STANDALONE STATIC BINARY EXECUTABLE REWRITING FOR SOFTWARE PROTECTION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ÖZGÜR SAYGIN BICAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JUNE 2015

Approval of the thesis:

STANDALONE STATIC BINARY EXECUTABLE REWRITING FOR SOFTWARE PROTECTION

submitted by **ÖZGÜR SAYGIN BICAN** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering** _____

Dr. Onur Tolga Şehitoğlu
Supervisor, **Computer Engineering Department, METU** _____

Examining Committee Members:

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Department, METU _____

Dr. Onur Tolga Şehitoğlu
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Ertan Onur
Computer Engineering Department, METU _____

Assist. Prof. Dr. Selim Temizer
Computer Engineering Department, METU _____

Assoc. Prof. Dr. Osman Abul
Computer Engineering Department, TOBB ETU _____

Date: _____

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: ÖZGÜR SAYGIN BICAN

Signature :

ABSTRACT

STANDALONE STATIC BINARY EXECUTABLE REWRITING FOR SOFTWARE PROTECTION

Bican, Özgür Saygın
M.S., Department of Computer Engineering
Supervisor : Dr. Onur Tolga Şehitoğlu

June 2015, 59 pages

This study introduces a static binary rewriting method for improving security of executable binaries. For software security, when the network and host-based precautions are passed by the adversary or they are not present at all, the software has to defend itself. Nevertheless, applying software protection methods during software development requires extra source code development and know-how. Furthermore, these methods inherently make the software undesirably complex. Applying these methods after compilation of the software will decouple the software development and protection processes. Binary rewriting is such a method that externally modifies an executable file in order to make binary hard to reverse engineer and tamper. Along with software protection, binary rewriting is also applied on other areas such as binary instrumentation and semantic patching etc. that are out of the scope of this study. Some prior proposed approaches use a special compiler and/or linker and some others use a third party commercial disassemblers to make analysis on the binary file, making process highly dependent on performance of these tools. In this study, a standalone static binary rewriting framework that can work directly on the output of the compiler without any third party disassembler or special compiler/linker dependency is developed. The framework uses debug information in binary to get function locations, and then relocates functions, then update the references to point to the new addresses in the binary. The implementation is tested on various open source software

written in C and C++ for performance overhead. Then, as a case study, a software protection method is applied to a program using our framework, and the security of resulting binary is compared in terms of how control flow graph reveals information about software structure.

Keywords: Static Binary Rewriting, Software Protection, Software Security, Software Obfuscation, Software Tamper-Proofing

ÖZ

YAZILIM KORUMA İÇİN BAĞIMSIZ STATİK İKİLİ ÇALIŞTIRILABİLİR DOSYA TEKRAR YAZIMI

Bican, Özgür Saygın
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Dr. Onur Tolga Şehitoğlu

Haziran 2015 , 59 sayfa

Bu çalışma çalıştırılabilir ikili dosyaların güvenliğini arttırmak için bir statik ikili dosya tekrar yazım (SİDTY) metodu tanıtmaktadır. Yazılımın güvenliği için, ağ ve bilgisayar tabanlı önlemler saldırgan tarafından aşıldığında yada hiç olmaması durumunda yazılımın kendi kendini savunması gerekir. Ancak, yazılım koruma yöntemlerinin yazılım geliştirme aşamasında uygulanması ek kaynak kod geliştirme ve teknik bilgi gerektirmektedir. Ayrıca, bu yöntemler yapıları gereği yazılımı istenmeyen bir seviyede karmaşık hale getirmektedir. Bu yöntemleri yazılım derlendikten sonra uygulamak, yazılım geliştirme sürecini ve yazılım koruma sürecini birbirinden ayrıştırır. SİDTY tersine mühendisliği ve istenmeyen değişikliği zorlaştırmak için çalıştırılabilir dosyaları dışarıdan değiştiren bir yöntemdir. İkili dosya tekrar yazımı, yazılım koruma ile birlikte bu çalışmanın kapsamının dışında olan ikili dosya enstrümantasyonu, semantik yamalama gibi alanlarda da kullanılır. Önceki bazı çalışmalar özel derleyici ya da bağlayıcı kullanan yaklaşımlar önermektedir. Bazıları ise üçüncü parti ticari ayrıştırıcılar kullanarak ikili dosya üzerinde analiz yapmaktadır. Bu durum süreci büyük oranda bu araçların başarısına bağımlı kılmaktadır. Bu çalışmada herhangi bir ayrıştırıcı yada özel derleyici/bağlayıcı bağımlılığı olmayan, doğrudan derleyicinin çıktısı üzerinde çalışabilen bağımsız bir SİDTY çatısı geliştirilmiştir. Geliştirilen çatı, fonksiyonların yerini tespit edebilmek için hata ayıklama bilgisi kullanmaktadır, daha sonra fonksiyonları taşıyıp referansları yeni adresleri gösterecek şekilde güncel-

lemektedir. Uygulama, performans ek yükü için C ve C++ dillerinde yazılmış çeşitli açık kaynak yazılımlar üzerinde test edilmiştir. Daha sonra, örnek çalışma olarak, uygulamamız kullanılarak bir yazılım koruma yöntemi bir programa uygulanmış ve ortaya çıkan ikili dosyanın güvenliği kontrol akış grafiğinin yazılım yapısı hakkında verdiği bilgi açısından karşılaştırılmıştır.

