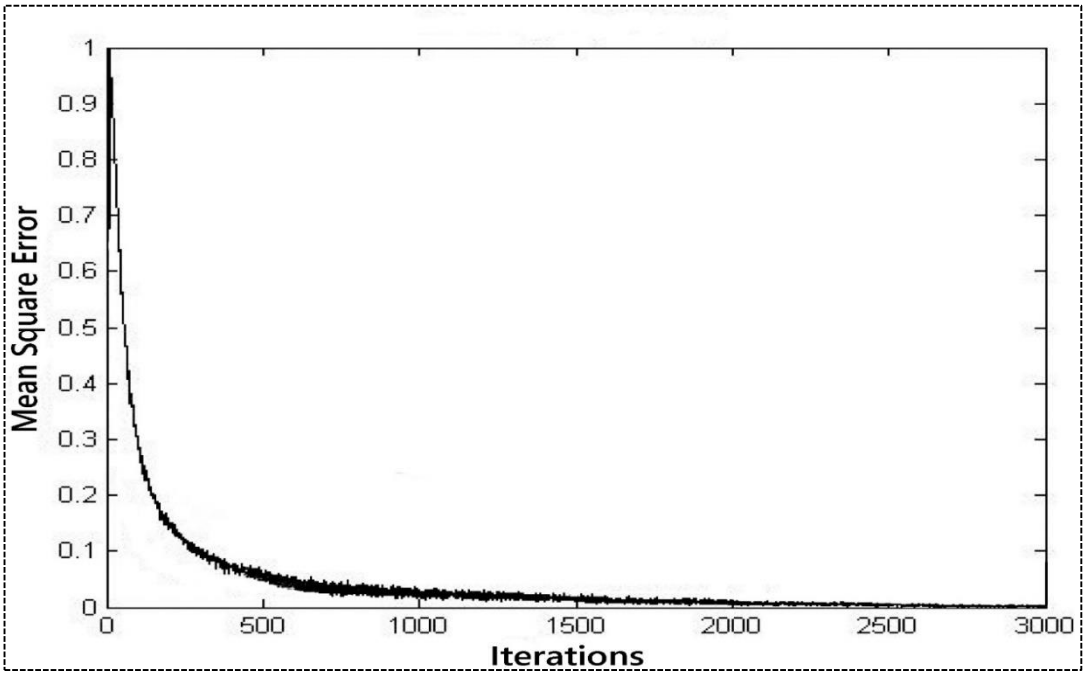


Figure 5.13: *Neural Network MSE During Training*

5.4.6 Neural Network Implementation

Recurrent Neural Network (RNN) is employed as an architecture of the proposed neural network. We employ the RNN because it has a memory that can capture information about what has been calculated so far in the previous time steps, while the traditional neural network assumed that all inputs and outputs are independent of each other, which is not a good solution for many approximation task problems.

RNN is a neural network with feedback connections that allow the data to flow both forwards and backwards within the network (Bengio, et al., 2013). RNN called recurrent since it can perform the same task for each element of the output based on the previous computations (Le, et al., 2015) (Schmidhuber, 2007). It can learn many tasks, processing sequence and behaviors that cannot be learnable by traditional neural network (Mikolov, et al., 2010). Furthermore, it can learn algorithms with or without a teacher to map input sequences to output sequences.

The proposed RNN contains two hidden layers each of them contains 10 neurons, one input layer with 3 inputs (Key, Data, and PV_i), and one output layer to generate the encrypted data (E). The tang-sigmoidal function is used inside the neurons as an activation function with backpropagation as a learning algorithm.

The proposed network after a set of training should capable to captures information about what has been calculated so far in previous time steps; therefore, it can remember how to encrypt and decrypt the data during the runtime of the program. The typical RNN applied the following formulas: (Mikolov, et al., 2010)

$$ft = \sigma g\,(Wf\,xt + Uf\,ht\text{-}1 + bf) \quad (5.3)$$

$$it = \sigma g\,(Wixt + Ui\,ht\text{-}1 + bi) \quad (5.4)$$

$$ot = \sigma g\,(Woxt + Uo\,ht\text{-}1 + bo) \quad (5.5)$$

$$ct = ft \circ ct\text{-}1 + it \circ \sigma c\,(Wc\,xt + Ucht\text{-}1 + bc) \quad (5.6)$$

$$ht = ot \circ \sigma h(ct) \quad (5.7)$$

Where ct: is a cell state vector, xt: input vector, ht: output vector, W, U, and b: are parameter's vector and matrices, ft: Forget gate vector (weights required to remember old information).

σg : is the sigmoidal function , σc ,σh: is the hyperbolic tangent (the activation functions).

it = Input gate vector (weight required to acquire the new information)

ot: Output gate vector (output candidate).

∘: denotes the Hadamard product (entry-wise product)

From above formulas, we can state that the hidden state c_t is the network's memory that can capture information from all preceding time steps. Furthermore, the output at step θ_t is calculated based on the memory at time *t*. As shown above, f_t is not able to capture information from too early time steps.

Traditional neural network can use different parameters at each layer, while RNN shares the same parameters within all time steps (Karpathy, et al., 2015). Furthermore, it can perform the same task at each time step, but with different inputs. Consequently, we reduce the total number of parameters that needed to be learned. Furthermore, the above formulas indicate that RNN has an output at each time step. However, we think that this task is not necessary because we are care only about the final RNN output, not the affection after each step. In addition, sometimes we don't need inputs at each time step.

5.4.7 Error Detection and Correction Mechanism

Due to sensitivity of the software, the errors is not acceptable in encryption and decryption process; therefore, we are applied an error detection and correction mechanism. The error is occurred when the value of a bit is changed from 0 to 1 or vice versa during the encryption or decryption process.

One of the most common mechanisms of error detection and correction is the Hamming Code. The Hamming Code can be applied for data of any size, in which n-parity bits are added to m-bits data, and generating a new data block of (m + n) bits. The positions of bits are numbered sequentially from 1 to m + n. These positions with powers of two are reserved for the parity bits, where the remaining bits are reserved for data bits (Bulo, et al., 2016).

Hamming code can be applied to detect and correct a single error. Therefore, this technique is applied for the error detection and correction process in the proposed model, since this scheme is commonly used in real-time applications.

We have applied the Hamming code scheme for all data types such as integer, float, and string, in which for each 8-bits data block, a four parity bits is generated. For example, the integer data type consists of 16 bits, we generate a hamming code of 8-bits, in which each 4-bits of this code is used to correct 8-bits of the integer value as shown in figure 5.14. For String data type, we generate a hamming code for the first four bytes only.

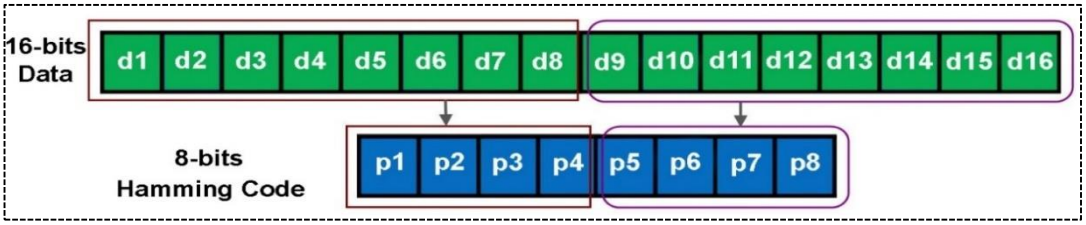


Figure 5.14: *Hamming code of 16-bits integer data type*

In order to improve the performance of the obfuscated software and avoid the overhead that my causes form the hamming code process, we suggest to obfuscate only the most critical data since obfuscating all data makes the generated software too suspicious, as well as, the execution time of the obfuscated software may become high.

The hamming code is implemented in the proposed model by creating a code for every critical program's variables and data, as shown in figure 5.15. When the data has been read from the memory, it will check against errors including their parity bits. The hamming code gives the position of the erroneous bit if only a single bit is incorrect, in which the erroneous bit will be corrected by flipped its value.

```
// data
String messageText = "Dear Sir, Hello";
int number = 12345;
int sum = get_sum(number);
//Hamming codes for these data
int Hamming_messageText = Hamming_Function(messageText);
int Hamming_number = Hamming_Function(number);
int Hamming_ sum  = Hamming_Function(sum);
```

Figure 5.15: *Example of Hamming Code Implementation*

5.5 Tampering Resistance (TR) Module

In this module, a tamper resistance mechanism based on obfuscation and diversification. The proposed module combined call graph obfuscating, stack obfuscating, diversification, memory layout obfuscating, randomization, and basic blocks reordering in order to thwart tampering and increase the difficulties of static reverse analysis and dynamic stack tracing analysis. This section is based on work described in (Nassra & Yasin, 2018). The author of this thesis is the principal author of this publication.

5.5.1 Overview of Diversification and Call Stack Analysis

Code Obfuscation techniques can effectively increase the difficulty of analysing and understanding the program. Static and dynamic reverse analyses techniques can combine together as a practical process of analysing the program. Data and execution flow of a program can be observed via reverse engineering analysis, in which behaviour of the program can be analysed and understand. Software owners apply various protection techniques in order to address this issue. For instance, control flow diversification is proposed to thwart static and dynamic engineering analysis of a program. Furthermore, software diversification methods are used to make it hard to obtain the structural information of

program via running it several times. Moreover, it provides a strong protection against side channel attacks and return oriented programming (ROP) attack

In conventional runtime attacks, the attacker injects a malicious code into the memory space of a program. However, data execution prevention (DEP) is an efficient countermeasure to thwart this kind of attack, nevertheless code reuse attacks don't require code injection (Davi, et al., 2012). Furthermore, attackers can exploit the existing code pieces which residing in the memory space of the program. On the other hand, they can perform a side channel attacks relying on dynamic properties of programs such as memory latencies, execution time, or power consumption (Crane, et al., 2015). Software diversity is an efficient and highly flexible defence mechanism to thwart these types of attacks, since it denies adversaries accurate knowledge of their target via randomizing implementation features of the software. Many previous works of software diversity focus on randomizing the program representation since code reuse attack and some relevant attacks depend on static features of a program. Most of these mechanisms are work by randomizing the programs before execution, either during loading, installation or compilation time (Crane, et al., 2015). Despite of software diversification throws up barriers to attackers, it also adds complexity to software development and maintenance. On the other hand, some mechanisms may impact the performance.

Moreover, call chains obtained via stack tracing and analysis can be exploited to understand and analyse the behaviour of program. Call graph is an abstract graph form of call relationship (Xie, et al., 2015). It has a major seriousness to understand and analyse and program. Two methods can be used to analyse the call graph, static and dynamic analysis. Static analysis techniques analyse the source code of program without execute it, while the

dynamic analysis techniques trace and monitor the data of call stack during the runtime of program. Call stack provides the function call and other operations with a continuous memory block during the runtime of a program (Davi, et al., 2012).

In x86 platforms, EBP, ESP, and SS registers are used to control the call stack, in which the EBP register refers to the current stack's bottom address, while the ESP refers the current stack's top address, and the SS show the position of the stack in memory segment (Bletsch, et al., 2011). When a function calls another function, the value of EBP is pushed on the stack, and the ESP's top stack address is assigned to EBP, in which the EBP register refers to the former address of EBP frame (Xie, et al., 2016). As we can see, that the call graph reflects the execution of program which can provide a significant clue for an adversary to analyse and understand the logic of a program.

5.5.2 Overview of the Proposed Module

We proposed a tamper resistance mechanism based on code obfuscation and diversification. Our mechanism combined call graph obfuscating, stack obfuscating, diversification, memory layout obfuscating, randomisation and basic blocks reordering in order to increase the difficulties of static reverse engineering analysis and dynamic stack tracing. First, a random obfuscation table of faked functions is dynamically created. After that, modules of encoding and decoding are constructed. Finally, the addresses of return and call instructions are modified to ensure the valid execution path of program. The proposed mechanism creates dynamically a random obfuscation table that contain number of faked functions which selected randomly from a pool of faked functions, when a program calls a function, the encoding module replaces the original call instructions with jumps to a chain of

faked branching functions, then it maps the selected faked function to original call in order to preserve the semantic of the program.

The random mapping table is used for mapping the addresses of call and return instructions while they are being executed. Moreover, a complex call graph of functions calls is generated to make the obfuscated program more hard to analyse and understood by attacker due to a complex dependency of the obfuscated graph. Furthermore, the adversary can't correctly separate the chain of faked branching functions form the software, because it is embedded in complex dynamic data dependencies. The jump from the faked branching function to the following code block is indirect in which it can't statically determine the memory address of the targeted jump, however the targeted jump address can be located during execution time of program; Therefore, the diversified code module and the random obfuscation table generates a chain of function calls that cannot be determined during the runtime of program, where the obfuscation table is dynamically created for each program's instance. By this way, we increase the difficulties of both static and dynamic reverse engineering analyses of the obfuscated program.

5.5.3 Call Graph Obfuscation sub module

Call graph obfuscation sub module determines the start execution address of the original program, then a variety of faked functions blocks are combined with the original functions blocks such as *f1, f2, …, fk* based on the random obfuscation table. By this way, the dynamic stack tracing and analysis become very difficult since the call stack frames will contain the original and faked function calls as shown in figure 6.1, where the actual execution path of the original program is "*main → f3 → f6 → f8 → f10"*. As obvious that the call graph

obfuscation module transforms the program execution path in which it adds a variety of faked functions, where those functions may call another faked function or it calls an original function, which will complicate the calls flow path of the obfuscated software. For instance, the original function *f3* calls a faked function *f7*, where the faked function *f11* calls the original function *f8* in order to preserve the semantic and actual execution of the program. Consequently, the attacker will be confused since he can't determine or separate the faked functions calls from the original functions calls. Furthermore, the attacker can't determine the original execution path of the program because the obfuscated program will execute all function calls including the original and faked ones without generating any semantic or runtime errors where all paths can be executed correctly. Call graph obfuscation sub module can transform the program to a complex call graph obfuscation form, as shown in figure 5.16; however, we suggest to obfuscate only the critical functions since obfuscating all program's functions makes the generated software too suspicious, as well as, it may degrade the overall performance.

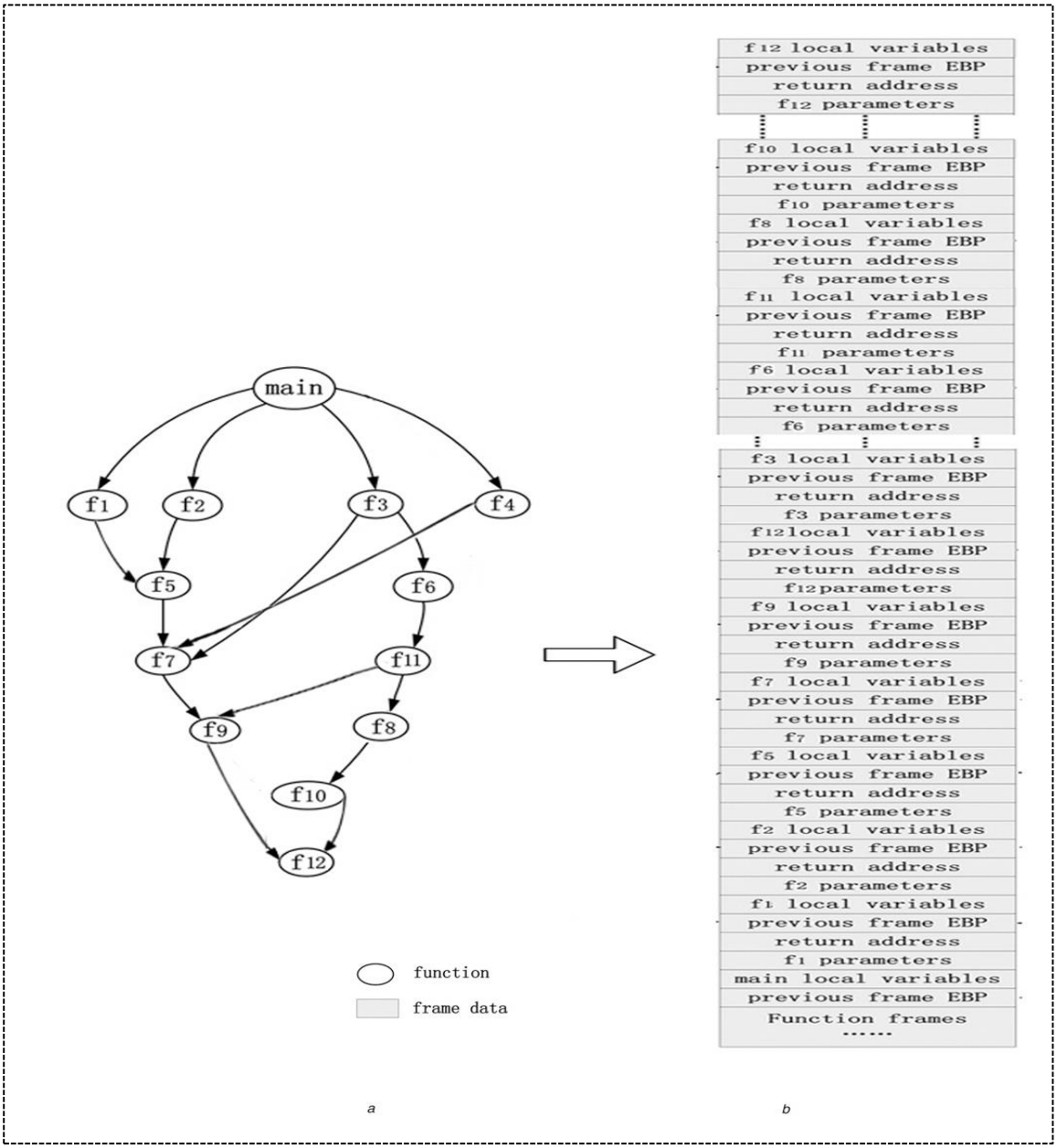


Figure 5.16*: Call graph obfuscation; (a) Call graph, (b) Call stack*

As noticed form figure 5.17, that one possible call flow path can be used to run the program correctly, while the number of other possible paths are extremely high in which they didn't implement the actual execute path of the program. Consequently, avoiding the huge combination of possible call flow paths is impossible.

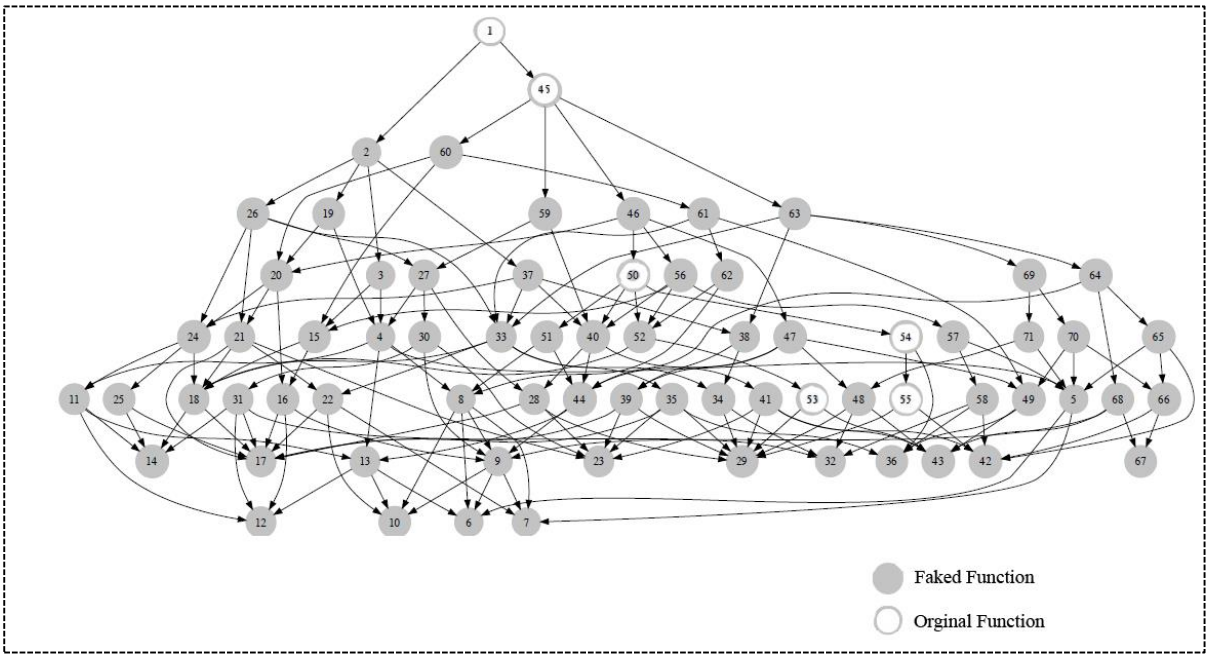


Figure 5.17*: An example of complex call graph obfuscation*

According to the basic idea, the steps of implementing the call graph obfuscation are described as follows:

(a) Analyze the dependences of the original program's functions to build their dependence graph using the dependence directed acyclic graph (DAG) algorithm.

(b) Derive the original call flow graph (CFG) of the program based on DAG algorithm.

(c) Generates the obfuscation table based on algorithm 5.5.

(d) Implements the call graph obfuscation based in algorithm 5.6.

(e) Generates the diversified code modules.

Algorithm 5.5: Obfuscation Table Creating Algorithm

INPUT: *FFP:* FAKED FUNCTION POOL OF SIZE *N; RN*: *RANDOM VALUE*
OUTPUT: *OT*: OBFUSCATION TABLE

1. K = RN MOD N // DETERMINED NUMBER OF SELECTED FAKED FUNCTIONS
2. I = 0 //INITIAL VALUE
3. WHILE I <= K
4. INDEX = RANDOM NUMBER GENERATOR () MOD N // USE TO SELECT FAKED FUNCTION RANDOMLY
5. FFS = FFP (INDEX) // FFS: THE SELECT FAKED FUNCTION FROM THE *FFP* POOL
6. DETERMINED WHETHER FFS HAS BEEN VISITED, IF VISITED GO TO STEP 4

7. OT [I] = FFS
8. I++
9. DETERMINED WHETHER LOOP REACH THE VALUE OF K, IF NOT GO TO STEP 3, ELSE ALGORITHM OUTPUTS
10. GENERATE THE OBFUSCATION TABLE OT

Algorithm 5.6: Call Graph Obfuscation Algorithm

INPUT: *OT*: OBFUSCATION TABLE OF SIZE N
OUTPUT: *OG*: OBFUSCATION GRAPH

1. M = NUMBER OF PROGRAM FUNCTIONS.
2. ANALYZE THE DEPENDENCES OF ORIGINAL FUNCTIONS CALLS
3. DERIVE THE ORIGINAL CALL FLOW GRAPH
4. FK = SELECT ONE FUNCTION FROM OT BASED ON RANDOM NUMBER GENERATOR () //FROM 1 UP TO M
5. I = 0 //INITIAL VALUE
6. WHILE I <= M
7. APPEND FAKED FUNCTION TO ORIGINAL FUNCTION
8. MAPPED ADDRESSES TO PRESERVE THE SEMANTIC OF PROGRAM
9. WHILE I < N
10. BUILD A CHAIN OF FAKED FUNCTIONS
11. APPEND SOME ORIGINAL FUNCTION TO SOME FAKED FUNCTIONS //SELECTED RANDOMLY
12. RETURN *OG*

5.5.4 Stack Obfuscation Sub Module

A variety of code blocks are being created by stack obfuscation module in order to encoding and decoding the data of call stack, as flows *E1, E2, ..., Ek* and *D1, D2, ..., Dk;* based on XOR operation and Hashing function.

First, the stack obfuscation module obtains the return and call addresses of the program's functions, then the module of address encoding and mapping are constructed based on hash function. Finally, a decoding and inverse mapping table of addresses is created dynamically.

The proposed mechanism creates dynamically a random mapping address table, when an obfuscated program calls a function the module of encoding and mapping address modifies the return addresses of the program's functions and replace them with a faked addresses selected from the random mapping address table based on hash function (see algorithm 5.8). The hash function is used to map between the obfuscated return address and the original

return address of the program. On the other hand, when the function returns, the module of decoding and inverse mapping will restore the original addresses (see algorithm 5.9). The random address mapping table make the chain of function calls cannot be determined during the runtime of the obfuscated program; therefore, the difficulties of static reverse analysis and dynamic stack tracing analysis become very difficult.

Algorithm 5.7: Hash Function Algorithm

INPUT: *S, N*: START AND END POSITION OF DATA *D*; **S*: THE DATA OF POINTER *S; ON:* ODD NUMBER
OUTPUT: *HF*: HASH FUNCTION

1. HF = 0
2. HF = ON * (*S + HF)
3. S = S+1
4. DETERMINED WHETHER THE VALUE OF *S* IS THE SAME VALUE OF *N;* IF NOT GO TO STEP 2, ELSE GO TO ALGORITHM OUTPUT

The steps of implementing the stack obfuscation are described as follows:

1. Creates the call address encoding and mapping based on Algorithm 5.8.
2. Creates the call address decoding and inverse mapping based on Algorithm 5.9.
3. Executes the control module to ensure the correct order of instructions' execution of the program based on address mapping table.
4. Implements the address mapping module to map the return address ORDi of function to new address ERDi, then it will be encoded and stored into the stack frames.
5. Implements the address inverse mapping module to decode and restored the original return address ORDi.

Algorithm 5.8: Call Address Encoding and Mapping Algorithm

INPUT: *RAMT*: RANDOM ADDRESS MAPPING TABLE; ERD $_{i-1}$: ENCODED RETURN ADDRESS OF FUNCTION $_{i-1}$ WHICH IS CALLER OF FUNCTION $_{i}$

OUTPUT: ERD $_i$: ENCODED RETURN ADDRESS OF FUNCTION $_i$ // WHICH WILL STORED IN STACK FRAME OF FUNCTION $_i$

1. SEARCH THE POSITION "*INDEX*" OF ORD $_i$ IN *RAMT* TABLE// ORD $_i$: ORIGINAL RETURN ADDRESS OF FUNCTION $_i$
2. SRA $_i$ = RAMT [INDEX]// SRA: SELECT FAKED RETURN ADDRESS FROM THE *RAMT* TABLE
3. BP $_I$ = EBP OF FUNCTION $_i$
4. IF BP $_i$ IS THE FIRST STACK FRAME OF FUNCTION $_i$
5. ERD $_i$ = HASH_FUNCTION (BP $_i$) $\oplus$ SRA $_i$
6. ELSE
7. ERD $_i$ = ERD $_i$ $\oplus$ ERD $_{i-1}$
8. RETURN ERD $_i$

Algorithm 5.9: Call Address Decoding and Inverse Mapping Algorithm

INPUT: *BP* $_i$: EBP OF FUNCTION $_i$; *RAMT*: RANDOM ADDRESS MAPPING TABLE; ERD $_{i-1}$, ERD $_i$: ENCODED DATA OF STACK FRAME OF FUNCTIONS $_{i-1}$ AND FUNCTION $_i$ RESPECTIVELY.

OUTPUT: ORD $_i$: ORIGINAL RETURN ADDRESS OF FUNCTION $_i$

1. IF BP $_i$ IS THE FIRST STACK FRAME OF FUNCTION $_i$
2. SRA $_i$ = HASH_FUNCTION (BP $_i$) $\oplus$ ERD $_i$
3. ELSE
4. ERD $_i$ = ERD $_i$ $\oplus$ ERD $_{i-1}$
5. SEARCH THE POSITION "*INDEX*" OF SRA $_i$ IN *RAMT* TABLE
6. ORD $_i$ = RAMT [INDEX]
7. RETURN ORD $_i$

5.5.5 Diversification

The proposed diversification technique works by varying the constructing of call graph obfuscation among different instances of the same program. It works by generating a different call graph for each program's instance, in which it transforms the program execution path by adding a variety of faked functions randomly and combined them with the original functions.

In order to increase the entropy and robustness of our diversification technique, we apply additional three methods as follows: splitting of independent basic blocs or functions, reordering of functions, and injection of dummy instructions. First, we split the independent basic blocs into multiple blocks and distribute them among the memory space with preserved of its execution order. For instance, block A is split in two blocks A1 and A2 as shown in figure 5.18. Before splitting the blocks, we determine the independent blocs by applying the dependence analysis using directed acyclic graph (DAG) algorithm (Muchnick, 1997). We

apply the DAG algorithm to analyse the interdependencies between instructions that can be used to build the instruction schedules and derive the basic flow graph of the program. Besides code splitting, we randomize the location of independent functions to achieve a randomized memory layout. Even if an attacker can analyse the obfuscated program, he cannot perform ROP attack because the structure and the location of functions have been randomized in memory. Additionally, if an adversary can determine the permutation and the memory layout of one instance, he cannot assume which the target device use this instance because the proposed diversification is applied for each program's instance. After that, the diversification module generates different functions and basic blocks ordering for each instance of the program. Moreover, we inject dummy code instructions and embedded them randomly into the redundant and new code section space of the obfuscated program. To preserve the semantics of the program, the dummy code will implement only nop operations. Additionally, we randomize the location of each section space to achieve a fully randomized memory layout. The proposed diversification technique thwarts ROP attack since the structure and location of all basic blocks have been randomized. Furthermore, the proposed technique is secure against disclosure attacks because all offsets among functions and diversified basic blocks have been randomly changed which obligate the adversary to revert the entire code.

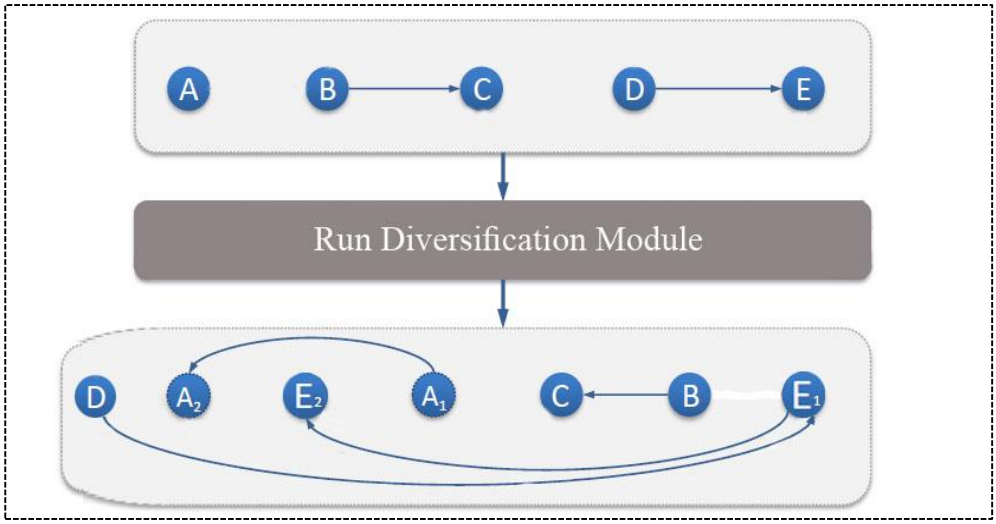


Figure 5.18:*Software diversification and reordering of functions*

CHAPTER SIX

EVALUATION & EXPERIMENTAL RESULTS

Chapter 6 : Evaluation & Experimental Results

6 Introduction

In this chapter, we evaluate the effectiveness of the proposed model against tampering, static and dynamic reverse engineering analysis. Our experimental results prove that the proposed technique provides stronger protection than other existing techniques. Moreover, the proposed obfuscation techniques can be implemented in other programming languages.

All experiments are implemented on a computer with the following specifications: Intel Core i7 2.80 GHz CPU, 16.0 GB RAM, 256 GB SSD with 64-bit Windows 10 Enterprise. Execution time and memory occupation is captured using NetBeans IDE Profiler 7.4. We used a java program to implement our experiment; source code of the tested programs are shown in Appendix.

6.1 Evaluation & Experimental Results of Proposed Static Analysis Prevention (SAP)

Our experiment proves that any attacker cannot be aware of program's data when trying to disassembly the code. In order to evaluate the proposed techniques against dissemblers, we select a portion of code from employee class shown in figure 5.5. The code we selected shown in figure 6.1.

```
PUBLIC STATIC VOID MAIN (STRING [] ARGS) {
PRIVATE STRING EMPLOYEENAME;
EMPLOYEENAME = "EMPLOYEE NAME";
SYSTEM.OUT.PRINTLN(EMPLOYEENAME); // PRINT THE VALUE ON CONSOLE
   }
```

Figure 6.1: *Portion of code from employee class (without obfuscation) (Yasin & Nassra, 2016).*

6.1.1 Experimental Results

We use Jasmin Java disassembler software which convert binary Java classes into files that are convenient for loading into a Java Virtual Machine by taking the ASCII descriptions of Java classes. The disassembly results of the code in figure 6.1, shown in figure 6.2 below:

```
_main           proc near
                push    rbp
                mov     rbp, rsp
                mov     rdi, cs:__ZNSt3__14coutE_ptr
                lea     rsi,  emplyeeName      ; " EMPLOYEE NAME"
                call    __ZNSt3__1lsINS_11char_traitsIcEEEERNS_1
                xor     eax, eax
                pop     rbp
                retn
_main           endp
```

Figure 6.2: *Disassembly results of code in figure 4.8 (Yasin & Nassra, 2016).*

As obvious from figure 6.2, that the values of the program variables detected easily after disassembly the code. The equivalent obfuscated code after applying the first and second levels of obfuscation using the proposed algorithms is shown in figure 6.3, also the disassembly results of the code in figure 6.3 is shown in figure 6.4 below:

```
public static void main (String [] args) {
private String Y_15l5$q0;                  //after obfuscate the identifier name
Y_15l5$q0 = Ob_f57_oR("t$_0olmyEepeN eaymeePsem", 3); // where “Ob_f57_oR”is the obfuscator
System.out.println(Y_15l5$q0);           // print the value on console
}
```

Figure 6.3: *Equivalent obfuscated code of the code in figure 4.8 (Yasin & Nassra, 2016).*

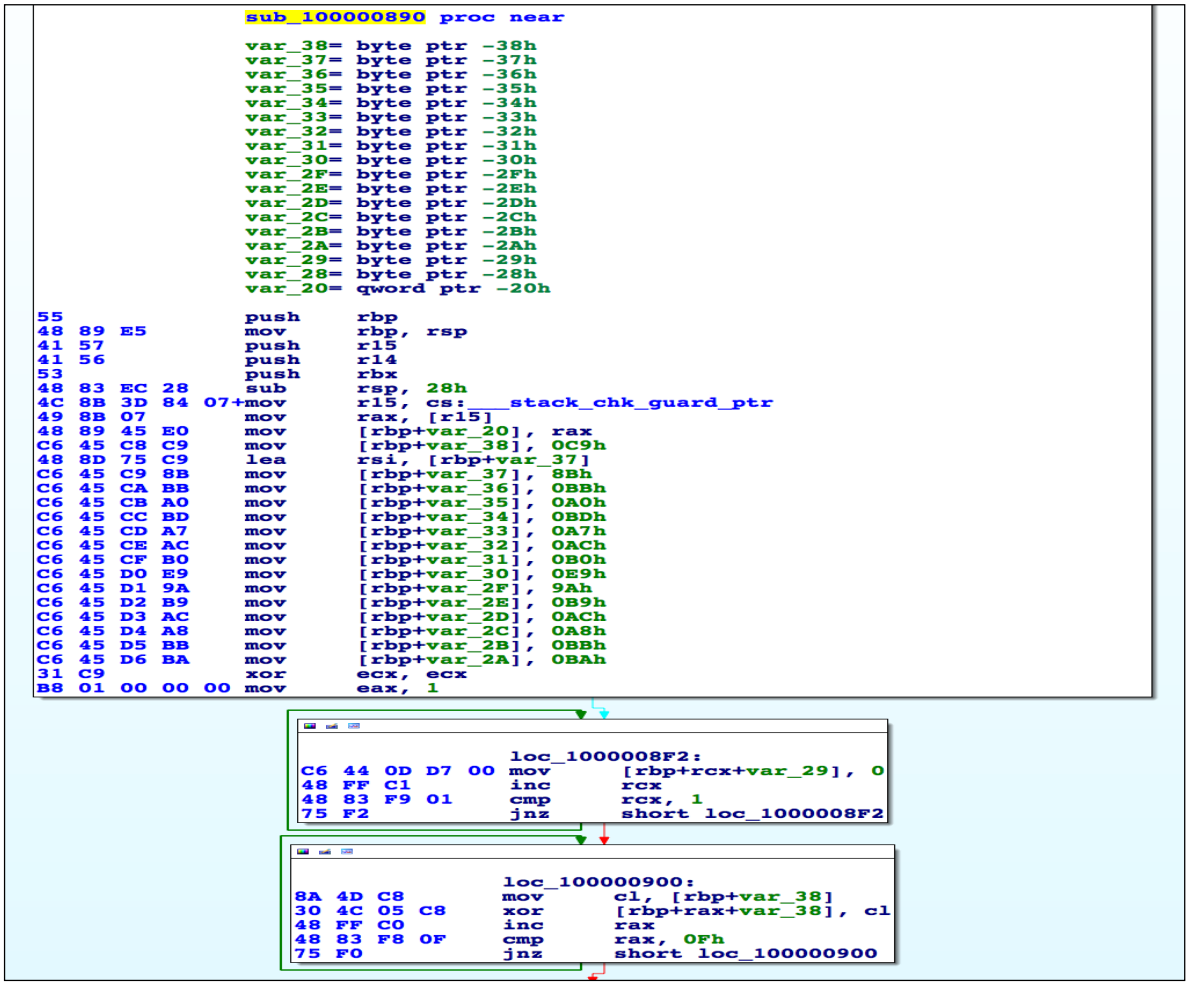


Figure 6.4: *Disassembly results of the code in figure 4.10 (Yasin & Nassra, 2016).*

As a result of implementing the proposed techniques, we notice that the disassembly of code became very difficult. In addition to that, the value of obfuscated variable can't be detected by disassembler because we replaced its value by calling a method that obfuscate its real the value by using encryption algorithm. We apply the same techniques described previously in order to obfuscate the other identifiers of an application.

In order to evaluate the proposed bytecode obfuscator, the obfuscated Employee class bytecode is tested against several available de-compilers. The results show that all decompiles are fooled by the proposed technique as the decompiled program contains

superfine bugs which it is difficult to discover. The results in Table 6.1 show that Java De-compiler Project (JD) is smarter than other de-compliers when handling keywords identifier as it can retrieve the ordinary keyword identifier automatically. Also it can replace some illegal symbols with underscore character, while the other decompiles can't handle the obfuscated identifiers, where they return the same obfuscated value similar to the keyword without any changes which will case a syntax or compilation errors. The results of testing the obfuscated code generated by our technique after applying many common Java De-compilers such as Cavaj, DJ, JBVD, and AndroChef, show that they all fooled by the illegal symbols and failed to decompile the obfuscated bytecode.

Table 6.1: *De-compilation testing results.*

	Decompiled results After using illegal symbols /*compilation error message*	**Decompiled results After using semi colon ";" /*compilation error message***	**Decompiled results After using dot "." /*compilation error message***
Original Identifier	STRING EMPLOYEENAME="EMPLOYEE NAME"	STRING EMPLOYEENAME="EMPLOYEE NAME"	STRING EMPLOYEENAME="EMPLOYEE NAME"
Obfuscated Identifiers	STRING #_1#L!Q%0 = OB_F57_OR("T$_0OLMY EEPEN EAYMEEPSEM",3)	STRING #_1;L!Q%0 = OB_F57_OR("T$_0OLMYEEPE N EAYMEEPSEM",3)	STRING #_1.L!Q%0 = OB_F57_OR("T$_0OLMYEEPE N EAYMEEPSEM",3)
JD	STRING Y_1_L_Q_0 = OB_F57_OR("T$_0OLMYE EPEN EAYMEEPSEM",3)	STRING Y_1; L_Q_0= OB_F57_OR("T$_0OLMYEEPEN EAYMEEPSEM",3) //**COMPILATION ERROR MESSAGE:** *CAN'T FIND SYMBOL : L_Q_0*	STRING Y_1.L!Q%0 = OB_F57_OR("T$_0OLMYEEPEN EAYMEEPSEM",3) //**COMPILATION ERROR MESSAGE:** *EXCEPTION ERROR*
Cavaj	*STRING Y_1#L!Q%0 = OB_F57_OR("*T$_0OLMYE EPEN EAYMEEPSEM",3*)* //**COMPILATION ERROR MSG:** *ILLEGAL CHARACTER : '#'* *<IDENTIFIER> EXPECTED* *ILLEGAL CHARACTER: '!'* *<IDENTIFIER> EXPECTED* *CAN'T FIND SYMBOL : CLASS $Q0*	STRING Y_1;L!Q%0 = OB_F57_OR("T$_0OLMYEEPEN EAYMEEPSEM",3) //**COMPILATION ERROR MSG:** *ILLEGAL CHARACTER : '#'* *<IDENTIFIER> EXPECTED* *ILLEGAL CHARACTER: '!'* *ILLEGAL CHARACTER: '%'* *CAN'T FIND SYMBOL L* *<IDENTIFIER> EXPECTED*	STRING Y_1.L!Q%0 = OB_F57_OR("T$_0OLMYEEPEN EAYMEEPSEM",3) //**COMPILATION ERROR:** *EXCEPTION ERROR*

DJ	*String Y_1#l!Q%0 = Ob_f57_oR("*T$_0olmyEepeN eaymeePsem",3*)* **//COMPILATION ERROR:** *Illegal character :'#'* *<identifier> expected* *Illegal character:'!'* *<identifier> expected* *Can't find symbol : class $Q0*	String Y_1;l!Q%0 = Ob_f57_oR("T$_0olmyEepeN eaymeePsem",3) **//COMPILATION ERROR:** *Illegal character :'#'* *<identifier> expected* *Illegal character:'!'* *Illegal character:'%'* *Can't find symbol l* *<identifier> expected*	String Y_1.l!Q%0 = Ob_f57_oR("T$_0olmyEepeN eaymeePsem",3) **//COMPILATION ERROR:** *Exception error*
JBVD	String Y_1#l!Q%0 = Ob_f57_oR("T$_0olmyEepeN eaymeePsem",3) **//COMPILATION ERROR**: *Illegal character :'#'* *<identifier> expected* *Illegal character:'!'* *<identifier> expected* *Can't find symbol : class $Q0*	String Y_1;l!Q%0 = Ob_f57_oR("T$_0olmyEepeN eaymeePsem",3) **//COMPILATION ERROR:** *Illegal character :'#'* *<identifier> expected* *Illegal character:'!'* *Illegal character:'%'* *Can't find symbol l* *<identifier> expected*	String Y_1.l!Q%0 = Ob_f57_oR("T$_0olmyEepeN eaymeePsem",3) **//COMPILATION ERROR:** *Exception error*
AndroChef	String Y_1#l!Q%0 = Ob_f57_oR("T$_0olmyEepeN eaymeePsem",3) **//COMPILATION ERROR:** *Illegal character :'#'* *<identifier> expected* *Illegal character:'!'* *<identifier> expected* *Can't find symbol : class $Q0*	String Y_1;l!Q%0 = Ob_f57_oR("T$_0olmyEepeN eaymeePsem",3) **//COMPILATION ERROR:** *Illegal character :'#'* *<identifier> expected* *Illegal character:'!'* *Illegal character:'%'* *Can't find symbol l* *<identifier> expected*	String Y_1.l!Q%0 = Ob_f57_oR("T$_0olmyEepeN eaymeePsem",3) **//COMPILATION ERROR:** *Exception error*

It is clear from the above table that JD are smarter than others when it handled illegal symbols such as *"!* , *#"*, where it replace them by underscore character "_", where the others return the same obfuscated statement as it is without any modification which cause compilation error and results an exceptions. On the other hand, all de-compliers fooled when they treated with semi colon and dot symbols, although the JD tried to handle the semi colon by dividing the variable into two different statements and replace the illegal characters with

underscore characters but it causes an exception error and failed to decompile the obfuscated code because the Java compiler cannot find declaration of variable *" l_q_0"*.