Anahtar Kelimeler: Statik İkili Dosya Tekrar Yazımı, Yazılım Koruma, Yazılım Güvenliği, Yazılım Karıştırma, Yazılımın Değişikliğe Karşı Korunması

To my family, my lovely girlfriend and people who are reading this thesis

ACKNOWLEDGMENTS

I wish to thank my advisor and committee members who were more than generous with their precious time. A special thanks to my advisor Dr. Onur Tolga Şehitoğlu for his continuous support and guidance during the entire process. Thank you İsmail Hakkı Toroslu, Ertan Onur, Selim Temizer, Osman Abul for agreeing to serve on my committee.

I would like to acknowledge and thank to my company MilSOFT Software Technologies for allowing me to continue my education and conduct my research while I am working there and providing any assistance requested. Special thanks goes to Serhat Toktamışoğlu, my supervisor at work, for his guidance and my colleagues for their support at the development.

Finally, I would like to acknowledge and thank to Türkiye Bilimsel ve Teknolojik Araştırma Kurumu (TUBİTAK) for their *2228-A Son Sınıf Lisans Öğrencileri için Lisansüstü (Yüksek Lisans/Doktora) Burs Programı* which provided me financial support by their scholarship during my two tears long Master of Science education.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiv

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xvi

CHAPTERS

1 INTRODUCTION 1

1.1 Software Protection 1

1.1.1 Software Protection Methods 3

1.2 Motivation 5

1.3 Challenges 5

1.4 Former Studies 7

1.5 Scope of the Thesis 8

2 BACKGROUND 11

2.1 x86 Assembly Language 11

2.2 Stack 13

2.3 ELF Format 15

2.4 Debug Information Format DWARF 17

2.5 Disassembler 17

2.6 Assembler 19

3 IMPLEMENTATION 21

3.1 Analyzing ELF Format 22

3.2 Finding Functions 22

3.3 Creating New Executable Section 23

3.4 Disassembly 25

3.5 Assembly 27

3.6 Rewriting 28

3.6.1 Handling Incoming Function Calls 30

3.6.2 Handling Dynamic Jumps 31

3.6.2.1 Finding Possible Dynamic Jump Targets 33

3.6.2.2 Redirection Map For Dynamic Jump Targets 35

3.6.2.3 Dynamic Jump Target Translation At Runtime 38

3.6.3 Handling PIC 40

4 TESTS AND RESULTS 43

4.1 Case Study 47

5 CONCLUSION 53

5.1 Future Work 54

REFERENCES 57

LIST OF TABLES

TABLES

Table 2.1 **if-else** Statement Example In x86 Assembly Language 12

Table 3.1 Example Program Header 23

Table 3.2 Before Creating New Section 25

Table 3.3 After Creating New Section 26

Table 3.4 Example Dynamic Jump With and Without Computation 39

Table 3.5 Translation Examples (Without Extra Array) 40

Table 3.6 Translation Examples (With Extra Array) 41

Table 3.7 Example PIC 42

Table 4.1 Numbers About The Tested Binaries 45

Table 4.2 Execution times in seconds; o: original, r: rewritten 46

LIST OF FIGURES

FIGURES

Figure 1.1 Overall Architecture 10

Figure 2.1 Stack Structure (Adapted from [5, Ch. 3.7.1]) 14

Figure 3.1 Rewriting Example: (a) Step 4 - Disassembled Function, (b) Step 7 - Function At New Address (old address + 0x1000) 31

Figure 3.2 Rewriting Example: (a) Step 8 - Initial Relocation Map, (b) Step 10 - New Code Pieces Inserted 32

Figure 3.3 Rewriting Example: (a) Step 13 - Updated Relocation Map, (b) Step 14 - Static Jumps Fixed 33

Figure 3.4 Rewriting Example: Step 11 - Redirecting Incoming Function Calls 34

Figure 3.5 Rewriting Example: (a) Function With A Dynamic Jump, (b) Redirection Entries For Possible Jump Targets 0x804850d, 0x8048513 and 0x8048519 36

Figure 3.6 Rewriting Example: (a) Redirection Entries For Possible Jump Targets 0x804850d, 0x804850e, 0x80485013, 0x8048516, 0x8048519, (b) Redirection Entries In Extra Array 38

Figure 4.1 Tested Binaries and Execution Times 47

Figure 4.2 Execution Time of A Function With A Dynamic Jump (for a Doxygen function) 48

Figure 4.3 Execution Time of A Function Without Dynamic Jump (for a Doxygen function) 49

Figure 4.4 Case Study: Original Control Flow Graph 51

Figure 4.5 Case Study: Control Flow Graph After Branch Function Method . . 52

LIST OF ABBREVIATIONS

IDS	Intrusion Detection System
IPS	Intrusion Prevention System
CPU	Central Processing Unit
IA32	Intel Architecture 32
AMD	Advanced Micro Devices
ELF	Executable and Linkable Format
DWARF	Debugging With Attribute Record Formats
NASM	Netwide Assembler
PIC	Position independent code
gcc	Gnu Compiler collection
ASLR	Address space layout randomization

CHAPTER 1

INTRODUCTION

1.1 Software Protection

With the developing technology and the growth in number of devices that are connected to the Internet, the number and effect of the cyber-attacks have increased as well. Therefore, providing confidentiality, integrity and availability of the information and services has become harder and more crucial.