The proposed techniques confused all de-compliers and prevent them from return the original code, and therefore the professional attacker has to spend a lot of time trying to debug the code.

6.1.2 Performance Evaluation

In this section, we present the results of evaluation the obfuscated code after applying each obfuscation level. Karnick et al., (2006) describe that cost measure of code obfuscation can be evaluated using three parameters as follows:

- Storage (program size).
- Execution time (performance).
- Memory usage.

We implement our evaluation by executing the original and obfuscated programs, and then comparing them based on the previous described parameters. We noticed that the performance overhead that is introduced due to code obfuscation is negligible.

6.1.2.1 Storage (Code Size)

The size of the obfuscated program is always greater than the original one since it introduces non trivial additional lines of code. However, we think that this measure is less accurate since it's cases as a result of obfuscation. The code size of the original Employee class and the obfuscated one are compared in Table 6.2 and figure 6.5.

Table 6.2: *Storage size of original and obfuscated codes*.

Obfuscation Type	Original Code (KB)	Obfuscated Code (KB)
Source code obfuscation	5	3.7
Data obfuscation	5	10.5
Bytecode obfuscation	5	14.6

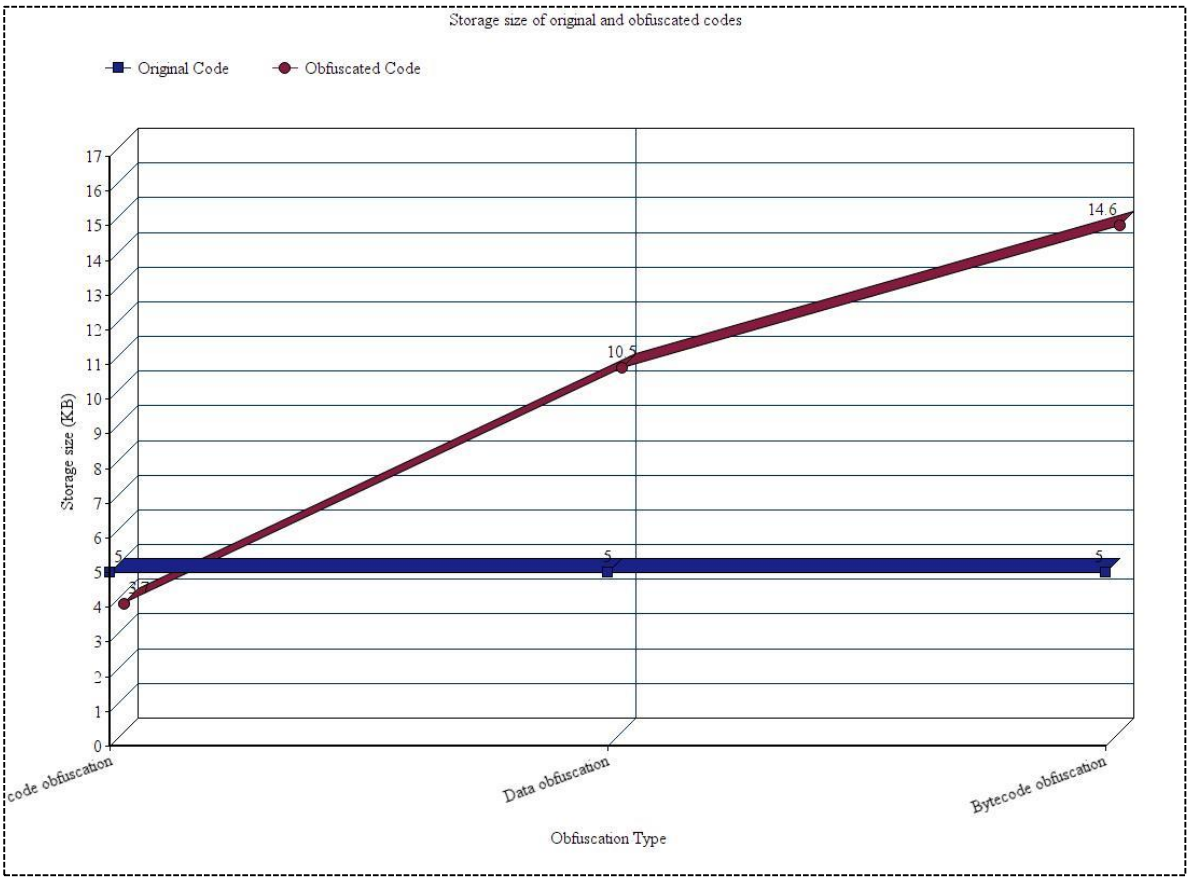


Figure 6.5 : *Evaluation of original and obfuscated code in term of Storage size*

As obvious form the table and figure above that source code obfuscation reduces the size of the obfuscated program since it removes whitespaces and indentation, strip comments and remove debug information. Other obfuscation level increase the size of obfuscated code due

to the obscurity that added to the program. The average size of the obfuscated program is about 9.6 KB.

6.1.2.2 Execution Time

The execution overhead causes by our obfuscator after being tested on all obfuscation levels is range from around 4 % to 6 %. It is obvious that our obfuscator causes an acceptable execution time. Table 6.3 and figure 6.6, present the execution time taken by original code and obfuscated code. The average execution time of the proposed obfuscator is about 1473.33 milliseconds.

Table 6.3: *Execution time of original and obfuscated codes.*

Obfuscation Type	**Original Code (ms)**	**Obfuscated Code (ms)**
Source code obfuscation	1425.8	1425.8
Data code obfuscation	1425.8	1482.83
Bytecode obfuscation	1425.8	1511.35

We notice that the execution time of the original and obfuscated programs at source code obfuscation level is almost the same. Thus, no overhead is observed after applying the source code obfuscation level. However, the overhead introduces due to dynamic decryption of obfuscated variable's values, in which the obfuscated code runs approximately twice slower than the original code because of implementing the advanced programming techniques during the decryption process.

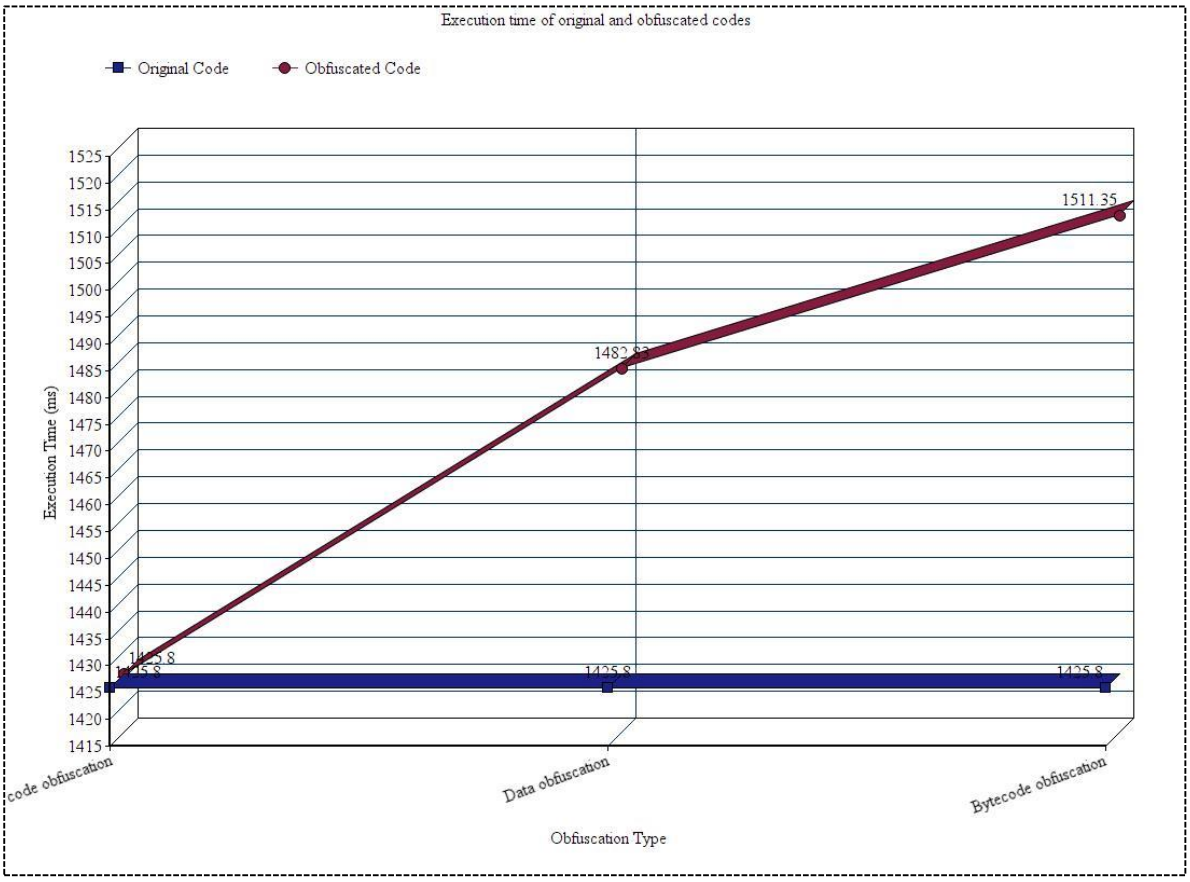


Figure 6.6 : *Evaluation of original and obfuscated code in term of Execution Time*

6.1.2.3 Memory

The differences in memory usage of original and obfuscated programs are negligible, as shown in table 6.4 and figure 6.7. Hence, the obfuscated code doesn't increase the memory requirement of the obfuscated program.

Table 6.4: *Memory usage of original and obfuscated codes.*

Obfuscation Type	**Original Code (MB)**	**Obfuscated Code (MB)**
Source code obfuscation	2	2
Data code obfuscation	2	2.3
Bytecode obfuscation	2	2.6

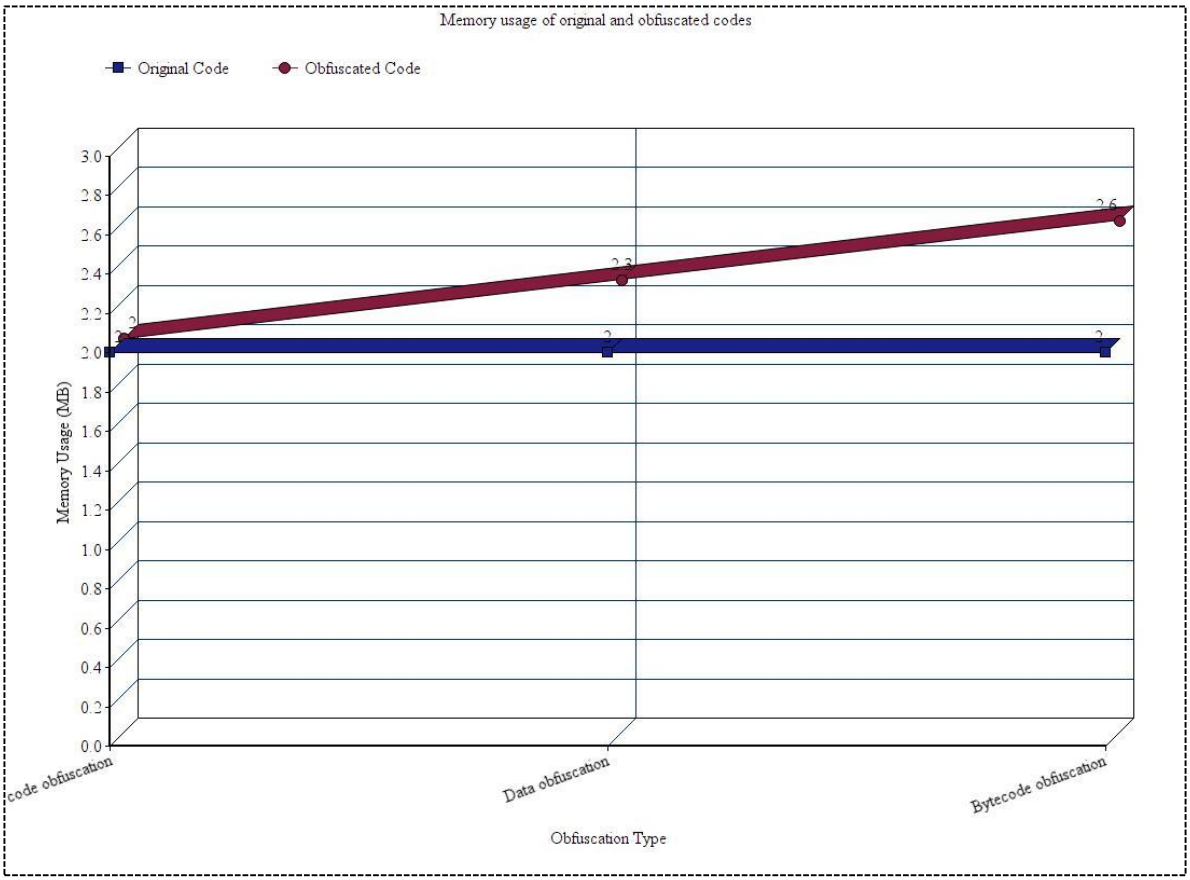


Figure 6.7 : *Evaluation of original and obfuscated code in term of Memory Usage*

6.1.3 Comparative Evaluation and Improvements

Many of obfuscation techniques are weak, since they are vulnerable to both dynamic and static analysis. Where the others are very costly to implement, since these mechanisms impose considerable performance penalties. On the other hand, other obfuscators only scramble identifier names, which we consider as a low level of protection; as well as, it doesn't work well with recent smart de-compilers.

Source code and layout obfuscation simply parsing the data structures of source code according to the language lexical rules and syntax, carry out the obfuscation, and then un-parsing them back to the original source code. In our approach, we improve such

transformation with a way that break the relationship between code statements through replacing the meaningful identifiers with nonsense names that convey no information, where the generated nonsense names should not violate the java language naming specifications or causing any compilation or syntax errors. Therefore, we did not need to un-parsing the code back to the original source code such as the tools that depend on layout transformation.

Array restructuring working only in transforming integer arrays where no string encryption added to the program. Other data obfuscators transform only one datatype such as integer or string while other datatypes are not being obfuscated, which leads to low level of obscurity. In our approach, we apply several encryption algorithms each of them dedicated for one datatype, we obfuscate string, characters, integers and decimal number such as float and double in a random manner. By this way, we complicated the process of data manipulation. Such improvement conveys most data types which give our approach more robustness compared with others.

Control flow obfuscation simply restructuring branching statements and loops to obfuscate the control flow of the program. However, altering the control flow may increase the runtime to such a drastic level that could affect the efficiency of the obfuscation. The criteria used in evaluating the quality of obfuscation depend on how much obscurity added to the program. However, the combination of control flow obfuscation with data obfuscations techniques might be a good way for an obfuscator to defy against de-compilers.

From discussion above, we see that the best way to improve the level of obfuscation is make a combination of different obfuscation techniques; therefore, in our approach we make a combination of the source code obfuscation with bytecode and data obfuscation, which make the proposed approach robust against many forms of static reverse engineering

analysis. In addition, we enhance the bytecode obfuscation by improve the way of producing illegal names that stored in bytecode in order to complicate the life of attacker. When we evaluated our approach with other relevant approaches, we see that it can defeat all common de-compilers efficiently, because it works on different obscure levels and obfuscate many datatypes while others approaches worked only on one level or one datatype. The results of comparative evaluation with other related approaches is shown in table 6.5.

Table 6.5: *Comparative evaluation with other related approaches (Yasin & Nassra, 2016).*

Approach	**Obscure levels**	**Methodology**	**Drawbacks & Limitations**
Chan, et al. [1]	Bytecode level.	Scramble the identifiers in the java bytecode.	- Increase size of bytecode. - require additional computation time. - reduce the efficiency of program.
Memon, et al. [2]	Variables and methods level.	Hide the name of the variables and methods	- Most of recent smart de-compilers can substitute these names with sequentially names and exceed this trick easily. - Cannot be applicable to all methods such as the instance method that implements an abstract method.
Balachandran, et al. [5]	Layout level.	Move and hide some of the vital source code information such as jump instruction from the original code into data segment.	Size of the program will be duplicated and increase about 2.2 of the original one.
P.Sivadasan, et al. [19]	Data level.	Hiding integer in java code using Y-factor.	Obfuscate only non-negative integer without obfuscating other types such as string.
S.Schrittwieser, et al. [24]	Control Flow level.	Split the code into small parts before diversification where the control flow graph of the software reconstructed before executing the code.	- Require additional computation time due to vast amount of inserted jumps, which will reduce the efficiency of program. - Not cover inter gadget diversification.
P.Sivadasan, et al. [22]	Array Restructuring level.	Restructuring arrays of java code.	Working only on transformation the integer arrays where no other

			datatypes encryptions added to the program.
Proposed Techniques	- Variables and methods level. - Layout level. - Data Level. - Bytecode level.	- Obfuscate the source code of the program by replaced identifiers such as variables, functions and classes names with nonsense names that convey no information. - Encrypts the values of constants, local and global program variables - Substitute the identifiers names that stored in byte code with Illegal obfuscated identifiers.	Not cover control flow level.

As noticed from the above table, that the proposed techniques are superior over other approaches, since they combined many mechanisms to increase the robustness against many forms of static reverse engineering analysis. Moreover, most dissembling and de-compilation tools cannot easily undo the obfuscation effects of our approach, as the attacker will consume a lot of time removing the bugs of the decompiled buggy program.

6.2 Evaluation & Experimental Results of Proposed Dynamic Analysis Prevention

Building software protection based on neural network will acquire the proposed scheme advantages over the traditional protection schemes as follows:

- There is no any single point of failure, because the encryption and decryption functions are embedded inside the structure of the neural network. Consequently, if incorrect input is given, the network will output unusable binary information.
- Retrieving the embedded functions from the neural network is impractical, because it is impossible to avoid the huge combination of inputs/outputs if the neural network is analyzed by the adversaries.
- Protection can be customized and make it harder as needed, via adding more neurons to the network, which will complicate the encryption and decryption process.

- Any attempts to change the neural network or the embedded encryption and decryption functions leads to unsuccessful decryption.

The effectiveness of the proposed model against memory analysis and runtime monitoring tools is evaluated, in which we assumed that the attacker can implement a memory tracing and concolic testing attack over the obfuscated software, hoping we can detect and prevent such type of attack using the neural network. On the other hand, we evaluate the proposed neural network to ensure that the adversary can't extract its embedded rules using any pattern matching tools or any neural network reverse tools. Furthermore, we evaluate the performances of the obfuscated software against brute force attack.

6.2.1 Evaluate the Performance of the Proposed Neural Network

We evaluate the performance of the proposed neural network using the testing sample that has been selected form training set, where the proposed network should correctly encrypt those inputs.

In order to calculate the error that is generated by each neuron, we used the Backpropagation Through Time (BPTT) algorithm as a learning algorithm. Sometimes it is called backward propagation since the error is computed at the output layer then redistributed back through the network layers. Practically BPTT calculates the gradient of the loss function by unrolling all inputs at each time steps. Each time step has only one input, one output, and one copy of the neural network, where the errors are calculated and accumulated for each time step. After that, the recurrent neural network is rolled up and updated the weights. Spatially, each time step of the unrolled network can be seen as an additional layer, where the internal state of the preceding time step is given as an input to the subsequent time step.

BPTT can be computationally expensive as the number of time steps increases, which require increasing the number of derivatives that are needed to update the weights. Consequently, the weights may be exploded or vanished, which may lead to slow the learning process and generating a noisy model. In order to avoid this problem, we trained the neural network until a suitable number of time steps are obtained.

During analyzing the results of the proposed network, it is noticed that the trained RNN generates output that is very close to the expected output. The expected and actual outputs of neural network are compared in figure 6.8, where it is found that the network can achieve 99.99999% accurate results. Given the neural network accuracy, it is shown that neural network is capable to encrypt the software data efficiently.