These facts lead the cyber-attacks to become more professional. Unlike the past, cyber-attacks are not organized by individual hackers or script kiddies anymore. Behind the cyber-attacks, now, there are professional groups with specific targets and purposes. In addition, nowadays, hacktivists — a group of hackers who organize cyber-attacks for protesting, activism, and civil disobedience — and even countries organize cyber-attacks.

In 2010, a computer worm, called Stuxnet, was discovered in Iran [15]. Stuxnet targeted Iran's nuclear centrifuges and it is considered as first cyber weapon. In Stuxnet, a couple of *zero day vulnerabilities*, the vulnerabilities previously unknown so that the developers cannot create a patch, are used. It accomplished to ruin some of Iran's nuclear centrifuges. Investigating this kind of professional attacks clearly shows that zero day vulnerabilities make every system prone to be hacked. It is important to note that zero day vulnerabilities are, by definition, caused by the software itself rather than the network perimeter.

In addition to cyber-attacks, another important issue targeting computer systems se-

curity is the reverse engineering and cracking of the software. Today, there are a lot of critical software in defense industry such as radars, command and control systems etc. including a lot of critical information that has to be kept secret. Consider a fighter jet of a country crashed in another country. Reversing the software of the jet may reveal crucial information compromising country defense. Furthermore, these critical systems should be prevented from *tampering* [1]. Consider tampering a radar software and changing its behaviour to make it blind for some countries' jets.

Software companies' products involve secrets as well. The companies' intellectual property, such as new algorithms, should be protected. In addition, software companies should prevent the illegal copies of their software. In other words, they should prevent tampering the software and bypassing license check mechanism. Preventing illegal copies has important commercial significance for software companies.

For example, Skype[2] is a software product that provides video chat and voice calls from computers and mobile devices via the Internet to other devices or telephones/smartphones. Skype is well protected with some techniques against reverse engineering. With the help of this protection it dominated the market. It remained uncracked and not-reversed for years so that after that time period cracking or reversing it did not mean too much to developers of Skype. However, it is reversed by professional hackers in 2006 and presented in Black Hat Europe [2]. These facts show that in addition to securing the network perimeter and the hosts, securing the software itself is really important. This brings us to a forgotten topic at cyber security, named *Software Protection*.

Software Protection is considered as the last point of defense in cyber security. When the firewall, Intrusion Detection System (IDS), Intrusion Prevention System (IPS), anti-viruses etc. are passed by adversary or they are not present at all, the software has to defend itself [30]. Moreover, in order to accomplish some of the attacks, the adversary has to examine the software to find vulnerabilities. Software protection methods make examining, analyzing and cracking the software more complicated. Therefore, finding and exploiting the zero day vulnerabilities become harder.

[1] Tampering is to make unauthorized modification that alters the software execution and behaviour.
[2] `http://www.skype.com/en/`. [Last accessed on June 26, 2015].

The purpose of the software protection is **(i)** keeping the internals of the software as secret, **(ii)** preserving the functionality and the behaviour of the software, **(iii)** preventing the unwanted copies, and cracking of the software.

Zero day vulnerabilities are found by analyzing and investigating the software. By keeping the internals of the software as secret, software protection makes it harder to find these vulnerabilities. Moreover, it protects the intellectual property, because it is harder to reverse engineer the software. Keeping internals as secret also helps to preserving the functionality and the behaviour of the software, because finding the function points and understanding the behaviour is harder.

Preserving the functionality and the behaviour of the software prevents the unwanted changes caused by the targeted attacks. Furthermore, it makes harder to crack the software's license mechanism.

1.1.1 Software Protection Methods

Software protection methods can be categorized according to their purpose. Some methods mainly aim to hardening the software against reverse engineering whereas some of them try to detect and prevent the changes in the behaviour and functionality of the software. Moreover, there are software labeling methods such as fingerprinting, watermarking etc. These methods can be used to prove the software is cracked, copied illegally and ownership/origin of the software. Detailed information can be found in [24].

Two examples for these software protection method categories which are named as *obfuscation* and *tamper-proofing* are explained below.

Software Obfuscation [24, Ch. 4-5] is hiding or making the internals of the program ambiguous. This can be done statically and dynamically. The dynamic obfuscation makes changes in the program at runtime, whereas static obfuscation does it at compile time or in the file after compilation or before program starts to execute. Obfuscation is accomplished differently in different programming languages. Compiled programming languages such as C, C++ are compiled to

machine codes that run directly on the processor. In this case, compiled code is kind of already obfuscated since it is nearly impossible to return back to original source code from machine code. Nevertheless, the *disassemblers* can generate accurate assembly code from machine code. Besides, they can even generate the control flow graph, call graph, and data flow graph with pretty good accuracy. In addition to disassemblers, with the help of debuggers the software can be analyzed dynamically. Therefore, reversing is not a hard task for a reverse engineer when the software is not obfuscated. Obfuscation of compiled programming languages includes techniques that misdirect disassemblers and reverse engineers. This task accomplished by mangling the machine instructions and data in the software without changing any behaviour and functionality of the software. On the other hand, in the interpreted programming languages which require virtual machine such as JAVA, intermediate bytecodes are produced. In this case, returning back to the original source code is quite easy by using *decompilers* which translates low level representation of the code into high level programming language form. For this case, in addition to the techniques for compiled programming languages, name mangling can be applied as an obfuscation technique.

Example obfuscation methods can be inserting bogus control flow [8], control flow flattening [18], self-modifying state machine [24, Ch. 6.2.2 p. 266-276] etc.

Software Tamper-proofing [24, Ch. 7] is making harder to modify the software. This is accomplished by self-checking the content of the software. By tamper-proofing, when an adversary try to change the instructions or behaviour of the software, the software either can repair this modification or be aware of this change and take action about it such as crashing, reporting or deleting itself etc. *Guards network* method by Atallah et al [6], can be an example to this category.

1.2 Motivation

Applying software protection methods during software development requires extra source code development and know-how. These methods usually do not provide much protection individually. Therefore, more than one protection method should be applied with layered approach. Furthermore, these methods make the software more complex inherently which is an unwanted situation in software development process.

If these methods can be applied after compilation of the software, the software development and protection processes will be decoupled. This task requires making changes and inserting new code pieces after compilation. This process can be called as *static binary rewriting*. It is called as static because, the changes are made on the binary file, not at the runtime.

In the thesis we focus on the executable binary files which are produced by compiling the source code written in compiled programming languages. These executable binary files include the native machine codes which run on the Central Processing Unit (CPU). Moreover, the executable file formats are different in different operating systems. Therefore, static binary rewriting process depends on the CPU architecture and the operating system. However, most of the challenges and steps to be followed are common. Therefore, by making minor changes, an existing solution for one architecture and operating system can be adapted to other architectures and operating systems.

This thesis aims to apply software protection methods without source code modification, propose a solution and address the challenges in static binary rewriting for the software written in compiled programming languages.

1.3 Challenges

A successful binary rewriting process has to preserve the software's all the existing functionalities and capabilities without corrupting the program. To achieve this there are some challenges to be addressed.

The memory addresses in the binary file are pre-computed and flow of execution is highly dependent on the memory addresses. It is a challenging task to make changes and insert new code pieces into the executable binary file without corrupting the memory addresses.

The executable binary files include all information required to load, dynamically link and optionally debug and profile the program. Executable file formats supported by different operating systems are strict formats including tightly coupled variable, symbol and code information on complex structures. After binary rewriting process the information in the file should be consistent and the file should be consistent with the format.

The challenges mentioned above are the main problems that are need be solved. In addition to those, the sub-problems in static binary rewriting can be listed as the following:

Space to insert new code pieces: The binary file is loaded to the memory according to mapping information in the file. These mappings depend on the file offsets. Hence, the positions of the bytes in the file are important. However, inserting new code pieces will make the offsets of the following bytes to be shifted. Therefore, this will create inconsistencies in the file.

Accuracy of disassembly: *Disassembly* is used to analyze the existing executable machine code before binary rewriting. The changes in the binary file are determined based on the output of the disassembly. Therefore, the success of the binary rewriting is highly dependent on the accuracy of disassembly.

Safely altering the addresses: When a new code piece is inserted, the memory address of the following bytes will be shifted because only one byte can be in a memory address. Moreover, the inserted codes have to be part of the execution of the program. This can be done either by inserting the new code piece into a function or jumping to the new code piece from a function and returning back (which also requires inserting a couple of bytes). In either way the inserted bytes will shift the memory address of the following bytes. This situation corrupts the program execution because all the operations which are dependent on

memory addresses will be wrong.

1.4 Former Studies

In binary rewriting there are some *dynamic binary rewriting* solutions such as PIN [20], Valgrind [25] and DynamoRIO [4] which make modifications at runtime. Dynamic binary rewriting, is generally is used for monitoring the software execution or measuring the performance. In this approach the changes last only throughout the execution. Nevertheless, our purpose is to make modifications in the file statically and these modifications should be permanent. Therefore the static binary rewriting is more appropriate for us.

In prior studies on static binary rewriting such as Sutter et al, 2005 (Diablo) [13] and Muth et al, 2009 (Alto) [23], they can operate on the binaries that are generated by the special compilers. They implemented their own compilers to make modification at link time. However, the disadvantages of these studies are the dependency to special compilers.

More lightweight approaches exist which do not require special compilers such as Romer et al, 1997 [27] and Hunt et al 1999 [17]. In these studies in order to insert code pieces to execution, they create *trampolines* which include the code to be inserted and then change the execution flow to access these trampolines. In other words, they do not insert the code piece in the code of the program. Instead, they write the code pieces at the end of the binary file, then in order to execute these code pieces they change the execution of the program by inserting branch instructions. Nevertheless, this approach is not successful at inserting code piece anywhere in the execution flow of the program. Moreover, it does not have enough flexibility to apply software protection methods to binary file. For some of the software protection methods it may not be enough to execute the new code pieces. In order to apply some software protection methods (for instance: Inserting garbage bytes to mislead disassembly [3]), it is required the inserted code pieces to be placed into the code of the program which is not supported by this approach.

The study Laurenzano et al, 2010 (PEBIL) [19] implemented a static binary rewriting

tool for Linux based operating systems without using any special compiler. In the implementation, they use *symbol table* which is a part of the debug information. In their approach, the main challenge is handling dynamic jumps. To overcome this challenge, they use backward analysis on the code in order to determine the targets of the dynamic jumps which is a heuristic method. This may cause corruption of the file when the analysis of the dynamic jump target cannot be accurate.