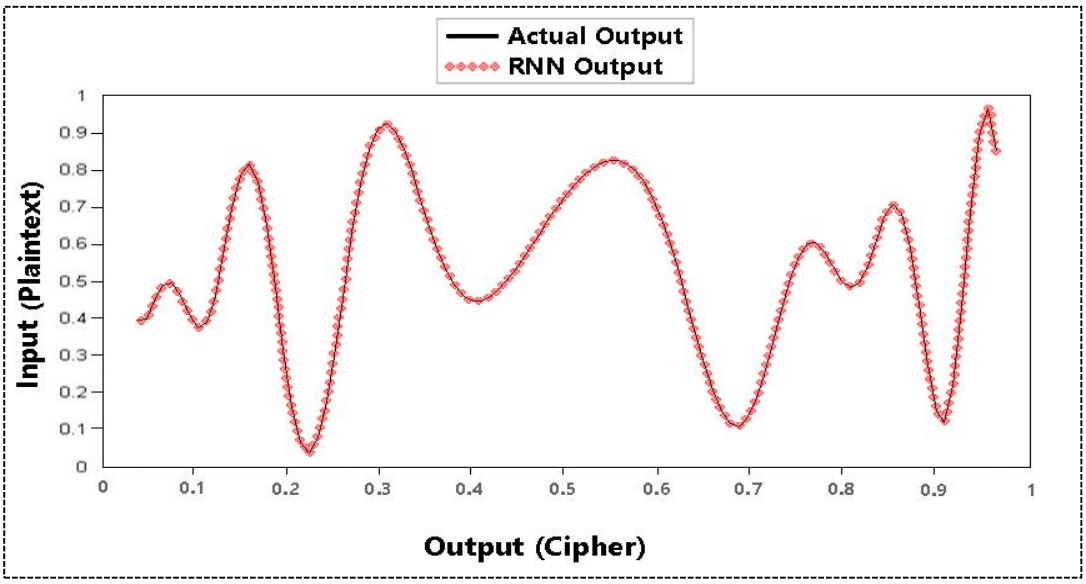


Figure 6.8: *Expected vs. Actual of Neural Network Output*

6.2.2 Evaluation Against Runtime Execution Monitoring and Memory Analysis

Our evaluation proves that any adversary cannot be aware of what happens during the run time of program, whether he is trying to monitor the execution of program or tracing the memory. In order to evaluate our technique against dissemblers, run time monitoring, and

memory tracing tools, we use the Radare2 framework, which is a reverse engineering framework with an advanced command line interface that is used to disassembling the binary files that run on various architectures. Radare2 generates a call graph of the software's functions with the extraction of the assembly code for each functions. Furthermore, it is produce a list of routines, in which each routine is a list of blocks, where each block contains a list of software's instructions, operands, offsets, and opcodes (Searles, et al., 2017).

As shown in figure 6.9 (a) that the data values, software instructions, opcodes, and memory addresses can be detected and traced easily using the disassembly framework when the code has not been obfuscated. The equivalent obfuscated code after applying the neural network encryption and the results of disassembling and tracing the execution of the obfuscated code are shown in figure 6.9 (b). As shown in figure 6.9 (b) that the static data has been obfuscated in advance using the neural network, where its value is replaced with the neural network decryption function. On the other hand, the dynamic data have been encrypted and decrypted using the neural network function during the run time of the program. Moreover, we replace the original branch logic with the neural network function "*NeuralNetwork_Dispatcher()*" to maintain the same control behavior.

As a result, our model makes the disassembly of the code very complicated. Furthermore, the value of obfuscated variables can't be detected or tracked using the de-assemblers and runtime monitoring tools because we replaced their values with a calling of the neural network function that obfuscate their real values. Moreover, tracing the memory does not show the original values while the obfuscated values are only appeared. Consequently, the neural network encrypts the data in such way that can't be restored using any disassembly or run time monitoring tools.

Figure 6.9 (a): *Original code and the results of disassembling and tracking its execution.*
Figure 6.9 (b): *Obfuscated code and the results of disassembling and tracking its execution.*

6.2.3 Evaluation Against Pattern Matching and Reverse Engineering Attack

Adversary in such type of attack, preferred to brutally testing the input and output behavior of the neural network rather than trying to extract its rules, because the available extraction

rule techniques can only explain the neural networks behavior via approximation their rules, but it cannot recover the exact input-output behavior that they represent.

The attacker tries to find the function of the neural network; therefore, our obfuscator embedded the neural network functions inside the software instructions. Thus, the neural network functions are merged with other program operations which makes it harder to be located. However, if the adversary locates the function that compute the output of the network, he cannot easily separate the network form the software correctly, because it is embedded in complex dynamic data dependencies.

In spite of the adversary can perform a path by path dynamic testing to monitor and trace the program dynamic execution and determined when the neural network function is called; however, he encounters a large number of undefined linear formulas that needed to be solved one by one. By this way, we can effectively increase the complexity of the internal structure of the obfuscated software.

Furthermore, the adversary may perform a concolic testing, even though the concolic testing is limited in solving nonlinear mathematical computations, while the neural network is typical a nonlinear model due to the activation functions that are used inside its neurons. Furthermore, the neural network is different from typical cryptographic primitives such as hashing functions, because it is not a random mapping and do not have problems like collision. Consequently, extracting rules from the neural network requires solving a complete set of constraints via reversing the neural network which empirically infeasible.

Suppose a fully connected neural network, where the output of all neurons at the current layer are passed as inputs to all other neurons at the next layer, where the neurons at the current layer compute a weighted sum values according to the activation function that is

embedded inside it. As a result, when the adversary tries to determine the corresponding inputs values of the network, given the output values, he should build an inverse system of the neural network which is extremely a hard task. Furthermore, the attacker needs to solve the inverse system under the given output values and replaces all neurons with their inverse formulas as shown in figure 6.10. Consequently, avoiding the huge combination of the reversely analyzed neural network is impossible because reversing the neural network will generate chains of undefined linear formulas, so that the number of potential solutions which needed to be solved become extremely huge.

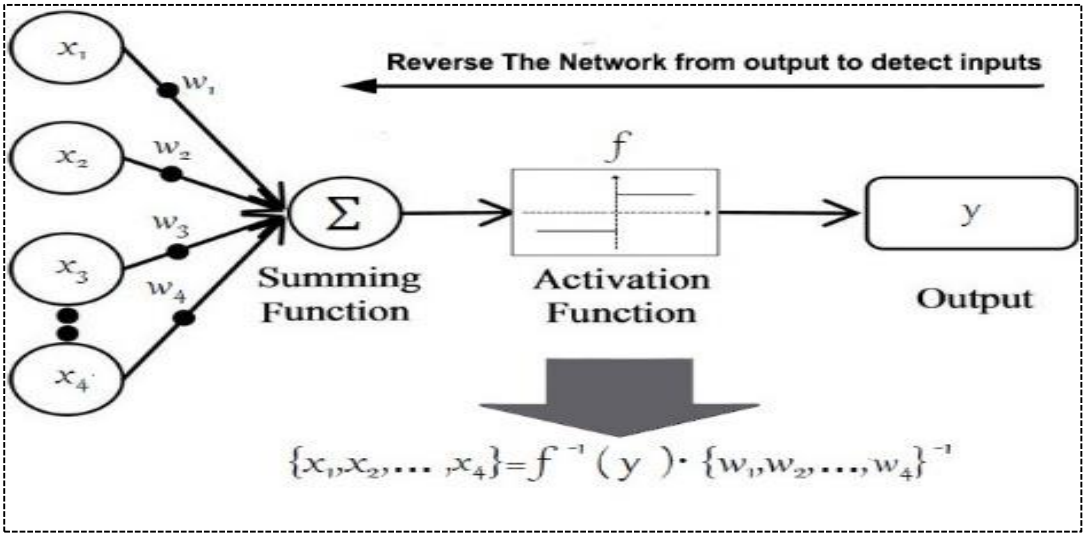


Figure 6.10: Reverse the neural network to extract its rules

6.2.4 Performance Evaluation

Embedding the neural network inside the software may case an overhead, since it increases the size of obfuscated software to be twice larger than the original one.

In order to evaluate the performance's overhead that may cause from the proposed technique, we simulated it using five benchmarks that are selected from the SPEC-2006 testing suite. As shown in Figure 6.11, the execution overhead caused by our obfuscator after being tested

on all selected benchmarks is range from around 5 % to 15 %. It is obvious that the neural network causes an acceptable execution time. Unfortunately, that neural network when constructed in dynamic way will cause a notable memory as noticed from table 6.6 and figure 6.12, which might be a small drawback while being applied in practice.

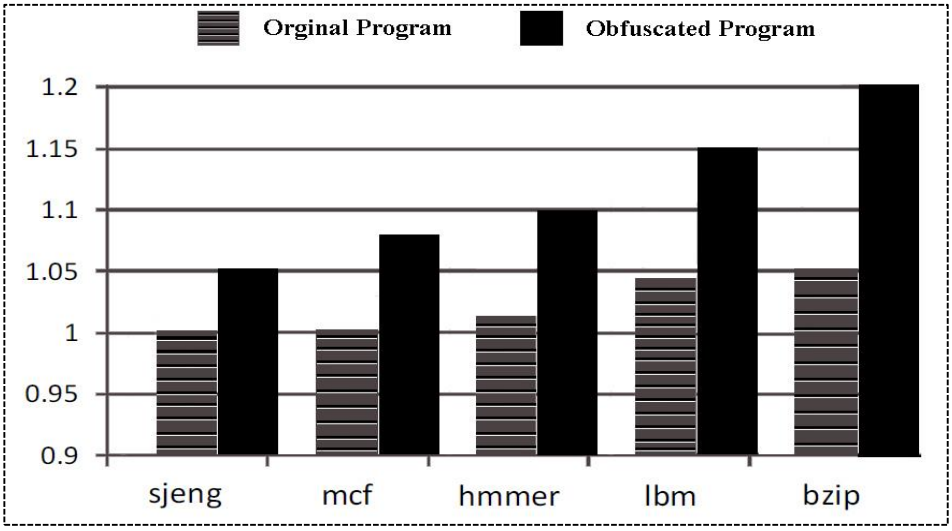


Figure 6.11: *Execution time (in seconds) of selected benchmark programs*

In order to improve the performance of the obfuscated software and avoid the overheads that my causes form the obfuscation process, we suggest to obfuscate only the most critical data since obfuscating all data makes the generated software too suspicious, as well as, degrading the overall performance.

Table 6.6: *Memory usage of original and obfuscated benchmark programs*

Benchmark	Original Program (MB)	Obfuscated Program (MB)
sjeng	180	228
mcf	190	241
hmmer	23.6	30.7
lbm	409	543
bzip	354	478

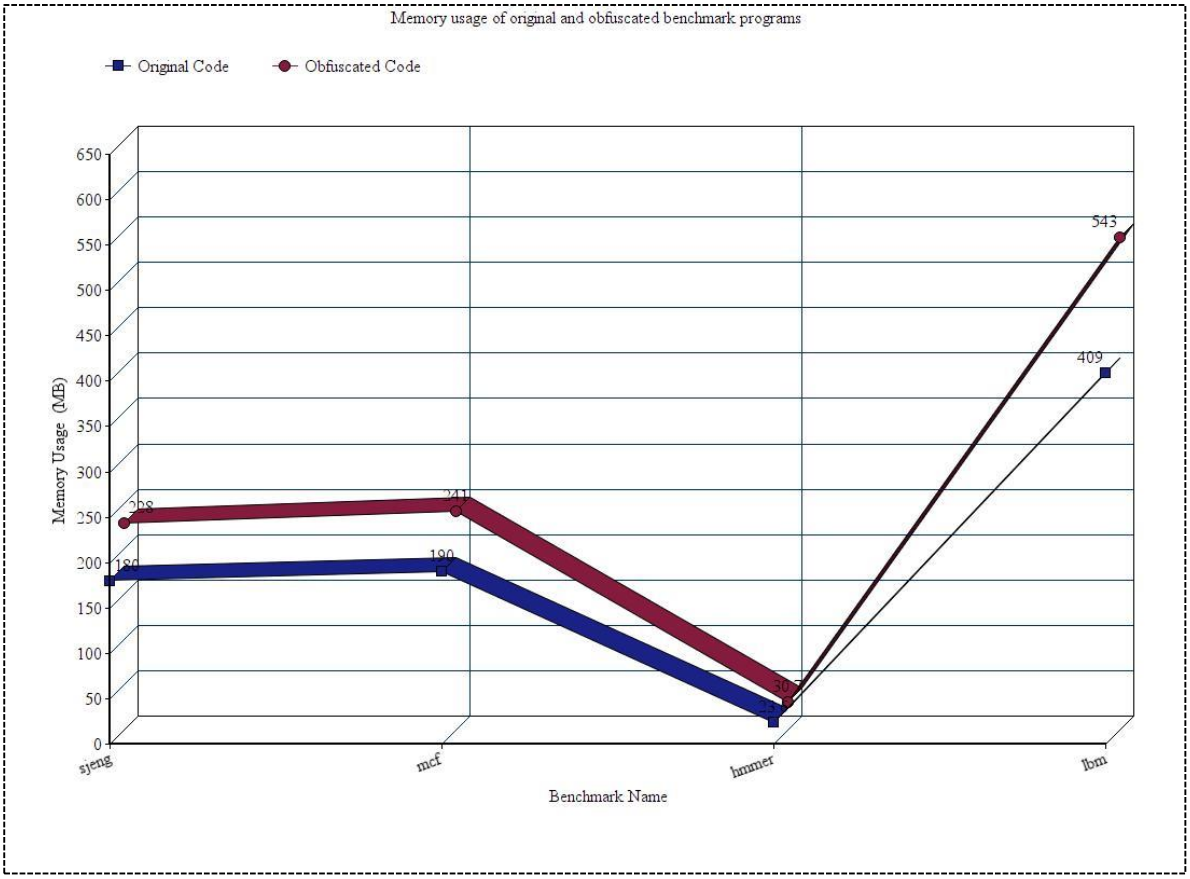


Figure 6.12 : *Evaluation of original and obfuscated benchmarks in term of Memory Usage*

6.2.5 Comparative Evaluation

A comparative evaluation with relevant techniques that apply the neural network in software protection is shown in table 6.7 and figures 6.13, 6.14. This evaluation is conducted based on two evaluation metrics: execution time and memory usage. We noticed that the obfuscated technique is a superior over other techniques in both evaluation metrics. In order to make a fair comparison, the programs that have been used in the experiments of the relevant techniques is obtained, and then it is obfuscated using our obfuscator. We compare the results after applying the proposed obfuscator on their tested program with the results that they are

obtained after applying their obfuscator on the same tested program. The source code of these programs are shown in Appendix.

Table 6.7*: Comparative evaluation with related techniques in terms of execution time and memory usage.*

Technique Name	Execution Time (ms) After applying their obfuscator	Execution Time (ms) After applying our obfuscator	Memory Usage (MB) Of their obfuscator	Memory Usage (MB) Of our obfuscator
Fingerprinting obfuscation via Neural-Network [104]	9240.54	4671.19	116.6	81.3
Control flow obfuscation using neural network [105]	12320.20	7461.26	128.6	110.7

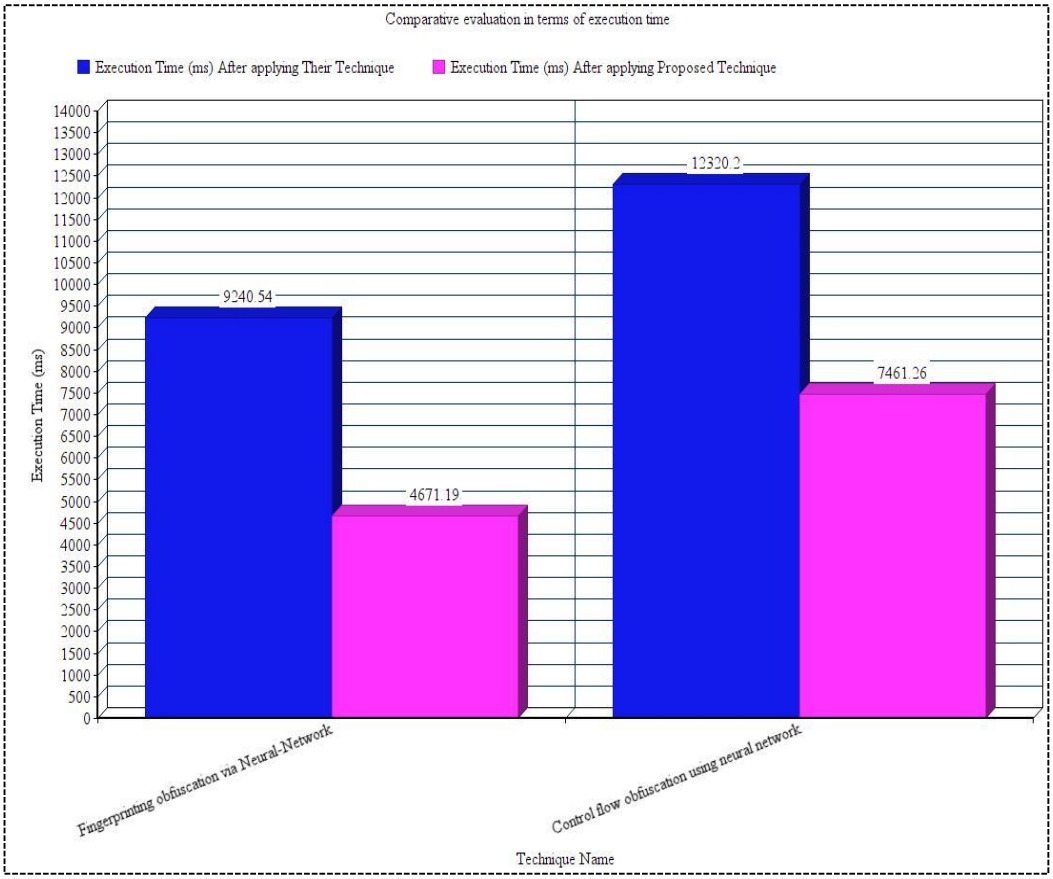


Figure 6.13 *: Comparative Evaluation with related techniques in term of Execution Time*

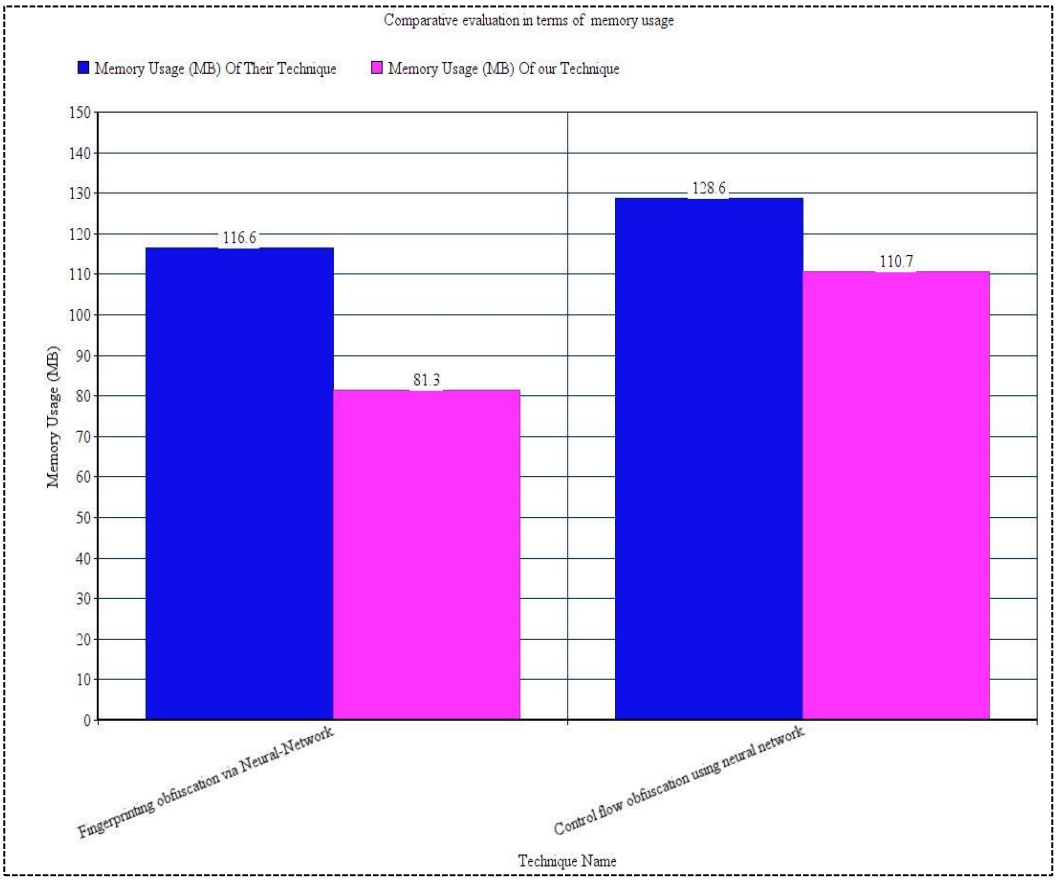


Figure 6.14 : *Comparative Evaluation with related techniques in term of Memory Usage*

6.3 Evaluation & Experimental Results of Proposed Tamper Resistance Mechanism

The effectiveness of the proposed mechanism against disassembly tools has been evaluated. Moreover, we evaluate the proposed call stack obfuscation against stack tracing analysis. Finally, we evaluate the performance of the proposed mechanism in terms of execution time and memory usage.