Wartell et al, 2012 [31] and Deng et al, 2013 [14] is the most inspiring two studies for us in static binary rewriting. They do not use any specific compiler. In order to handle dynamic jump instructions they use runtime translation approach for redirecting dynamic jump targets. However, they do not use debug information and they have dependency on IDA [26] which is a commercial disassembly tool.

Deng et al, assumes the program is perfectly disassembled by IDA which is impractical. As cited in Wartell et al. the perfect disassembly cannot be achieved in general. Deng et al, makes an analysis on the binary for finding possible dynamic jump targets and proposes some criteria for pruning these targets. We inspired from this analysis method and pruning criteria for finding possible jump targets.

On the other hand, Wartell et al, uses IDA for finding possible dynamic jump targets which brings a dependency to a commercial disassembly tool. He uses the old space of the code that is relocated to store the entries which are used in runtime dynamic jump redirection. Therefore, in this approach if a data is interpreted as a dynamic jump target by IDA it may corrupt the data because it overwrites some values for runtime redirection of dynamic jumps. In this study, we inspired from the idea of storing entries for redirection at old space of the relocated code.

In our solution we aimed to remove the IDA dependency and perfect disassembly assumption in these two studies.

1.5 Scope of the Thesis

In this thesis, we implemented a standalone function based static binary rewriting framework for x86 32 bit Linux based operating systems and for the software written

in compiled languages. It has no third party tool dependency nor it uses any specific compiler and/or linker. It works directly on the output of the compiler.

In this framework, the inputs are the executable binary file, debug information of the binary file, and code insertion map (function descriptions in which the code pieces will be inserted, code pieces to be inserted and the offsets to where the code pieces will be inserted). The output is the rewritten binary file. The debug information is used for finding the codes of the function whose descriptions (function name and if available *namespace* and *class* name) are provided in input. This gives opportunity to make different operations on the different functions. Note that our motivation is applying software protection methods. Hence, instead of applying same methods to all functions, being able to apply different methods to different function is an advantage. Moreover, since the functions can be selected, the performance overhead of the protection can be adjusted according to the user's will. Another use of debug information is increasing the disassembly accuracy. Details about this will be explained in the sections 2.4 and 3.2.

The framework inserts each code piece to its given offset of a function according to the code insertion map. The framework supports inserting multiple code pieces to multiple offset. The offset is the relative offset according to the beginning of the function.

In order to accomplish static binary rewriting, we made some fair assumptions about the input executable binary file. First, the input file should not be obfuscated. As we mentioned before, obfuscation hardens the disassembly and our framework makes analysis depending on the disassembly. Secondly, the input binary file should not use *self-modification*. Self-modification is a technique which is used generally by malwares. In this technique the program modifies its code during runtime. Therefore, insertion of new code piece will corrupt the program because the modifications which are done at runtime will not be done correctly. The last assumption about the input binary file is that the input file should not be hand crafted; it should be output of a compiler. In other words the source code of the input file should not include hand coded assembly code. In this case the programmer can go beyond the restrictions of the programming language such as jumping to middle of a function from

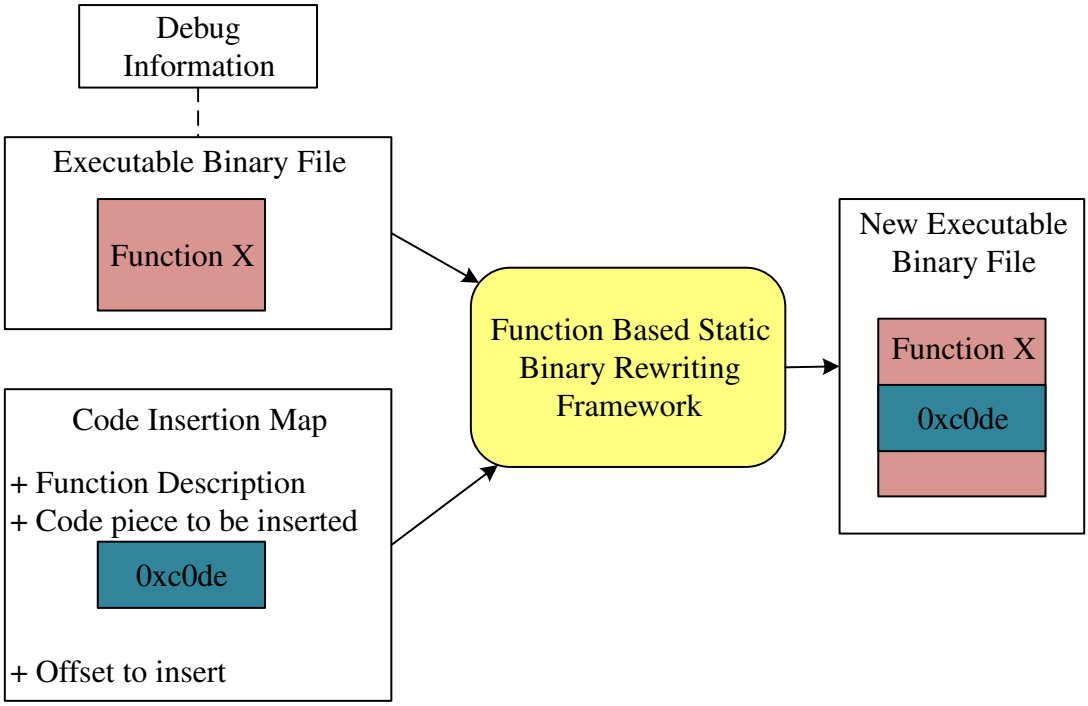


Figure 1.1: Overall Architecture

another function. Hence, these cases may cause our framework to fail in preserving the existing functionality.

We tested our framework against open source C and C++ programs. In our experiments we force rewriting of nearly all of the functions that are called in these programs. We tested if the programs produce same output after they were processed by our framework. Moreover, we measured performance overhead of binary rewriting in these programs.

In chapter 2 some important terms and concepts are defined. Next, our approach is described and the challenges of it are addressed in the chapter 3. Then, the experiments and the results are explained in chapter 4. Finally, the conclusion and the future work discussions are provided in the chapter 5.

CHAPTER 2

BACKGROUND

2.1 x86 Assembly Language

An *assembly language* is a low level programming language which corresponds to a human readable symbolic version of the machine code. Therefore, it is highly dependent on the CPU architecture. The interpretation or compilation process of assembly language is named as *assembly*. Outcome of this process is the executable machine code, to be executed directly by the CPU. For this reason, in contrast to high level programming languages, an assembly language is not portable among different CPU architectures. Assembly languages use mnemonic to represent each instruction in order to ease reading and writing. Each mnemonic corresponds to an instruction which has a corresponding machine code instruction for the CPU.

x86 architecture introduced by Intel for their 8086 CPU [11, Ch. 2.1]. The first design was a 16 bit architecture which is the ancestor of Intel Architecture 32 (IA32). 16 bit architecture was improved to become IA32 that supports 32 bit data and addresses. Later, IA32's 64 bit extension was developed by Advanced Micro Devices (AMD). x86's 64 bit version is referred as x64. This successive processor line has been designed to be backward compatible, in order to be able to run code compiled for any earlier version. This study targets the 32 bit version of x86 and we refer it as x86.

In x86 assembly language there are eight registers (**eax**, **ebx**, **edx** etc.) to hold 32-bit integer values [11, Ch. 3.4]. In order to track the address of the next instruction to execute, there is a register named **eip** - *instruction pointer*. Plus, there are a set of floating point registers to store floating point data. Furthermore, there are *flags* such

as zero, carry, and overflow. These flags help in conditional branching and arithmetic operations. Some instructions have side affects of modifying flags.

In x86 each instruction has an *opcode* that serves as the id for that instruction. Some of the instructions have arguments which are called as *operands*. The opcodes can be at most 2 bytes and the operands have variable size. Therefore, the instructions have variable sizes.

Like almost all imperative languages, in addition to data manipulation instructions, x86 assembly language has instructions for branching. It has **call** (call function at the address given in operand) instruction in order to handle function calls and **ret** (return from the function) instruction for returning from a function [5, Ch. 3.7.2]. The unconditional and conditional jumps are handled by **jmp** (jump to the address given in operand) and ***jcc*** (conditional jump - jump if the condition is met) instructions respectively where cc is the condition (it can be e - equal, ne - not equal, g - greater than, le - less than or equal to etc.) [5, Ch. 3.6]. The conditions mostly checked by **cmp** (compare two operands) instruction. This instruction modifies some of the flags according to its operands. According to which flags are set, the result of the compare operation determines whether the following conditional jump will be skipped or not. Not skipping the jump is conventionally called as taking the jump which results in the address given in the operand is loaded to the instruction pointer as next instruction. Intuitively, skipping the jump is conventionally called as not taking the jump. Table 2.1 shows an example of **if-else** statements written in C and the corresponding x86 assembly language code. Detailed information about x86 and full list of instructions can be found in [11, 12].

Table 2.1: **if-else** Statement Example In x86 Assembly Language

C code piece	The corresponding assembly code

`if (x == 0)`
`  //if statements`
`  else`
`  //else statements`
` `
	`  cmp eax, 0  ;eax holds the x value`
	`  jnz else`
	`  ;if action statements`
	`else:`
	`  ;else action statements`

In x86 assembly language the **call** and **jmp** instructions can have the address operand as literal. This type of branch instructions can be referred as *static branch instructions*

such as **jmp 0x1234**. This type of branch instructions' target address, the address that the instruction will direct to, can be determined without any backward analysis on the operand. On the other hand, there are also *dynamic branch instructions* whose operand is a register or a memory location such as **jmp [eax]** or **call ebx**. This kind of branch instructions are also referred as *indirect branch instructions*. Since the value at the register or memory location is determined at runtime of the program, this type of instructions' target address may not be known without dynamic analysis or backward analysis on the operand. For example, when the C or C++ **switch** statements are compiled, they are generally handled by dynamic jumps [5, Ch. 3.6.6]. The addresses of the statements in the case conditions are held in a jump table. Therefore, the corresponding statement's address is obtained from the jump table at the runtime according to the matched case.

One of the important challenges of the static binary rewriting is handling the dynamic branch instructions. When an instruction's address in the program is changed after compilation, the dynamic branch instructions' target address may become a dangling address. This study aims static rewriting which involves rewriting without dynamic analysis. Therefore handling this case without prior information on run time behavior is a challenging task.