6.3.1 Evaluation Against Disassembly and Control Flow Analysis

Our evaluation proves that an adversary cannot determined the return and call address of the obfuscated functions calls during the runtime of the program, also the generated disassembly results of the obfuscated program is very complicated to understood. In order to

evaluate the proposed mechanism against dissembling tools, we use the IDA disassembly tool, which is a multi-processor disassembler and debugger tool which produces an assembly language from a machine executable code. Furthermore, it produces a list of software instructions, operands, offsets, opcodes, and memory addresses. As shown in figure 6.15 (b), that implementing of our obfuscation reduces the recognition rate of functions in which the value of return address can't be detected or tracked due to obfuscation of call stack data. Moreover, the disassembly of code became very complicated which increase the difficulties of both static and dynamic reverse engineering analysis in a certain extent. Consequently, the attacker can't determine the original execution path of program because the obfuscated program will execute all function calls including the original and faked ones without generating any semantic or runtime errors.

Furthermore, one possible call flow path can be used to run the program correctly, while the number of other possible paths are extremely high where they didn't implement the actual execute flow path of the program. Consequently, avoiding the huge combination of possible call flow paths is impossible. however, the attacker encounters a large number of undetermined possible paths that needed to be traced one by one. By this way, we can effectively increase the complexity of the internal structure of the obfuscated program.

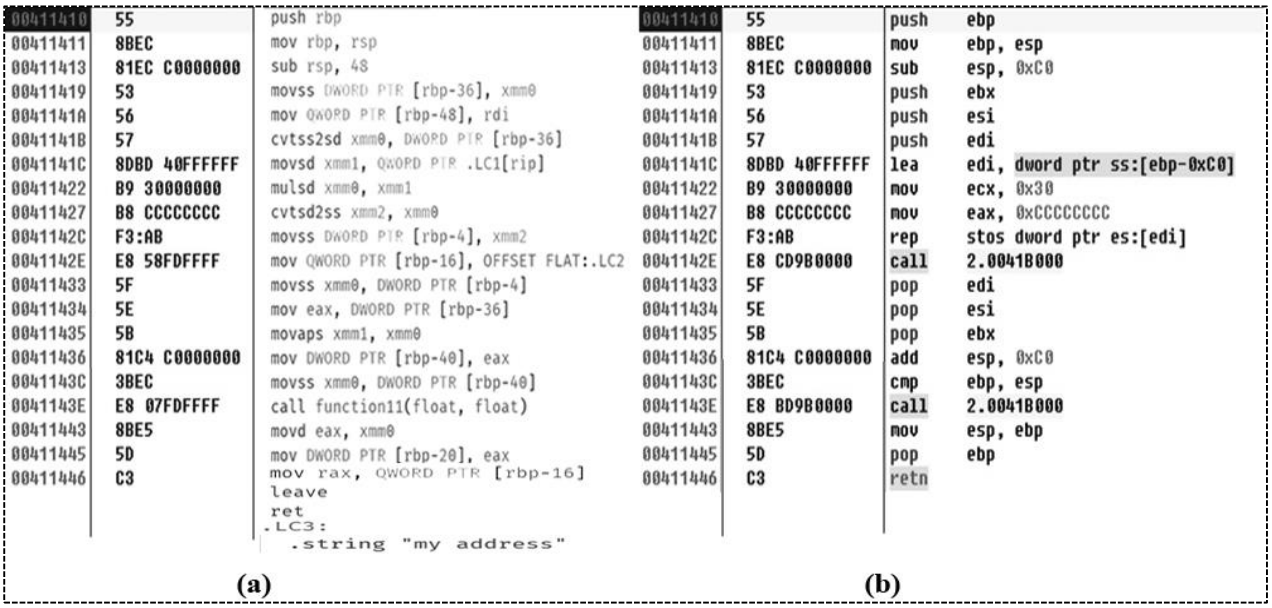


Figure 6.15*: (a) Disassembly results of original program; (b) Disassembly results of obfuscated program*

6.3.2 Evaluation Against Stack Tracing and Analysis

In order to evaluate our mechanism against stack tracing, we use the dynamic debugging tool "Ollydbg", which is an x86 debugger and assembler level analysing tool used for reverse engineering of programs. Furthermore, its used by crackers to crack the software and trace the registers, API calls, recognize function and procedures address space, and locates routines from binary files and objects (OllyDbg, January, 2018).

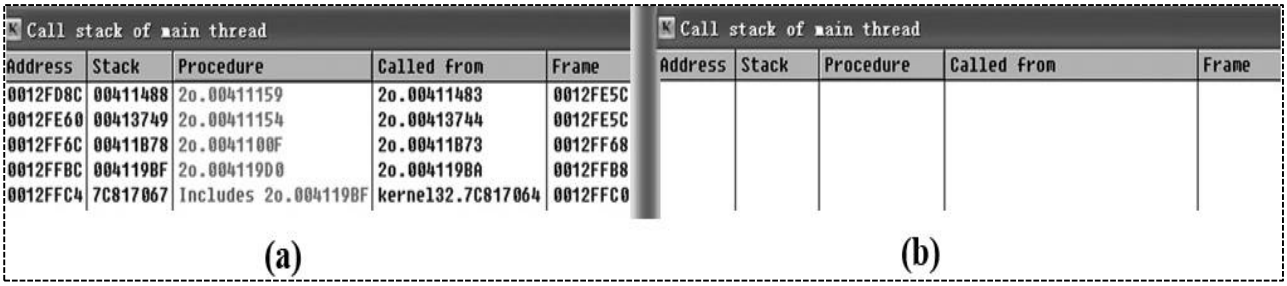


Figure 6.16*: (a) Stack tracing of original program; (b) Stack tracing of obfuscated program*

As shown in figure 6.16 (b), call chain information can't be obtained by stack tracing and analysis where the attacker can't obtain the statistical information from the stack traces and

analysis, as well as the chain of function call cannot be determined during the runtime of the program. Additionally, the recognition rate of function is reduce as shown in figure 6.17.

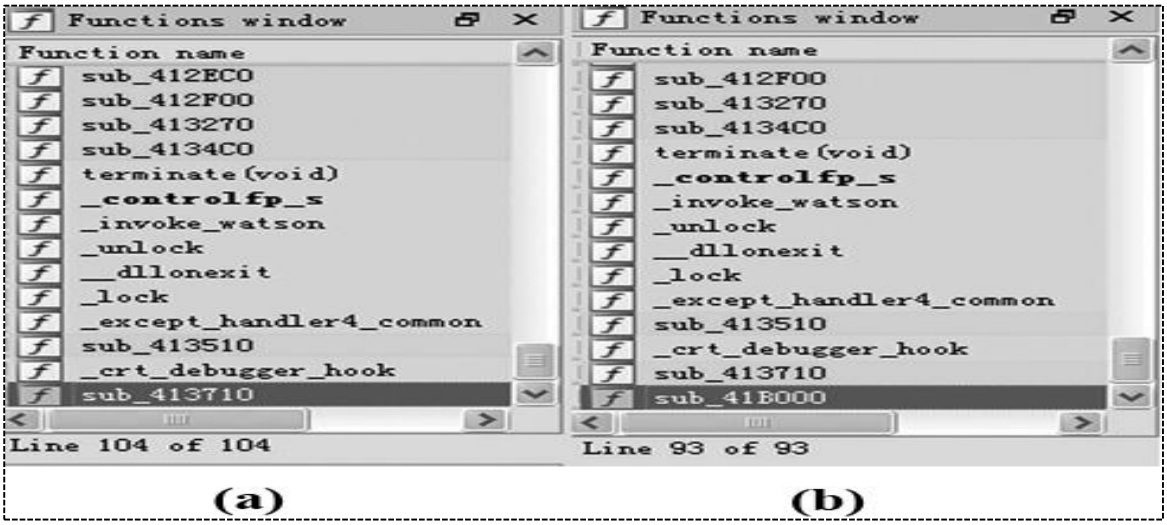


Figure 6.17*: (a) Function recognition of original program; (b) Function recognition of obfuscated program*

Consequently, the proposed call stack obfuscation hinders stack tracing in an efficient way, and also protect the integrity of call stack data, and increase the difficulties of both static reverse engineering analysis and dynamic stack tracing analysis.

6.3.3 Performance Evaluation

Collberg et al. (1997) presented a methodology to evaluate and measure the obfuscation transformations. The quality of any obfuscating transformation can be evaluated according to the following criteria: how difficult for an automatic de-obfuscator to break the obfuscation (resilience), how well the obfuscated transformation integrates with the rest of the program (stealth), how much obscurity level adds to the program (potency), and how much computational overhead it adds to the obfuscated program (the cost) (Collberg et al. ,1997).

The authors of this thesis define four metrics to evaluate the quality of the obfuscated transformation (T):

1. **Resilience:** It evaluates and measures how an obfuscated transformation (T) can resist the automated de-obfuscation tools.
2. **Potency:** It evaluates and measures to which extent a transformation T modifies the complexity of a program P. This metric can be used based on other program complexity metrics such as McCabe's cyclomatic (McCabe, 1976).
3. **Stealth**: it evaluates and measures how well the obfuscation introduced by transformation T integrates with the rest of the program P.
4. **Execution Cost:** is the space or the time penalties which a transformation T introduces to a program P. Subsequently, a metric quality of a transformation T is defined as a combination of the previous metrics to express how suitable a transformation T is *(Cappaert, 2012)*:

$$Tquality\ (P) = (Tpotency\ (P), Tresilience\ (P), Tstealth\ (P)) / Tcost\ (P) \quad (6.1)$$

We select potency, resilience, stealth, and execution cost measures in order to evaluate the effectiveness of the proposed tamper resistance technique, while we use the resilience and the execution cost measures in order to evaluate the other proposed techniques. The higher potency, resilience, stealth, and lower execution cost is the better obfuscation.

- ***Potency:***

It can be described as how much obscurity T adds to program P.

Let Pot(P) is the potency measurement of P and Pot(T(P)) is the potency measurement of T(P) then:

$$Transformation\ Potency\ (TPot) = Pot(T(P))/Pot(P) - 1 \quad (6.2)$$

Potency for both P and T(P) can be measured by various software complexity metrics such as: Program Length, Cyclomatic Complexity, and Nested Complexity. In this thesis we used the Cyclomatic Complexity to measure the potency of the obfuscated program.

The obfuscated call graph and the obfuscated stack data values are calculated randomly based on either random obfuscation table or random hash mapping table. Consequently, there is a low likelihood for an adversary to build the same collision values. The complexity of the obfuscated program is demonstrated in recursive disassembly algorithm, therefor a call graph can be constructed as a tree structure when the obfuscated program is disassembling using a recursive traversal tool. Furthermore, the execution time of constructed tree is O(m), where m is the number of the constructed tree nodes. In program, nodes denote call instructions; consequently, the call graph obfuscation complexity is O(m).

Software obfuscation complexity can be defined as: the degree to which the obfuscation components which are embedded into the software make it difficult to understand and verify (Sharma & Kushwaha, 2010). Increasing the complexity in any software gives the software owner more benefits to reduce the risk of introducing malicious instructions into the software. Furthermore, it preserves the value of the software asset, prolong its useful lifetime, and protecting intellectual property against tampering.

There are many methods for measuring the software complexity, where many of them are language independent. The basis of all these metrics lies in the capacity and limitations of the human mind to engage in logical processing and understanding the software.

In order to measure the complexity of our obfuscation implementation we used Cyclomatic complexity metric. Programs with more control and call flow are more difficult

to understand, therefore implementing of our call graph obfuscation will increase the complexity of the internal structure of the obfuscated software and make it too hard to analyse by reverse engineers where it increases the time and efforts that are needed to understand it. The cyclomatic complexity *CC* of the proposed call graph obfuscating *G* is computed by using the following formula:

$$CC(G) = E - N + P \quad (6.3)$$

Where E = the number of connections, N = the number of nodes, and P=number of processor or threads.

- ***Resilience:***

It evaluates and measures how an obfuscated transformation (T) can resist the automated de-obfuscation tools. Resilience can be measured by summing the total of programmer's effort and de-obfuscator's effort.

If P cannot be constructed from T(P), means some information from P is removed in T(P) at the time of obfuscation, then:

$$Transformation\ Resilience\ (TRes) = OneWay \quad (6.4)$$

Otherwise

$$Transformation\ Resilience\ (TRes) = Res(PEff + DeoEff) \quad (6.5)$$

Where PEff = Programmer Effort (The amount of time require by the programmer to build the automatic de-obfuscator to regenerate P from T(P).)
And DeoEff = De-obfuscator Effort (The amount of execution time and space required for the automated de-obfuscator to de-obfuscate the obfuscated program.)

The adversary can attack the obfuscated program in different ways such as: tracing based on debug points attack, locating of obfuscating functions attack, and an application programming interface (API) hijack.

In tracing based on debug points attack, attackers may assign some debug points on the obfuscated program, and then analyse and trace the function call chains. In order to prevent this type of attack, the encoding and decoding modules, address mapping, address inverse mapping, and diversified code module should embed randomly into redundant and new code section of the obfuscated program. If an adversary uses instruction tracing to get the chain of function calls, he should analyse all modules which increase the difficulty of adversary to understand and analyse the program.

In locating of obfuscating functions attack, the adversary may locate and analyse the address of the original functions and determined the semantics of the function, which may enable the attacker to construct the obfuscation table (OT), Therefore we obfuscate the original and faked functions by replacing their names with nonsense names that convey no information, which complicates the statistical analysis of the source code and increase the difficulty of adversary to understand and analyse those functions. In API hijack, some API calls can be captured by API hijack methods, to prevent this type of attack, the API calls should be obfuscated likewise.

- ***Stealth***

It evaluates and measures how well the obfuscation introduced by transformation T integrates with the rest of the program P. In the proposed obfuscation mechanism, the target call instructions' addresses are changed to entry addresses of encoding and mapping

module. When the function call is completed, the address decoding and inverse mapping will restore the original addresses. Additionally, the entry address of decoding and inverse mapping is pushed on the stack, where they control the execution to return the addresses of the original call instructions. Moreover, the proposed basic block reordering will improve the stealth of the obfuscated program and increase the difficulties of reverse engineering analysis.

- **Execution Cost**

The execution time of obfuscated program will be increased since the total number of calls instructions increases. Assume that the execution time of the original instructions is donated by t_o, and the execution time of the obfuscated instructions is donated by t_b, while the execution time of mapping and inverse mapping of stack data obfuscation is donated by t_d, and number of call instructions is donated by n; consequently, the execution time of obfuscated program is computed by using the following formula:

$$Execution\ time = t_o + n \times (t_b + t_d) \qquad (3)$$

In order to evaluate the performance's overhead that may cause from the proposed mechanism, we evaluate it using four search tree programs. A Binary Search Tree (BST), Breadth First Search (BFS), Depth First Search (DFS), and Greedy Search are obfuscated, the source code of the four programs are shown in Appendix. The performance of original and obfuscated programs is compared in table 6.8 and figures 6.18, 6.19. As obvious from table that the proposed mechanism causes an acceptable execution time, however it causes a notable memory occupation, which may consider as a small drawback especially when being applied in practice.

Table 6.8: *Performance of original and obfuscated programs*

Program	Execution Time (sec) (Original Program)	Execution Time (sec) (Obfuscated Program)	Memory(MB) (Original Program)	Memory(MB) (Obfuscated Program)
BST	3	3.3	53.8242	64.5606
BFS	3	4.8	54.2422	68.0297
DFS	2.6	4.6	54.6484	63.6210
Greedy	3	4.4	53.3242	71.6415

In order to improve the performance of the obfuscated software and avoid the overhead that my causes form obfuscation process, we suggest obfuscating only the critical functions since obfuscating all functions makes the generated software too suspicious, as well as, degrading the overall performance.

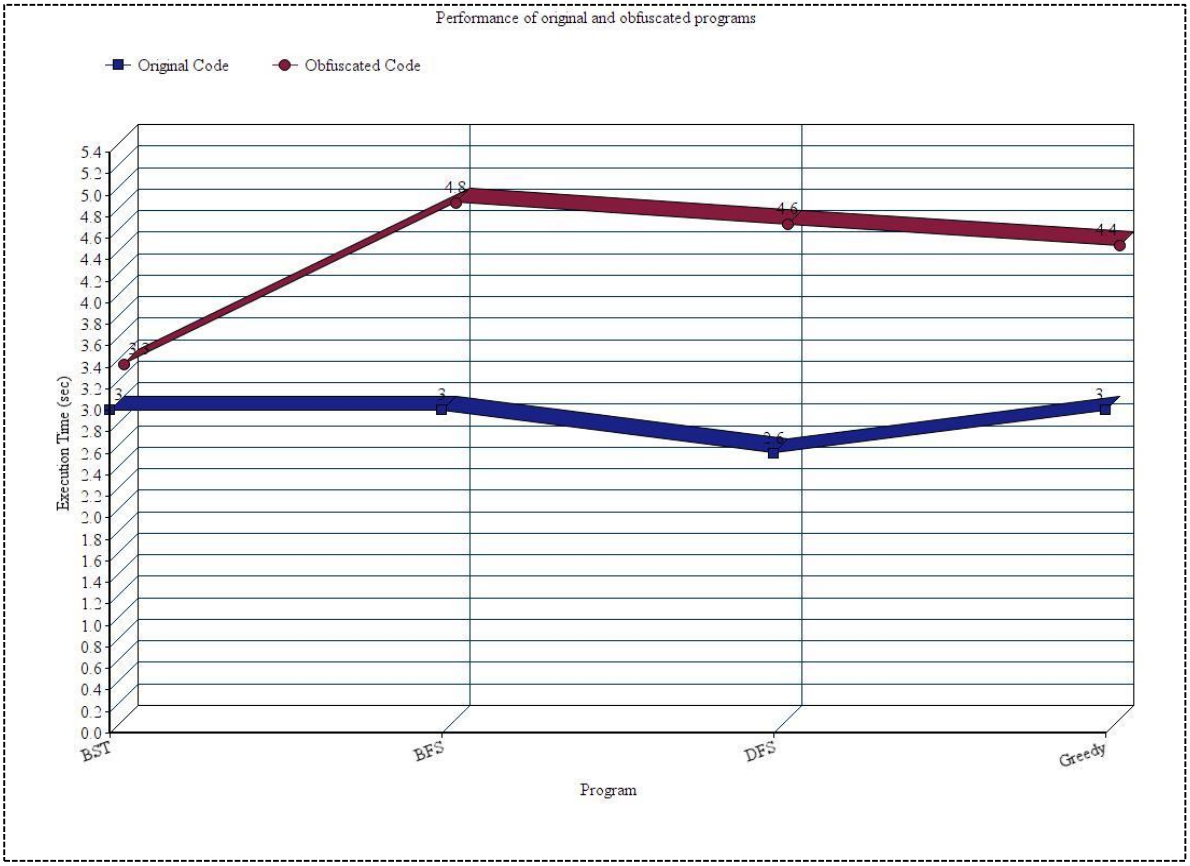


Figure 6.18: *: Evaluation of original and obfuscated programs in term of Execution Time*

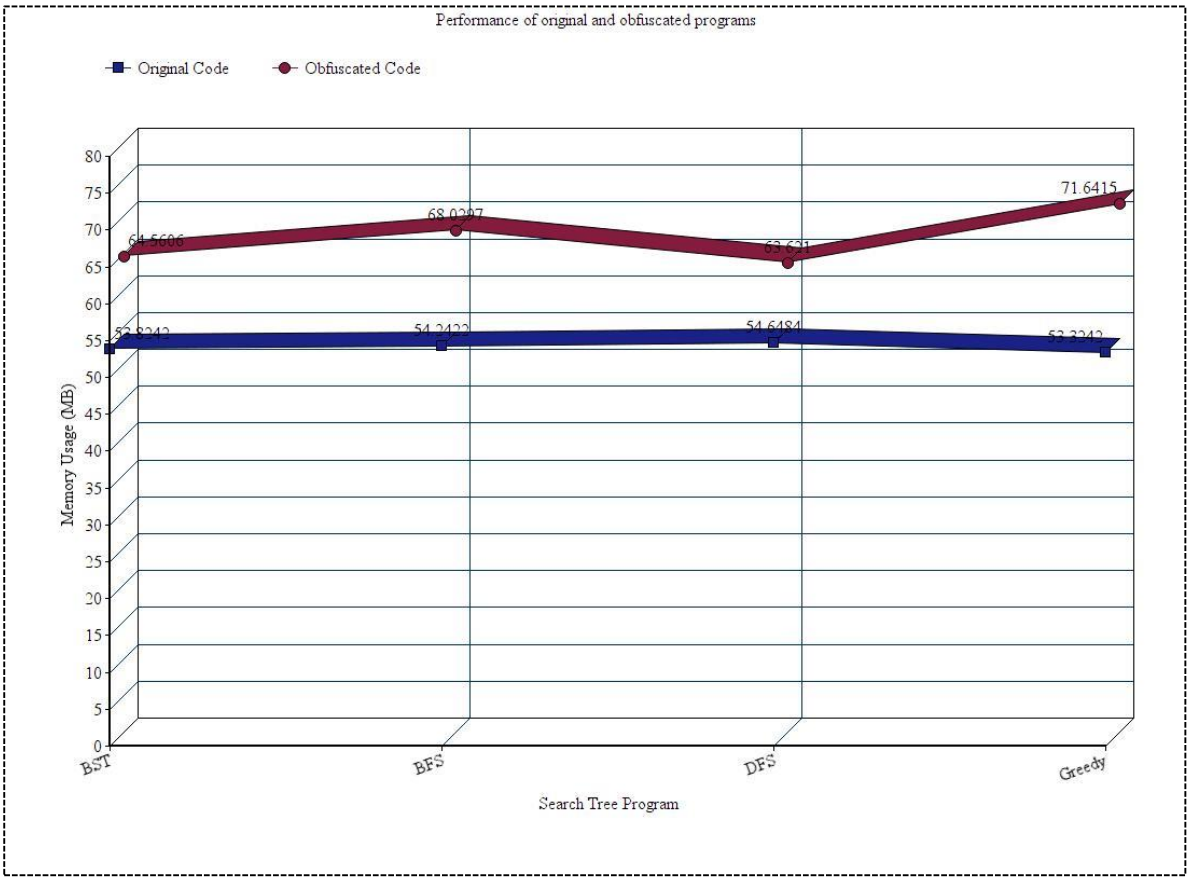


Figure 6.19 : *Evaluation of original and obfuscated programs in term of Memory Usage*

6.3.4 Comparative Evaluation

A comparative evaluation with relevant techniques is shown in table 6.2 and figures 6.20, 6.21. This evaluation is conducted based on two evaluation metrics: execution time and memory usage. We noticed that the obfuscated technique is a superior over other techniques in both evaluation metrics. In order to make a fair comparison, the programs that have been used in the experiments of the relevant techniques is obtained, and then it is obfuscated using our obfuscator. We compare the results after applying the proposed obfuscator on their tested program with the results that they are obtained after applying their obfuscator on the same tested program. The source code of these programs are shown in Appendix.

Table 6.9*: Comparative evaluation with related algorithms in terms of execution time and memory usage.*

Technique/Algorithm	Execution Time (s) After applying their obfuscator	Execution Time (s) After applying our obfuscator	Memory Usage (MB) Of their obfuscator	Memory Usage (MB) Of our obfuscator
Nirvana [13]	6.89	4.75	30.0	28.3
Graph Approach of Control-Flow Obfuscating [154]	6.20	4.32	32.86	30.66
Dynamic CFG Construction Algorithm [172]	4.7	3.14	43.0	41.48
Call address Obfuscation Algorithm [168]	7.70	5.34	45.4	43.89
Diversified Instruction Fragments Generation Algorithm [167]	6.33	5.1	52.3	50.57

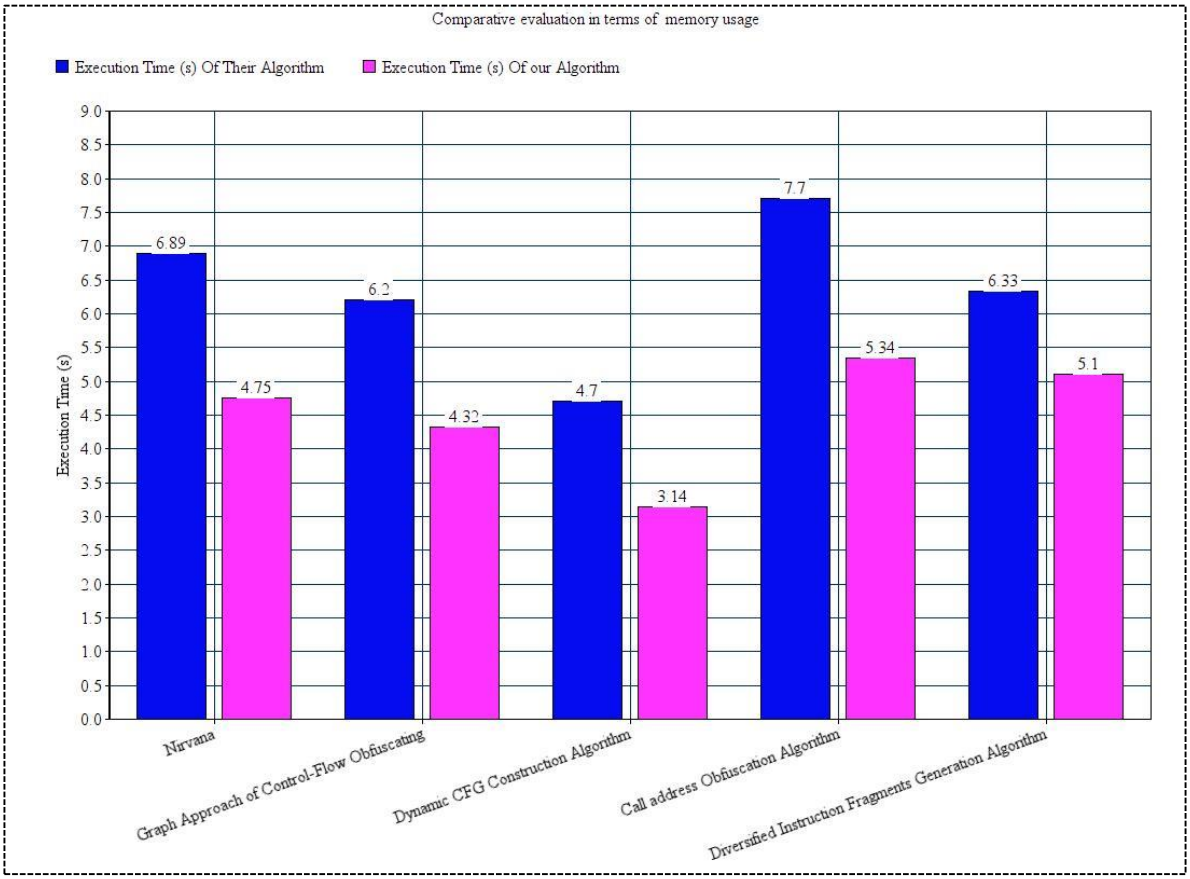


Figure 6.20 : *Comparative evaluation with related algorithms in terms of execution time*

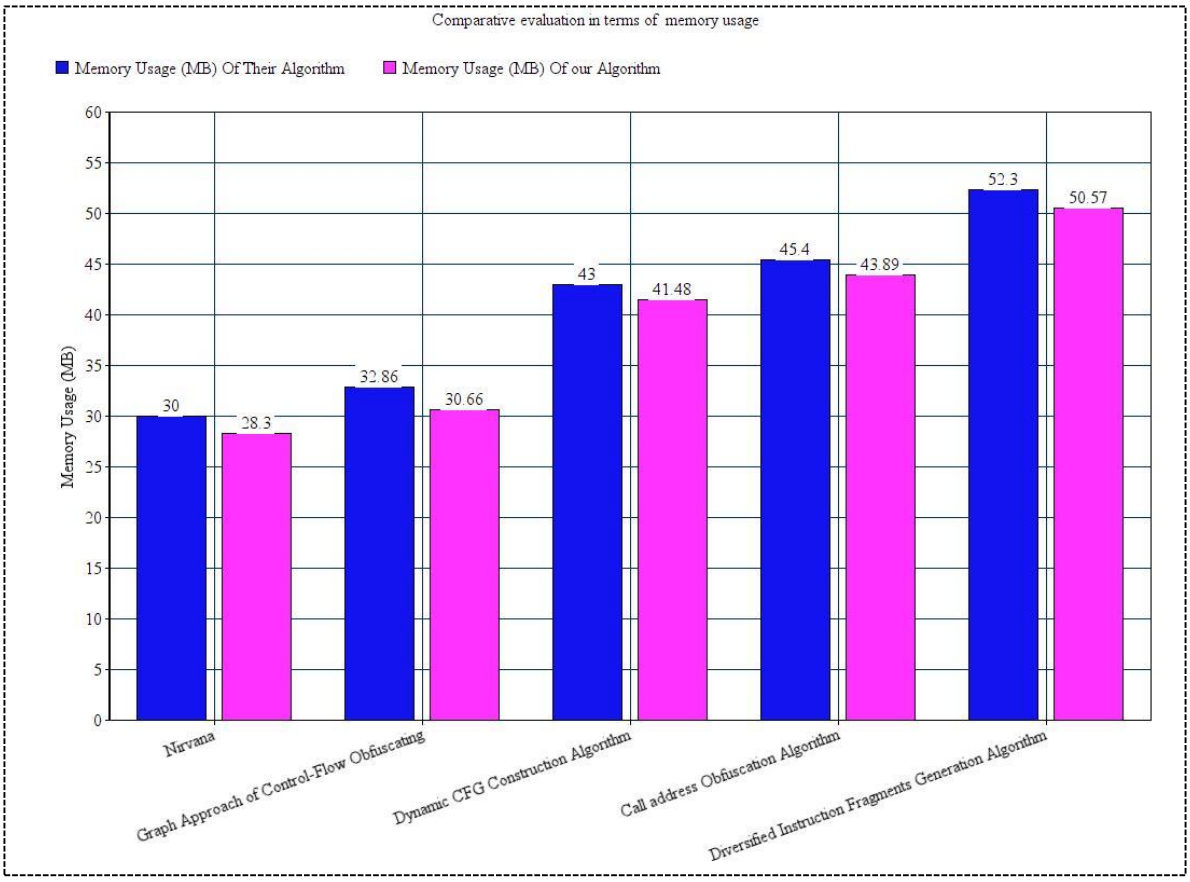


Figure 6.21 : *Comparative evaluation with related algorithms in terms of memory usage*

CHAPTER SEVEN
CONCLUSION AND FUTURE WORK

Chapter 7 : Conclusions and Future Work

Software protection provides an efficient mean of securing intellectual property, where the attackers struggle every new technique to analyze the software. Despite numerous number of protection techniques, programs are still suffering from tampering and analysis, where an illegal access could be obtained when the software is being tampered. Moreover, reverse engineering remains a considerable threat to software developers and security experts. Many efforts have been introduced to address this issue, but most of them are failed since they are vulnerable to both static and dynamic analysis. Furthermore, many of available software protection techniques are often not good as they rely on “security through obscurity”, where these techniques may deter some impatient adversaries, but against a dedicated adversary they offer little to no security.

In this thesis, we proposed a software protection techniques based on code obfuscation to protect software against tampering and reverse engineering analysis. First, a combination of many obfuscation techniques to protect software against static analysis is proposed, where these techniques integrate three levels of obfuscation; source code, data transformation, and bytecode transformation level. By combining these levels, we achieved immune resistance against many forms of static reverse engineering analysis. Second, an obfuscating technique based on integrating encryption mechanism within recurrent neural network (RNN) is proposed to enhance the software protection level against dynamic analysis, where the system designed to enable the neural network generation of different encryptions for the same protected data. This creates a many to one relationship between the keys and the encryption. Moreover, we replace the decryption function of the encrypted data with a semantically

equivalent neural network in order to complicate the reverse engineering analysis of the software. Third, we proposed a tamper resistance mechanism based on obfuscation and diversification. The proposed mechanism combined call graph obfuscating, stack obfuscating, diversification, memory layout obfuscating, randomization, and basic blocks reordering in order to thwart tampering and increase the difficulties of static reverse analysis and dynamic stack tracing analysis. A random mapping table is used for mapping the addresses of call and return instructions during the runtime of program. Moreover, a complex call graph of functions is generated to make the obfuscated program harder to attacker's analyses and understanding due to a complex dependency of the obfuscated graph.

The power of the proposed framework come from the combination of many obfuscation techniques used to build it; which makes it immune against many forms of reverse engineering analysis and tampering. The proposed framework satisfies all levels of software protection including the static analysis, dynamic analysis and tampering. Thus, by combining these levels, we achieved a high level of code confusion, which makes the understanding or analyzing the programs very complex or infeasible. Most analysis tools cannot easily undo the obfuscation effects of our techniques, as the attacker will consume a lot of time removing the bugs of the decompiled buggy program. Moreover, all common decompiles are deceived by the proposed obfuscation techniques.

Furthermore, Neural network provides a robust security characteristic in software protection, due to its ability of representing nonlinear algorithms with powerful computational capability. Moreover, understanding the operations of neural network is complicated as the internal knowledge is embedded in a complicated, self-contradictory, and distributed structure. The potency of the protected software relies on the fact that the

encryption and decryption functions are embedded in the structure of the RNN itself and on the assumption that the complexity of understanding the rules embedded in the neural network will provide a robust resistance against dynamic reverse engineering analysis that attempt to analyze the embedded logic of the obfuscated software routines. Moreover, the neural network is a non-human readable structure, in which many neurons could be used to protect a single part of code. The evaluations of this technique confirm that employing neural networks in software protection will significantly increase the difficulties in revealing the obfuscated software, whether using a pattern matching, implementing a reverse engineering analysis, running a concolic testing, or performing a brute force attack.

Our experimental results prove that the proposed techniques provide a stronger software protection than other existing techniques and it is immune against both static and dynamic analysis. Moreover, Evaluation results confirm that the proposed techniques significantly increase the difficulties in revealing the obfuscated software, whether using disassemblers, de-compilers, reverse engineering tools, tracing the memory, or implementing a stack tracing analysis. On the other hand, the performance evaluation confirms that our techniques protect software efficiently with an acceptable excess in execution time and memory usage.

Future Work

Future researches may be conducted on binary level of obfuscation; which required working on hardware level, where integrating the hardware support acts as a complement to the strength of the protection technique. Therefore, the software protection techniques should be designed in a modular fashion. Moreover, a hardware support such as Intel's

Advanced Encryption Standard (AES) instruction set, can be used with future solutions relying on the AES encryption.

With the current evolution, users shift from desktop to mobile devices, such as tablets and smartphones devices. Simultaneously, attackers struggle every new technique to exploit weaknesses in these devices and their applications. Future software protection research should concern on light weighted solutions which are suitable to run on mobile and embedded devices with limited resources.

Finally, we aim to improve the proposed framework to achieve a better trade-off between protection level, performance, and memory usage.

REFERENCES

[1] Amankwah, R., Kudjo, P. K., & Antwi, S. Y. (2017). Evaluation of Software Vulnerability Detection Methods and Tools: A Review. Evaluation, 169(8).

[2] Anckaert, B., Jakubowski, M. H., Venkatesan, R., & Saw, C. W. (2009, June). Runtime protection via dataflow flattening. In Emerging Security Information, Systems and Technologies, 2009. SECURWARE'09. Third International Conference on (pp. 242-248). IEEE.

[3] Anckaert, B., Jakubowski, M., Venkatesan, R., & De Bosschere, K. (2007). Run-time randomization to mitigate tampering. Advances in Information and Computer Security, 153-168.

[4] Anckaert, B., Madou, M., De Sutter, B., De Bus, B., De Bosschere, K., & Preneel, B. (2007, October). Program obfuscation: a quantitative approach. In Proceedings of the 2007 ACM workshop on Quality of protection (pp. 15-20). ACM.

[5] Andrivet, S. (2014, September). Black Hat Europe 2014, Retrieved on November, 28, 2017, From http://www.blackhat.com/eu-14/archives.html.

[6] Aucsmith, D. (1996). Tamper resistant software: An implementation. In Information Hiding (pp. 317-333). Springer Berlin/Heidelberg.

[7] Avoine, G., M. A. Bingol, Ioana Boureanu, S. Capkun, G. Hancke, S. Kardas, C. H. Kim et al. "Security of Distance-Bounding: A Survey." ACM Computing Surveys (2017).

[8] Balachandran, V., & Emmanuel, S. (2013). Potent and stealthy control flow obfuscation by stack based self-modifying code. IEEE Transactions on Information Forensics and Security, 8(4), 669-681.

[9] Balachandran, V., Keong, N. W., & Emmanuel, S. (2014, July). Function level control flow obfuscation for software security. In Complex, Intelligent and Software Intensive Systems (CISIS), 2014 Eighth International Conference on (pp. 133-140). IEEE.

[10] Banescu, S., Lucaci, C., Krämer, B., & Pretschner, A. (2016, October). VOT4CS: A Virtualization Obfuscation Tool for C. In Proceedings of the 2016 ACM Workshop on Software PROtection(pp. 39-49). ACM.

[11] Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., & Yang, K. (2001, August). On the (im) possibility of obfuscating programs. In Annual International Cryptology Conference (pp. 1-18). Springer, Berlin, Heidelberg.

[12] Batchelder, M., & Hendren, L. (2007, March). Obfuscating java: The most pain for the least gain. In International Conference on Compiler Construction (pp. 96-110). Springer, Berlin, Heidelberg.

[13] Bhansali, S., Chen, W. K., De Jong, S., Edwards, A., Murray, R., Drinić, M., ... & Chau, J. (2006, June). Framework for instruction-level tracing and analysis of program

executions. In Proceedings of the 2nd international conference on Virtual execution environments (pp. 154-163). ACM.

[14] Billet, O., Gilbert, H., & Ech-Chatbi, C. (2004, August). Cryptanalysis of a White Box AES Implementation. In Selected Areas in Cryptography (Vol. 3357, pp. 227-240).

[15] Birrer, B. D., Raines, R. A., Baldwin, R. O., Mullins, B. E., & Bennington, R. W. (2007, August). Program fragmentation as a metamorphic software protection. In Information Assurance and Security, 2007. IAS 2007. Third International Symposium on (pp. 369-374). IEEE.

[16] Blazy, S., & Trieu, A. (2016, January). Formal verification of control-flow graph flattening. In Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs (pp. 176-187). ACM.

[17] Bletsch, T., Jiang, X., Freeh, V. W., & Liang, Z. (2011, March). Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (pp. 30-40). ACM.

[18] Bos, J. W., Hubain, C., Michiels, W., & Teuwen, P. (2016, August). Differential computation analysis: Hiding your white-box designs is not enough. In International Conference on Cryptographic Hardware and Embedded Systems (pp. 215-236). Springer Berlin Heidelberg.

[19] Bos, J. W., Hubain, C., Michiels, W., Mune, C., Gonzalez, E. S., & Teuwen, P. (2017). White-Box Cryptography: Don't Forget About Grey Box Attacks. IACR Cryptology ePrint Archive, 2017, 355.

[20] Bosboom, J., Rajadurai, S., Wong, W. F., & Amarasinghe, S. (2014). StreamJIT: A commensal compiler for high-performance stream programming (Vol. 49, No. 10, pp. 177-195). ACM.

[21] Braga, A. M., & Dahab, R. (2016, August). Towards a Methodology for the Development of Secure Cryptographic Software. In Software Security and Assurance (ICSSA), 2016 International Conference on (pp. 25-30). IEEE.

[22] Bringer, J., Chabanne, H., & Dottax, E. (2006, January). Perturbing and protecting a traceable block cipher. In Communications and Multimedia Security (Vol. 4237, pp. 109-119).

[23] Buzatu, F. (2012). Methods for obfuscating Java programs. Journal of Mobile, Embedded and Distributed Systems, 4(1), 25-30.

[24] Canfora, G., Di Penta, M., & Cerulo, L. (2011). Achievements and challenges in software reverse engineering. Communications of the ACM, 54(4), 142-151.

[25] Cappaert, J. (2012). Code obfuscation techniques for software protection. Katholieke Universiteit Leuven, 1-112.

[26] Cappaert, J., & Preneel, B. (2010, October). A general model for hiding control flow. In Proceedings of the tenth annual ACM workshop on Digital rights management (pp. 35-42). ACM.

[27] Cappaert, J., Kisserli, N., Schellekens, D., & Preneel, B. (2006, November). Self-encrypting code to protect against analysis and tampering. In 1st Benelux Workshop Inf. Syst. Security.

[28] Cappaert, J., Preneel, B., Anckaert, B., Madou, M., & De Bosschere, K. (2008). Towards tamper resistant code encryption: Practice and experience. Information Security Practice and Experience, 86-100.

[29] Carlini, N., Barresi, A., Payer, M., Wagner, D., & Gross, T. R. (2015, August). Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In USENIX Security Symposium (pp. 161-176).

[30] Chan, J. T., & Yang, W. (2004). Advanced obfuscation techniques for Java bytecode. Journal of Systems and Software, 71(1), 1-10.

[31] Chan, P. P., Hui, L. C., & Yiu, S. M. (2013). Heap graph based software theft detection. IEEE Transactions on Information Forensics and Security, 8(1), 101-110.

[32] Chen, H. Y., Hou, T. W., & Lin, C. L. (2007). Tamper-proofing basis path by using oblivious hashing on Java. ACM Sigplan Notices, 42(2), 9-16.

[33] Chen, H., Yuan, L., Wu, X., Zang, B., Huang, B., & Yew, P. C. (2009, December). Control flow obfuscation with information flow tracking. In Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (pp. 391-400). ACM.

[34] Chen, J., Li, K., Wen, W., Chen, W., & Yan, C. (2017, September). Software Watermarking for Java Program Based on Method Name Encoding. In International Conference on Advanced Intelligent Systems and Informatics (pp. 865-874). Springer, Cham.

[35] Cho, J., Choi, K. Y., & Moon, D. (2015). Hybrid WBC: Secure and efficient encryption schemes using the White-Box Cryptography. IACR Cryptology ePrint Archive, 2015, 800.

[36] Chow, S., Eisen, P., Johnson, H., & Van Oorschot, P. C. (2002, August). White-box cryptography and an AES implementation. In Selected Areas in Cryptography (Vol. 2595, pp. 250-270).