2.2 Stack

In x86 the programs uses the program *stack* in order to handle function calls. During runtime of the program a portion of the stack, named as *stack frame*, is allocated for the running function. The stack frame can be used for parameter passing, as a local storage and saving registers for later restoration. Figure 2.1 shows the conventional stack structure [5, Ch. 3.7.1]. In x86 the **esp** register is called as *stack pointer* which is used to hold the address of the top of the stack frame and the **ebp** register is called as *frame pointer* which holds the address of the bottom of it. The stack pointer can move during execution of the function. As the figure shows the stack frame grows towards decreasing addresses. In other words as the stack frame grows the address in the **esp** decreases. Intuitively, as the stack frame shrinks, the address in the **esp** increases. x86 has **push** and **pop** instructions in order to manipulate stack frame. The

push instruction decreases the **esp** by four and writes the operand of it to top of the stack frame. On the other hand, the **pop** does the inverse of the **push**, it reads the top of the stack frame into its operand and increases the **esp** by four.

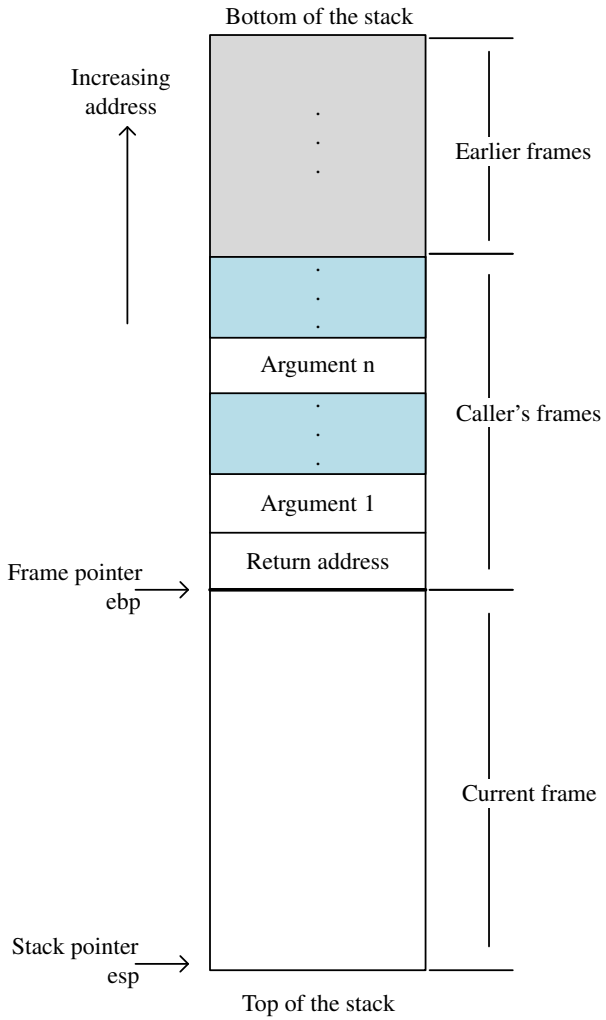


Figure 2.1: Stack Structure (Adapted from [5, Ch. 3.7.1])

Function calling and returning from the functions is done by the **call** and **ret** instructions, respectively [5, Ch. 3.7]. The **call** instruction actually does two operations. First, it pushes the address of the next instruction. Then, it loads the operand of it to instruction pointer, so the execution will continue from the called function. The pushed address is referred as *return address* which is actually the address that the execution will continue after the called function returns. The parameters of the function can be pushed into the stack (It is not mandatory to use the stack. The registers may be used for parameter passing) before the function call so the called function can read

the parameters from the stack. The **ret** instruction pops the return address pushed by the **call** instruction to instruction pointer, so the execution will continue from return address. The return value conventionally is held in the **eax** register; however, the stack can be used as well.

As it is mentioned before, the stack frame is used for storing the local variables also. When a local variable is defined, a place to hold the value is opened in the stack frame. Opening space actually means growth of the stack frame. Before the function returns, the allocated stack frame for the function is released back with the help of **leave** instruction. Therefore, when the function returns the local variables become unreachable.

2.3 ELF Format

Operating systems have special file format for binary files. In these formats the information about the binary file is stored according to the specific structures of the format. The format of binary files in Linux based operating systems is named as Executable and Linkable Format (ELF). This is a common format for executable files, static or dynamic libraries, object files etc. Some of the most important information that is held in the structures in ELF format:

ELF Header contains the general information about the file. For example file type, entry point, program header offset, section header offset, program header entry count and section header entry count information is held in ELF header. File type indicates the type of binary file such as executable file, shared object file etc. Entry point indicates what the virtual address of the instruction, that will be executed first, is. Program header offset and section header offsets indicates the offset where these headers start at file. The entry counts indicate the number of the entries in these headers.

Program Header contains information about segments in the file. The segments are mapped into the memory when the binary loaded into the memory. Program header indicates the type of the segment, how the segments will be mapped into

the memory and the read, write, execute rights of the segment. One segment can be located in another segment in the memory.

Section Header contains information about sections. Sections are the subparts of the segments and each section has to be mapped to at least one segment. For each section there is an entry for it in the section header. In the entries each section have file offset, virtual address, size, alignment and read, write, execute rights information.

Relocation Entries are the entries that are used the process of adjusting the code and data in the program. The information about the relocation process is actually held in some specific sections. Relocation entries tells loader of the operating system how to modify some parts of the program so that the executable or shared object (.so) files run properly when it is loaded to the memory. For example, a shared object file can be loaded different virtual addresses. Thus, the direct data accesses, direct function calls etc. has to be adjusted when it is loaded to the memory. Otherwise, these addresses will be dangling addresses.