[37] Chow, S., Eisen, P., Johnson, H., & Van Oorschot, P. C. (2002, November). A white-box DES implementation for DRM applications. In Digital Rights Management Workshop (Vol. 2696, pp. 1-15).

[38] Christensen, L. B., Johnson, B., Turner, L. A., & Christensen, L. B. (2011). Research methods, design, and analysis.

[39] Cohen, F. B. (1993). Operating system protection through program evolution. Computers & Security, 12(6), 565-584.

[40] Collberg, C. S., & Thomborson, C. (2002). Watermarking, tamper-proofing, and obfuscation-tools for software protection. IEEE Transactions on software engineering, 28(8), 735-746.

[41] Collberg, C., Thomborson, C., & Low, D. (1997). A taxonomy of obfuscating transformations. Department of Computer Science, The University of Auckland, New Zealand.

[42] Crane, S., Homescu, A., Brunthaler, S., Larsen, P., & Franz, M. (2015, February). Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In NDSS (pp. 8-11).

[43] Das, S. S. (2014). Code Obfuscation using Code Splitting with Self-Modifying Code (Doctoral dissertation).

[44] Davi, L., Dmitrienko, A., Nürnberger, S., & Sadeghi, A. R. (2012, August). XIFER: a software diversity tool against code-reuse attacks. In 4th ACM International Workshop on Wireless of the Students, by the Students, for the Students (S3 2012) (Vol. 174).

[45] De Mulder, Y., Wyseur, B., & Preneel, B. (2010, January). Cryptanalysis of a Perturbated White-Box AES Implementation. In INDOCRYPT (Vol. 6498, pp. 292-310).

[46] Debray, S., & Evans, W. (2002). Profile-guided code compression. ACM SIGPLAN Notices, 37(5), 95-105.

[47] Debray, S., & Patel, J. (2010, October). Reverse engineering self-modifying code: Unpacker extraction. In Reverse Engineering (WCRE), 2010 17th Working Conference on (pp. 131-140). IEEE.

[48] Dedić, N., Jakubowski, M., & Venkatesan, R. (2007). A graph game model for software tamper protection. In Information Hiding (pp. 80-95). Springer Berlin/Heidelberg.

[49] Diffie, W., & Hellman, M. (1976). New directions in cryptography. IEEE transactions on Information Theory, 22(6), 644-654.

[50] Drape, S., Thomborson, C., & Majumdar, A. (2007, September). Specifying imperative data obfuscations. In ISC (Vol. 7, pp. 299-314).

[51] Ertaul, L., & Venkatesh, S. (2005, June). Novel obfuscation algorithms for software security. In Proceedings of the 2005 International Conference on Software Engineering Research and Practice, SERP (Vol. 5).

[52] Falcarin, P., Di Carlo, S., Cabutto, A., Garazzino, N., & Barberis, D. (2011, February). Exploiting code mobility for dynamic binary obfuscation. In Internet Security (WorldCIS), 2011 World Congress on (pp. 114-120). IEEE.

[53] Fang, H., Wu, Y., Wang, S., & Huang, Y. (2011, October). Multi-stage Binary Code Obfuscation Using Improved Virtual Machine. In ISC (pp. 168-181).

[54] Foket, C., De Sutter, B., & De Bosschere, K. (2014). Pushing java type obfuscation to the limit. IEEE Transactions on Dependable and Secure Computing, 11(6), 553-567.

[55] Frederiksen, B., Courtney, S. (2011). Reverse Engineering: Vulnerabilities and Solutions. Retrieved on November,11,2017, From https://www.pnsqc.org.

[56] Friedman, S. E., Musliner, D. J., & Keller, P. K. (2015, February). Chronomorphic programs: Using runtime diversity to prevent code reuse attacks. In Proceedings ICDS 2015: The 9th International Conference on Digital Society.

[57] Gautam, P., & Saini, H. (2017). A Novel Software Protection Approach for Code Obfuscation to Enhance Software Security. International Journal of Mobile Computing and Multimedia Communications (IJMCMC), 8(1), 34-47.

[58] Ge, J., Chaudhuri, S., & Tyagi, A. (2005, November). Control flow based obfuscation. In Proceedings of the 5th ACM workshop on Digital rights management (pp. 83-92). ACM.

[59] Gearhart, A., & Kelly, D. (2017). The Development of a Stochastic Model for Software Diversity. In International Conference on Cyber Warfare and Security (p. 517). Academic Conferences International Limited.

[60] Geoffray, N., Thomas, G., Lawall, J., Muller, G., & Folliot, B. (2010, March). VMKit: a substrate for managed runtime environments. In ACM Sigplan Notices (Vol. 45, No. 7, pp. 51-62). ACM.

[61] Ghosh, S., Hiser, J. D., & Davidson, J. W. (2010, June). A secure and robust approach to software tamper resistance. In International Workshop on Information Hiding (pp. 33-47). Springer, Berlin, Heidelberg.

[62] Ghosh, S., Hiser, J., & Davidson, J. W. (2012). Replacement attacks against VM-protected applications. ACM SIGPLAN Notices, 47(7), 203-214.

[63] Ghosh, S., Hiser, J., & Davidson, J. W. (2013, January). Software protection for dynamically-generated code. In Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (p. 1). ACM.

[64] Gomes, N. D., Cerqueira, P. A., & Almeida, L. A. (2015). A survey on software piracy empirical literature: Stylized facts and theory. Information Economics and Policy, 32, 29-37.

[65] Gu, Y., Wyseur, B., & Preneel, B. (2011). Software-based protection is moving to the mainstream. IEEE Software, Special Issue on Software Protection, 28(2), 56-59.

[66] Hamilton, J., & Danicic, S. (2011, February). A survey of static software watermarking. In Internet Security (WorldCIS), 2011 World Congress on (pp. 100-107). IEEE.

[67] Hamilton, J., & Danicic, S. (2011, February). A survey of static software watermarking. In Internet Security (WorldCIS), 2011 World Congress on (pp. 100-107). IEEE.

[68] Han, S., Ryu, M., Cha, J., & Choi, B. U. (2014, September). HOTDOL: HTML Obfuscation with Text Distribution to Overlapping Layers. In Computer and Information Technology (CIT), 2014 IEEE International Conference on (pp. 399-404). IEEE.

[69] Hosseinzadeh, S., Rauti, S., Laurén, S., Mäkelä, J. M., Holvitie, J., Hyrynsalmi, S., & Leppänen, V. (2016, June). A Survey on Aims and Environments of Diversification and

Obfuscation in Software Security. In Proceedings of the 17th International Conference on Computer Systems and Technologies 2016 (pp. 113-120). ACM.

[70] Hou, T. W., & Chen, H. Y. (2010). Toward robustly protecting data of a dominant path in a java method. Journal of the Chinese Institute of Engineers, 33(2), 251-262.

[71] IDA multi-processor disassembler and debugger. Retrieved on January, 2018, from http://www.hex-rays.com.

[72] Imran, S., Chaudhry, R., Gilani, J., & Nehvi, S. (2015). The Techniques and Applications of Digital Watermarking. International Journal of Recent Research Aspects, ISSN: 2349-7688, Special Issue: Engineering Research Aspects Feb 2015, pp. 74-78.

[73] Jacob, M., Boneh, D., & Felten, E. (2002, November). Attacking an obfuscated cipher by injecting faults. In Digital Rights Management Workshop (Vol. 2696, pp. 16-31).

[74] Jacob, M., Jakubowski, M. H., & Venkatesan, R. (2007, September). Towards integral binary execution: Implementing oblivious hashing using overlapped instruction encodings. In Proceedings of the 9th workshop on Multimedia & security (pp. 129-140). ACM.

[75] Jakubowski, M., Naldurg, P., Patankar, V., & Venkatesan, R. (2007). Software integrity checking expressions (ICEs) for robust tamper detection. In Information Hiding (pp. 96-111). Springer Berlin/Heidelberg.

[76] Jensor - (Java Profiler). Retrieved on October, 11, 2017, From http://jensor.sourceforge.net.

[77] Jeon, C., & Cho, Y. (2012, October). A robust steganography-based software watermarking. In Proceedings of the 2012 ACM Research in Applied Computation Symposium (pp. 333-337). ACM.

[78] Jung, D. W., Kim, H. S., & Park, J. G. (2008). A Code Block Cipher Method to Protect Application Programs from Reverse Engineering. Journal of the Korea Institute of Information Security and Cryptology, 18(2), 85-96.

[79] Junod, P., Rinaldini, J., Wehrli, J., & Michielin, J. (2015, May). Obfuscator-LLVM--software protection for the masses. In Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on (pp. 3-9). IEEE.

[80] Kanzaki, Y., & Monden, A. (2010, November). A software protection method based on time-sensitive code and self-modification mechanism. In Proc of IASTED (Vol. 10, pp. 325-331).

[81] Kanzaki, Y., Monden, A., Nakamura, M., & Matsumoto, K. I. (2003, November). Exploiting self-modification mechanism for program protection. In Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International (pp. 170-179). IEEE.

[82] Kanzaki, Y., Monden, A., Nakamura, M., & Matsumoto, K. I. (2006). A software protection method based on instruction camouflage. Electronics and Communications in Japan (Part III: Fundamental Electronic Science), 89(1), 47-59.

[83] Kanzaki, Y., Monden, A., Nakamura, M., & Matsumoto, K. I. (2008). Program camouflage: a systematic instruction hiding method for protecting secrets. World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering, 2(7), 2365-2371.

[84] Kapi, A. Y., & Ibrahim, S. (2011, December). Fixed size encoding scheme for software watermarking. In Information Assurance and Security (IAS), 2011 7th International Conference on (pp. 35-39). IEEE.

[85] Karnick, M., MacBride, J., McGinnis, S., Tang, Y., & Ramachandran, R. (2006, November). A qualitative analysis of java obfuscation. In proceedings of 10th IASTED international conference on software engineering and applications, Dallas TX, USA.

[86] Karroumi, M. (2010, December). Protecting White-Box AES with Dual Ciphers. In ICISC (Vol. 6829, pp. 278-291).

[87] Khalilian, A., Golbaghi, H., Nourazar, A., Haghighi, H., & Vahidi-Asl, M. (2016, October). MetaSPD: Metamorphic Analysis for Automatic Software Piracy Detection. In Computer and Knowledge Engineering (ICCKE), 2016 6th International Conference on (pp. 123-128). IEEE.

[88] Khan, M., Akram, M., & Riaz, N. (2015). A Comparative Analysis of Software Protection Schemes. International Arab Journal of Information Technology (IAJIT), 12(3).

[89] Kiehtreiber, P., & Brouwer, M. (2006). U.S. Patent No. 7,103,779. Washington, DC: U.S. Patent and Trademark Office.

[90] Kim, N. Y., Shim, J., Cho, S. J., Park, M., & Han, S. (2016). Android Application Protection against Static Reverse Engineering based on Multidexing. J. Internet Serv. Inf. Secur., 6(4), 54-64.

[91] Kimball, W. B., & Baldwin, R. O. (2012). Emulation-based software protection. U.S. Patent No. 8,285,987. Washington, DC: U.S. Patent and Trademark Office.

[92] Kisserli, N., Cappaert, J., & Preneel, B. (2007). Software security through targeted diversification. Software Security Assessments—CoBaSSA 2007—, 10.

[93] Kulkarni, A., & Lodha, S. (2012). Software protection through code obfuscation. Project Report, Tata Research Development and Design Centre, Pune.

[94] Kulkarni, A., & Metta, R. (2014, April). A new code obfuscation scheme for software protection. In Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on (pp. 409-414). IEEE.

[95] László, T., & Kiss, Á. (2009). Obfuscating C++ programs via control flow flattening. Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica, 30, 3-19.

[96] Lazar, D., Chen, H., Wang, X., & Zeldovich, N. (2014, June). Why does cryptographic software fail? a case study and open problems. In Proceedings of 5th Asia-Pacific Workshop on Systems (p. 7). ACM.

[97] Lee, H., & Kaneko, K. (2009, April). Two new algorithms for software watermarking by register allocation and their empirical evaluation. In Information Technology: New Generations, 2009. ITNG'09. Sixth International Conference on (pp. 217-222). IEEE.

[98] Lepoint, T., Rivain, M., De Mulder, Y., Roelse, P., & Preneel, B. (2013, August). Two attacks on a white-box AES implementation. In International Conference on Selected Areas in Cryptography (pp. 265-285). Springer, Berlin, Heidelberg.

[99] Li, J., & Liu, Q. (2010, May). Design of a software watermarking algorithm based on register allocation. In e-Business and Information System Security (EBISS), 2010 2nd International Conference on (pp. 1-4). IEEE.

[100] Link, H. E., & Neumann, W. D. (2005, April). Clarifying obfuscation: improving the security of white-box DES. In Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on (Vol. 1, pp. 679-684). IEEE.

[101] Linn, C., & Debray, S. (2003, October). Obfuscation of executable code to improve resistance to static disassembly. In Proceedings of the 10th ACM conference on Computer and communications security (pp. 290-299). ACM.

[102] Lungu, C., & Potolea, R. (2012, August). Designing software locking mechanisms against reverse engineering, using artificial neural networks. In Intelligent Computer Communication and Processing (ICCP), 2012 IEEE International Conference on (pp. 83-86). IEEE.

[103] Luo, L., Ming, J., Wu, D., Liu, P., & Zhu, S. (2014, November). Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 389-400). ACM.

[104] Ma, H., Li, R., Yu, X., Jia, C., & Gao, D. (2016). Integrated Software Fingerprinting via Neural-Network Based Control Flow Obfuscation. IEEE Transactions on Information Forensics and Security, 11(10), 2322-2337.

[105] Ma, H., Ma, X., Liu, W., Huang, Z., Gao, D., & Jia, C. (2014, September). Control flow obfuscation using neural network to fight concolic testing. In International Conference on Security and Privacy in Communication Systems (pp. 287-304). Springer International Publishing.

[106] Madou, M., Anckaert, B., Moseley, P., Debray, S., De Sutter, B., & De Bosschere, K. (2005, August). Software protection through dynamic code mutation. In International Workshop on Information Security Applications (pp. 194-206). Springer, Berlin, Heidelberg.

[107] Mana, A., & Pimentel, E. (2011). An efficient software protection scheme. in Proceedings of the 16th International Conference on Information Security: Trusted Information, Paris, France, pp. 385-401.

[108] Masoumi, M., & Amiri, S. (2014). Content protection in video data based on robust digital watermarking resistant to intentional and unintentional attacks. Int. Arab J. Inf. Technol., 11(2), 204-212.

[109] Mavrogiannopoulos, N., Kisserli, N., & Preneel, B. (2011). A taxonomy of self-modifying code for obfuscation. Computers & Security, 30(8), 679-691.

[110] McCabe, T. J. (1976). A complexity measure. IEEE Transactions on software Engineering, (4), 308-320.

[111] Memon, J. M., Mughal, A., & Memon, F. (2006, November). Preventing reverse engineering threat in Java using byte code obfuscation techniques. In Emerging Technologies, 2006. ICET'06. International Conference on (pp. 689-694). IEEE.

[112] Michiels, W. (2010). Opportunities in white-box cryptography. IEEE Security & Privacy, 8(1).

[113] Michiels, W., & Gorissen, P. (2007, October). Mechanism for software tamper resistance: an application of white-box cryptography. In Proceedings of the 2007 ACM workshop on Digital Rights Management (pp. 82-89). ACM.

[114] Microsoft Corporation. Driver signing requirements for Windows (2002), Retrieved on October, 03, 2017, From http://msdn.microsoft.com/en-us/windows/hardware/gg487317.

[115] Ming, J., & Wu, D. (2016, October). BinCFP: Efficient Multi-threaded Binary Code Control Flow Profiling. In Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on (pp. 61-66). IEEE.

[116] Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K. W., & Franz, M. (2015, February). Opaque Control-Flow Integrity. In NDSS (Vol. 26, pp. 27-30).

[117] Monden, A., Iida, H., Matsumoto, K. I., Inoue, K., & Torii, K. (2000). A practical method for watermarking java programs. In Computer Software and Applications Conference, 2000. COMPSAC 2000. The 24th Annual International (pp. 191-197). IEEE.

[118] Moser, A., Kruegel, C., & Kirda, E. (2007, May). Exploring multiple execution paths for malware analysis. In Security and Privacy, 2007. SP'07. IEEE Symposium on (pp. 231-245). IEEE.

[119] Muchnick, S. S. (1997). *Advanced compiler design implementation*. Morgan Kaufmann.

[120] Natani, P., & Vidyarthi, D. (2014). An Overview of Detection Techniques for Metamorphic Malware. In Intelligent Computing, Networking, and Informatics (pp. 637-643). Springer, New Delhi.

[121] Neves, S., & Araujo, F. (2012, September). Binary code obfuscation through C++ template metaprogramming. In INForum (pp. 28-40).

[122] Ogheneovo, E. E., Oputeh, C. K., Gomathy, M. M., Ponni, M. M., Baskar, R., Rai, Y., ... & Grewal, J. S. Source Code Obfuscation: A Technique for Checkmating Software Reverse Engineering. International Journal of Engineering Science Invention ISSN (Online), 2319-6734.

[123] OllyDbg 32-bit assembler level analyzing debugger for Windows. http://www.ollydbg.de, accessed January, 2018.

[124] Pang, J., Zhang, Y., Dai, C., Sun, D., & Wang, Q. (2013). A novel disassemble algorithm designed for malicious file. Research Journal of Applied Sciences, 5.

[125] Pappas, V., Polychronakis, M., & Keromytis, A. D. (2013). Practical software diversification using in-place code randomization. In Moving Target Defense II (pp. 175-202). Springer, New York, NY.

[126] Patel, S., & Pattewar, T. (2014, October). Software birthmark based theft detection of JavaScript programs using agglomerative clustering and improved frequent subgraph mining. In Advances in Electronics, Computers and Communications (ICAECC), 2014 International Conference on (pp. 1-6). IEEE.

[127] Peng, Y., Liang, J., & Li, Q. (2016, August). A control flow obfuscation method for Android applications. In Cloud Computing and Intelligence Systems (CCIS), 2016 4th International Conference on (pp. 94-98). IEEE.

[128] Popov, I. V., Debray, S. K., & Andrews, G. R. (2007, August). Binary Obfuscation Using Signals. In USENIX Security Symposium (pp. 275-290).

[129] Protsenko, M., Kreuter, S., & Müller, T. (2015, August). Dynamic Self-Protection and Tamperproofing for Android Apps Using Native Code. In Availability, Reliability and Security (ARES), 2015 10th International Conference on (pp. 129-138). IEEE.

[130] Qu, G., & Potkonjak, M. (1998, November). Analysis of watermarking techniques for graph coloring problem. In Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design (pp. 190-193). ACM.

[131] Radkani, E., Hashemi, S., Keshavarz-Haddad, A., & Haeri, M. A. (2017). An entropy-based distance measure for analyzing and detecting metamorphic malware. Applied Intelligence, 1-11.

[132] Rasch, A., & Wenzel, T. (2013). Piracy in a two-sided software market. Journal of Economic Behavior & Organization, 88(0), 78-89. In International Workshop on Information Hiding (pp. 33-47). Springer, Berlin, Heidelberg.

[133] Roeder, T., & Schneider, F. B. (2010). Proactive obfuscation. ACM Transactions on Computer Systems (TOCS), 28(2), 4.

[134] Samawi, V., & Sulaiman, A. (2013). Software Protection via Hiding Function Using Software Obfuscation. Int. Arab J. Inf. Technol., 10(6), 587-594.

[135] Santos, N., Raj, H., Saroiu, S., & Wolman, A. (2014, February). Using ARM TrustZone to build a trusted language runtime for mobile applications. In ACM SIGARCH Computer Architecture News (Vol. 42, No. 1, pp. 67-80). ACM.

[136] Sasdrich, P., Moradi, A., & Güneysu, T. (2016, March). White-Box Cryptography in the Gray Box. In International Conference on Fast Software Encryption (pp. 185-203). Springer Berlin Heidelberg.

[137] Sasirekha N. and Hemalatha M., "A Survey on Software Protection Techniques Against Various Attacks," Global Journal of Computer Science and Technology, vol. 12, no. 1, pp. 53-58, 2012.

[138] Sasirekha, N., & Hemalatha, M. (2012). An enhanced code encryption approach with HNT transformations for software security. International Journal of Computer Applications, 53(10).

[139] Schrittwieser, S., & Katzenbeisser, S. (2011). Code obfuscation against static and dynamic reverse engineering. In Information Hiding (pp. 270-284). Springer Berlin/Heidelberg.

[140] Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., & Weippl, E. (2016). Protecting software through obfuscation: Can it keep pace with progress in code analysis? ACM Computing Surveys (CSUR), 49(1), 4.

[141] Sebastian, S. A., Malgaonkar, S., Shah, P., Kapoor, M., & Parekhji, T. (2016, February). A study & review on code obfuscation. In Futuristic Trends in Research and Innovation for Social Welfare (Startup Conclave), World Conference on (pp. 1-6). IEEE.

[142] Sha, Z., Jiang, H., & Xuan, A. (2009, October). Software watermarking algorithm by coefficients of equation. In Genetic and Evolutionary Computing, 2009. WGEC'09. 3rd International Conference on (pp. 410-413). IEEE.

[143] Sharif, M. I., Lanzi, A., Giffin, J. T., & Lee, W. (2008, February). Impeding Malware Analysis Using Conditional Code Obfuscation. In NDSS.

[144] Sharma, A., & Kushwaha, D. S. (2010, September). Complexity measure based on requirement engineering document and its validation. In Computer and Communication Technology (ICCCT), 2010 International Conference on (pp. 608-615). IEEE.

[145] Sharma, A., & Sahay, S. K. (2014). Evolution and detection of polymorphic and metamorphic malwares: A survey. arXiv preprint arXiv:1406.7061.

[146] Shi, J. C. C. W. Q., & Lv, G. (2010). Implementation of bytecode-based software watermarking for java programs. In the IASTED International Conference on

Communication and Information Security,2010. Marina Del Rey, USA, 4–6 December, pp. 542–547.

[147] Shirali-Shahreza, M., & Shirali-Shahreza, S. (2008, April). Software watermarking by equation reordering. In Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on (pp. 1-4). IEEE.

[148] Sivadasan, P., & Lal, P. S. (2008). Array based java source code obfuscation using classes with restructured arrays. arXiv preprint arXiv:0807.4309.

[149] Sivadasan, P., & Lal, P. S. (2009). Jconsthide: a framework for java source code constant hiding. arXiv preprint arXiv:0904.3458.

[150] Sullivan, D., Arias, O., Gens, D., Davi, L., Sadeghi, A. R., & Jin, Y. (2017). Execution Integrity with In-Place Encryption. arXiv preprint arXiv:1703.02698.

[151] Tang, Z., Chen, X., Fang, D., & Chen, F. (2009, December). Research on Java software protection with the obfuscation in identifier renaming. In Innovative Computing, Information and Control (ICICIC), 2009 Fourth International Conference on (pp. 1067-1071). IEEE.

[152] Tanter, É., Toledo, R., Pothier, G., & Noyé, J. (2008). Flexible metaprogramming and AOP in Java. Science of Computer Programming, 72(1), 22-30.

[153] Tian, Z., Zheng, Q., Liu, T., Fan, M., Zhuang, E., & Yang, Z. (2015). Software plagiarism detection with birthmarks based on dynamic key instruction sequences. IEEE Transactions on Software Engineering, 41(12), 1217-1235.

[154] Tsai, H. Y., Huang, Y. L., & Wagner, D. (2009). A graph approach to quantitative analysis of control-flow obfuscating transformations. IEEE Transactions on Information Forensics and Security, 4(2), 257-267.

[155] Uppal, D., Mehra, V., & Verma, V. (2014). Basic survey on malware analysis, tools and techniques. International Journal on Computational Sciences and Applications (IJCSA), 4(1), 103-12.

[156] Van Oorschot, P. C., Somayaji, A., & Wurster, G. (2005). Hardware-assisted circumvention of self-hashing software tamper resistance. IEEE Transactions on Dependable and Secure Computing, 2(2), 82-92.

[157] Vasudevan, M., Pratik, S., Gopichand, C., & Amalanathan, A. (2015). An Architecture of Class Loader System in Java Bytecode Obfuscation. Indian Journal of Science and Technology, 8(S2), 291-294.

[158] Viticchié, A., Regano, L., Torchiano, M., Basile, C., Ceccato, M., Tonella, P., & Tiella, R. (2016, October). Assessment of source code obfuscation techniques. In Source Code Analysis and Manipulation (SCAM), 2016 IEEE 16th International Working Conference on (pp. 11-20). IEEE.

[159] Vrba, Ž., Halvorsen, P., & Griwodz, C. (2010, February). Program obfuscation by strong cryptography. In Availability, Reliability, and Security, 2010. ARES'10 International Conference on (pp. 242-247). IEEE.

[160] Wang, S., Wang, P., & Wu, D. (2017, September). Composite software diversification. In Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on (pp. 284-294). IEEE.

[161] Wang, Z., Jia, C., Liu, M., & Yu, X. (2012, July). Branch obfuscation using code mobility and signal. In Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual (pp. 553-558). IEEE.

[162] Wartell, R., Mohan, V., Hamlen, K. W., & Lin, Z. (2012, October). Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 157-168). ACM.

[163] Wu, Y., Suhendray, V., Saputra, H., & Zhao, Z. (2016, December). Obfuscating Software Puzzle for Denial-of-Service Attack Mitigation. In Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), 2016 IEEE International Conference on (pp. 115-122). IEEE.

[164] Wu, Z., Gianvecchio, S., Xie, M., & Wang, H. (2010, October). Mimimorphism: A new approach to binary code obfuscation. In Proceedings of the 17th ACM conference on Computer and communications security (pp. 536-546). ACM.

[165] Wurster, G., Van Oorschot, P. C., & Somayaji, A. (2005, May). A generic attack on checksumming-based software tamper resistance. In Security and Privacy, 2005 IEEE Symposium on (pp. 127-138). IEEE.

[166] Xianya, M., Yi, Z., Baosheng, W., & Yong, T. (2015, December). A Survey of Software Protection Methods Based on Self-Modifying Code. In Computational Intelligence and Communication Networks (CICN), 2015 International Conference on (pp. 589-593). IEEE.

[167] Xie, X., Liu, F., Lu, B., & Xiang, F. (2016). An Iteration Obfuscation Based on Instruction Fragment Diversification and Control Flow Randomization. International Journal of Computer Theory and Engineering, 8(4), 303.

[168] Xie, X., Lu, B., Gong, D., Luo, X., & Liu, F. (2015). Random table and hash coding-based binary code obfuscation against stack trace analysis. IET Information Security, 10(1), 18-27.

[169] Xu, H., Zhou, Y., Kang, Y., & Lyu, M. R. (2017). On Secure and Usable Program Obfuscation: A Survey. arXiv preprint arXiv:1710.01139.

[170] Yang, L. (2013). White Box Cryptography. Final Project-CPCS4600.

[171] Yasin, A., & Nasra, I. (2016). Dynamic Multi Levels Java Code Obfuscation Technique (DMLJCOT). International Journal of Computer Science and Security (IJCSS), 10(4), 140.

[172] Zeng, J., Fu, Y., Miller, K. A., Lin, Z., Zhang, X., & Xu, D. (2013, November). Obfuscation resilient binary code reuse through trace-oriented programming. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (pp. 487-498). ACM.

[173] Zhang, M., & Sekar, R. (2013, August). Control Flow Integrity for COTS Binaries. In USENIX Security Symposium (pp. 337-352).

[174] Zhang, X., He, F., & Zuo, W. (2008, December). Hash function based software watermarking. In Advanced Software Engineering and Its Applications, 2008. ASEA 2008 (pp. 95-98). IEEE.

[175] Zhang, X., He, F., & Zuo, W. (2010). Theory and practice of program obfuscation. In Convergence and Hybrid Information Technologies. InTech.

[176] Zhang, Y., Huang, G., Liu, X., Zhang, W., Mei, H., & Yang, S. (2012, October). Refactoring android java code for on-demand computation offloading. In ACM SIGPLAN Notices (Vol. 47, No. 10, pp. 233-248). ACM.

[177] Zhe, C., Zhi, W., Xiaochu, W., & Chunfu, J. (2015). Using Code Mobility to Obfuscate Control Flow in Binary Codes. Journal of Computer Research and Development, 8, 023.

[178] Zhu, J., Liu, Y., Wang, A., & Yin, K. (2011). H Function based Tamper-proofing Software Watermarking Scheme. JSW, 6(1), 148-155.

[179] Zhu, W., & Thomborson, C. (2006). Algorithms to watermark software through register allocation. Lecture Notes in Computer Science, 3919, 180.

APPENDIX

Source Code of java program used in our experiment (call graph of this code is shown in figure 1)

```
public  void main(String[] args) {
    int f1,f2,f3,f4;

    f1 = function1(2, 5);          // Faked function call
    f2 =  function2(5);            // Faked function call
    f3 = function3(22);            // Real function call
    f4 = function4(12,5);          // Faked function call
    }//end main

int function1(int a,int b) {  // Faked function definition
    int result, output;
    result = a + b;
    output = function5(result);
    return output;
  }

int function2(int num) {     // Faked function definition
    int x;
    x = function5(num);
    return num * x ;
  }

int function3(int a) {  // Real function definition
    int result;
    String data;
    data =function6(a,"my name");
    result = function7(a);
    return result;
  }

int function4(int a,int b)   // Faked function definition
  {
    int result;
    result = a + b;
    result = function7(result);
    return result;
  }

int function5(int a)         // Faked function definition
  {
    int x,y;
    y = function7(33);
    return y  * a;
  }

String function6(float a, String b)  // Real function
  {
    float value, result;
    value = (float) (a *  3.14);
    String name = "My name =";
  int function7(int a)    {     // Faked function definition
    int x;
    x = function9(3,2);
    return a * x;
  }

float function8(int x, float y, float z) // Real function definition {
    String info;
    float result;
    result = y - z;
    info = function10(x,"my address",y,z);
    return result;
  }

  int function9(int x, int y) {  // Faked function definition
    int sqaure, result;
    sqaure = x * y;
    result = function12(sqaure,y);
    return result;
  }

String function10(int a, String b, float c, float d)
// Real function
  {
    int age = a;
    int result;
    String address = b;
    float weight = c;
    float height = d;
    double normal_range = height - weight;
    result = function12(a,12);
    return "My info is : \n" + " Age : " + a + " Address" + b + "
Weight" + c + "\n" + " Normal Range=" + normal_range;
  }

float function11(float x, float y) // Faked function definition
  {
    int age;
    int value1 = 15,value2 = 5;
    float value;
    value = (float) (x *  3.14);
    age = function9(value1,value2);
    value = function8(value1,x,y);
    return x * value + age;
  }

  int function12(int a, int b)        // Faked function definition
  {
    int sum;
    sum = a + b;
```

```
    result = function11(a,value);
    return name + b + " my age" + a;
}
```

```
    retrn sum;
}
```

Binary Search Tree Program

```
// Java program to demonstrate insert operation in binary search tree
class BinarySearchTree {
        /* Class containing left and right child of current node and key value*/
        class Node {
                int key;
                Node left, right;

                public Node(int item) {
                        key = item;
                        left = right = null;
                }}
// Root of BST
        Node root;
        // Constructor
        BinarySearchTree() {
                root = null;
        }
        // This method mainly calls insertRec()
        void insert(int key) {
        root = insertRec(root, key);
        }

        /* A recursive function to insert a new key in BST */
        Node insertRec(Node root, int key) {
                /* If the tree is empty, return a new node */
                if (root == null) {
                        root = new Node(key);
                        return root;
                }

                /* Otherwise, recur down the tree */
                if (key < root.key)
                        root.left = insertRec(root.left, key);
                else if (key > root.key)
                        root.right = insertRec(root.right, key);
        inorderRec(root);
        }
        // A utility function to do inorder traversal of BST
        void inorderRec(Node root) {
                if (root != null) {
                        inorderRec(root.left);
                        System.out.println(root.key);
                        inorderRec(root.right);
                }}
}
```

Depth First Search Program

```
class DepthFirstSearch {
   private int V;   // No. of vertices
   // Array  of lists for Adjacency List Representation
   private LinkedList<Integer> adj[];
   // Constructor
   DepthFirstSearch(int v){
      V = v;
      adj = new LinkedList[v];
      for (int i=0; i<v; ++i)
         adj[i] = new LinkedList();
   }
   //Function to add an edge into the graph
   void addEdge(int v, int w){
      adj[v].add(w);  // Add w to v's list.
   }
   // A function used by DFS
   void DFSUtil(int v,boolean visited[]) {
      // Mark the current node as visited and print it
      visited[v] = true;
      System.out.print(v+" ");

      // Recur for all the vertices adjacent to this vertex
      Iterator<Integer> i = adj[v].listIterator();
      while (i.hasNext()) {
         int n = i.next();
         if (!visited[n])
            DFSUtil(n, visited);
      } }
   // The function to do DFS traversal. It uses recursive DFSUtil()
   void DFS(int v) {
      // Mark all the vertices as not visited(set as
      // false by default in java)
      boolean visited[] = new boolean[V];
      // Call the recursive helper function to print DFS traversal
      DFSUtil(v, visited);
   }
}
```

Breadth First Search Program

```
public class BreadthFirstSearch{
private Queue<Node> queue;
	static ArrayList<Node> nodes=new ArrayList<Node>();
	static class Node {
		int data;
		boolean visited;
		List<Node> neighbours;
		Node(int data){
			this.data=data;
			this.neighbours=new ArrayList<>();
		}
		public void addneighbours(Node neighbourNode){
			this.neighbours.add(neighbourNode);
		}
		public List<Node> getNeighbours() {
			return neighbours;
		}
		public void setNeighbours(List<Node> neighbours) {
			this.neighbours = neighbours;
		}
	}

	public BreadthFirstSearch(){
		queue = new LinkedList<Node>();
	}

	public void BFS(Node node) {
		queue.add(node);
		node.visited=true;
		while (!queue.isEmpty()){
			Node element=queue.remove();
			System.out.print(element.data + "  ");
			List<Node> neighbours=element.getNeighbours();
			 for (int i = 0; i < neighbours.size(); i++) {
				Node n=neighbours.get(i);
				if(n!=null && !n.visited){
					queue.add(n);
					n.visited=true;
				}}}}
}
```

Greedy Search Program

```
class GreedySearch {
        static final int V=9;
        int minDistance(int dist[], Boolean sptSet[]) {
                // Initialize min value
                int min = Integer.MAX_VALUE, min_index=-1;
                for (int v = 0; v < V; v++){
                        if (sptSet[v] == false && dist[v] <= min){
                                min = dist[v];
                                min_index = v;
                        }}
                return min_index;
        }
        // A utility function to print the constructed distance array
        void printSolution(int dist[], int n) {
                System.out.println("Vertex Distance from Source");
                for (int i = 0; i < V; i++)
                        System.out.println(i+" tt "+dist[i]);
        }
        // algorithm for a graph represented using adjacency matrix representation
        void dijkstra(int graph[][], int src) {
        int dist[] = new int[V];   // The output array. dist[i] will hold the shortest distance from src to i
        Boolean sptSet[] = new Boolean[V];  // path tree or shortest distance from src to i is finalized
                // Initialize all distances as INFINITE and stpSet[] as false
                for (int i = 0; i < V; i++) {
                dist[i] = Integer.MAX_VALUE;
                sptSet[i] = false;
                }
                // Distance of source vertex from itself is always 0
                dist[src] = 0;

                // Find shortest path for all vertices
                for (int count = 0; count < V-1; count++) {
                        // Pick the minimum distance vertex from the set of vertices
                        int u = minDistance(dist, sptSet);
                        // Mark the picked vertex as processed
                        sptSet[u] = true;
                        // Update dist value of the adjacent vertices of the picked vertex.
                        for (int v = 0; v < V; v++) {
if (!sptSet[v] && graph[u][v]!=0 && dist[u] != Integer.MAX_VALUE && dist[u]+graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
                }}
                printSolution(dist, V); // print the constructed distance array
        }
}
```

Source code of tested program used by Ma, et al., (2014)

```
void main ()
{
int Var = 12 ;
for (x = 0; x<20; x++) {
Var +=x;
}}
```

Source code of tested program used by Ma, et al., (2016)

```
void main ()
{
int x, y ;
x = 0;
for (y = 0; y<10; y++)
{
if (y == 5)
Y += x;
X++;
}
}
```

Source code of tested program used by Bhansali , et al., (2006)

```
void main () {
i = 1;
for (j = 0; j < 10; j++) {
i = i + j;
}
k= i; // value read is 46
system_call ();
k = i; // value read is 0
}
```

Source code of tested program used by Tsai , et al., (2009)

```
int k(int b){
Int I;
For (i=2;i<=b/2;i++){
If(b%i==0) return 0;
Return i
}
void main () {
int a,b,sum;
system.out.println(a);
for(sum = 0, b=2;b<=a;b++){
if(k(b))  system.out.println(b);
sum + =b;
} return 0;
}
```

Source code of tested program used by Zeng , et al., (2013)

```
void main () {
if (year_of_service > 10) {
 if (salary < 100000.0)
 salary = 100000.0;
 else
 salary = salary*1.02;
}
```

Source code of tested program used by Xie, et al., (2015)

```
int func3() {
return 2
}

Int func3(){
Return func2();
}

int x_fun2(int a, int b, int c){
int temp;
temp = a+b+c;
temp = temp * func2();
temp =temp-func1(temp,a);
return temp;
}
```

```
int func2(){
return func3();
        }

 int x_func1(){
  int a=2;
  a+= x_func2(1,2,3);
   return a;
    }
 void main () {
    int a=1,b=2,result1,result2;
     result1=func1(a,b);
     result2=x_func1();
     system.out.println(result1);
     system.out.println(result2);
          return0;
                }
```

Source code of tested program used by Xie, et al., (2016)

ShellSort.exe

```
void ShellSort(int v[],int n){
int gap,i,j,temp;
for(gap=n/2;gap>0;gap /= 2){
for(i=gap;i<n;i++){
for(j=i-gap;(j >= 0) && (v[j] > v[j+gap]);j -= gap ){
temp=v[j];
v[j]=v[j+gap];
v[j+gap]=temp;
}
}
}
}
```

ثانيا، اقترحنا تقنية التشويش على أساس دمج آلية التشفير ضمن الشبكة العصبية الصناعية المتكررة وذلك من أجل تعزيز مستوى حماية البرمجيات ضد التحليل الديناميكي. توفر الشبكة العصبية سمة أمنية قوية في حماية البرمجيات، نظرا لقدرتها على تمثيل الخوارزميات الغير خطية مع القدرة الحسابية القوية. النظام مصمم لتمكين الشبكة العصبية من توليد تشفيرات مختلفة لنفس البيانات المحمية. هذا يخلق علاقة متعددة بين مفاتيح فك التشفير والبيانات المشفرة. علاوة على ذلك، استبدلنا وظيفة فك التشفير للبيانات المشفرة بالشبكة العصبية المكافئة والتي تقوم بدورها بعملية فك التشفير، من أجل تعقيد التحليل الهندسي العكسي للبرنامج.

ثالثا، اقترحنا آلية مقاومة العبث والتعديلات الغير مرغوب بها والمبنية على أساس التشويش والتنويع. الالية المقترحة تدمج عده تقنيات معا من أجل إحباط العبث وزيادة صعوبات التحليل العكسي الثابت والتحليل الديناميكي للبرمجيات.

الحماية التي تقدمها التقنيات المقترحة محصنة ضد التحليل الثابت، والتحليل الديناميكي، والتلاعب. كما أنه لا يمكن بسهولة إزالة وفك التشفير والتشويش المطبقين باستخدام التقنيات المقترحة من البرمجيات المشفرة، حيث أن المخترق سوف يستهلك الكثير من الوقت والجهد من اجل إزالة هذا التشفير. علاوة على ذلك، تؤكد التقييمات والتجارب التي تم اجراءها أن آثار التشويش في نظامنا المقترح يزيد بشكل كبير من الصعوبات في عملية كشف البرمجيات المشفرة. من ناحية أخرى، تؤكد تقييمات الأداء أن تقنياتنا المقترحة تحمي البرمجيات بكفاءة مع زيادة مقبولة في وقت التنفيذ واستخدام الذاكرة.

الملخص

إطار لحماية البرمجيات قائم على تقنيات تشويش الكود

حماية الملكية الفكرية من العبث والتحليل العكسي مسألة ملحة لكثير من مصممي البرمجيات، حيث يعتبر الوصول غير المشروع إلى البيانات الحساسة شكل من أشكال التعدي على حقوق الملكية الفكرية. لذلك يطبق مطورو البرمجيات تقنيات حماية مختلفة من اجل معالجة هذه المسألة الهامة وحماية برمجياتهم من السرقة أو التعديل. لكم للأسف فان الكثير من التقنيات المستخدمة ضعيفة، لأنها معرضة للتحليل الديناميكي والثابت، في حين أن باقي التقنيات مكلفة جدا لأنها تؤثر سلبا على كفاءة وسرعة استجابة وتشغيل البرمجيات المشفرة. علاوة على ذلك، فان هذه التقنيات غالبا ما تكون غير جيدة لأنها تعتمد على "الأمن من خلال الغموض" التي قد تردع بعض المخترقين، ولكنها توفر حماية قليلة جدا ضد الخصوم المحترفين. أضف الى ما سبق أن العواقب ستكون وخيمة والخسائر ستكون فادحة إذا نجح الخصم في استخراج وإعادة استخدام خوارزميات البرنامج. علاوة على ذلك، لا تزال الهندسة العكسية تشكل تهديدا كبيرا لمطوري البرمجيات وخبراء الأمن.

في هذه الأطروحة، اقترحنا إطار لحماية البرمجيات قائم على تقنيات تشويش الكود من أجل حماية البرامج من التحليل العكسي والتعديلات غير المرغوب فيها.

أولا، اقترحنا تقنية لتشويش وتشفير البرامج المبنية على لغة الجافا من أجل حمايتها من التحليل العكسي الثابت. التقنية المقترحة تدمج ثلاثة مستويات من التشويش. شفرة المصدر، تشفير البيانات، وتشفير ملفات الجافا المحملة على الجهاز عند تشغيل هذه البرامج والتي تسمى بالبايت كود (Bytecode). من خلال الجمع بين هذه المستويات، حققنا مستوى عال من التشويش والتشفير، الأمر الذي يجعل من فهم أو فك تجميع البرامج المشفرة عملية معقدة جدا أو غير قابلة للتطبيق.