In addition, ELF holds a lot of other information about the program such as string table for holding the strings, procedure linkage table for function calls from external libraries, debug related information etc. The detailed information can be found in reference [10].

For reading and parsing the ELF format, in Linux the command line tool named **readelf**[1] can be used . It parses the ELF format and gives the information at ELF header, section header, program header etc. as output.

In an executable binary file generally there is one **LOAD** type segment which has read and execute right for executable parts of the binary and there is another **LOAD** type segment which has read and write right for the data parts of the binary. The executable machine code is generally located in an executable section named **.text** which is located in the executable **LOAD** type segment. The code lies in **.text** section generally with no or a little alignment space between functions. Thus, when a new code piece inserted, the following bytes should be shifted. However, since the

[1] `http://linux.die.net/man/1/readelf`. [Last accessed on June 26, 2015].

addresses in the code bytes will be dangling when they are shifted, the program will be corrupted. Plus, fixing the shifted addresses is a very challenging task. Therefore, instead of shifting, creating a new executable section will be a better approach.

2.4 Debug Information Format DWARF

After the compilation usually the most of the source code level information such as variable names, function names, function definitions etc. will be removed because, they become redundant at executable machine code level. However, when the program is wanted to be debugged, this information is useful. In binary level debugging connecting the assembly code and the virtual address values with the source code like function names and variable names, is necessary. Otherwise, it would be really hard to find where each function starts and ends or what the variable's addresses are.

For debug purposes in ELF optionally there are debug related sections which can be referred as debug information. The debug information has a format named Debugging With Attribute Record Formats (DWARF) [9] and the debug information can be in a separate file.

For static binary rewriting the information in the debug information is helpful. Knowing function start and end addresses makes possible to relocate a function for opening space for the code piece to be inserted.

2.5 Disassembler

Disassemblers are the tools for translating executable machine code into assembly language. Due to the strict correspondence between assembly language and the machine code, disassemblers can do successful disassembly operation. However, there are some problems that prevent disassembly with 100% accuracy. For x86 assembly language, the instruction sizes are not fix, so each instruction's size depends on the opcode of it. Thus, starting point and the address of the next instruction to be disassembled is crucial.

Disassemblers use two approaches for disassembly operation [28].

Linear sweep is the straightforward approach that disassembles instructions in a linear way. It assumes each instruction is located next to the previous one. Once it disassembles one instruction, it knows the size of the current instruction. Thus, it calculates the beginning of the next instruction. Starting with an instruction it continues with the next one without considering the control flow of the program. Since, it does not consider the control flow, linear approach fails in some cases. For example, when there is data after a unconditional jump that jumps over the data, it assumes the data as instruction and disassembles the data. The data may correspond to some instruction that has a different size from the size of the data. Thus, the disassembler will be misled about the beginning of the next instruction. In Linux based operating systems disassembler named **objdump**[2] uses this approach.

Recursive traversal takes into consideration the branch instructions. It disassembles instruction by instruction like the linear approach. However, it takes into account the control flow of the program. When it reaches to a branch instruction, it continues from the target address first. Then, it returns back to the branch instruction and continues with the next instruction after the branch instruction. By this way the accuracy of disassembly operation increases.

In addition, there may be improvements on these approaches. However, there are still challenges on the disassembly operation. One of the issues is the code interleaving with data. Compilers may insert data between the code. In some cases even recursive traversal approach may have problems at providing 100% accuracy. For example, analyzing the targets of dynamic branch instructions is a hard problem, so distinguishing code and data very is challenging. Another issue is the hardening the disassembly intentionally. There are some techniques to do so. These methods are mostly used by malwares since; they are not wanted to be analyzed.

In addition to the disassembly operation the disassemblers analyze the program also. Using the disassembled code which is now in assembly language, they can generate control flow, call flow, data flow diagrams with pretty good accuracy.

[2] `http://linux.die.net/man/1/objdump`. [Last accessed on June 26, 2015].

2.6 Assembler

Assembly operation is the reverse of the disassembly. *Assemblers* are the programs that translate the assembly language code into executable machine code. It matches the corresponding instruction opcode using mnemonics in assembly language.

In static binary rewriting using assemblers the executable machine code of the code pieces to be inserted can be generated. In addition, in assemblers the origin address can be adjusted. Thus, while inserting the code piece to a specific address in the program, by using this property, the direct address usages can be handled according to where the code will be inserted.

CHAPTER 3

IMPLEMENTATION

In static binary rewriting framework, the code piece to be inserted should be executed, so, it should be also a part of the execution flow of the program. To achieve this, we propose inserting the code pieces into pre-existing functions in the binary or changing the control flow of the function so that the inserted code pieces executed during the execution of the function. We developed a static binary rewriting framework for Linux based x86 32 bit architectures. The developed framework is dependent to the architecture and operating system because rewriting is done on native binaries. First of all, the task requires disassembly and assembly operations which are dependent on the architecture by nature. Secondly, the executable file formats are operating system dependent. This affects the analysis and manipulation of the executable file. Thus, operations like creating a new section, retrieving information from file format are dependent on the operating system. Plus, apart from the operating system, the executable file format might be the architecture dependent as well.

It is worth to note that, although the implementation is dependent on the architecture and operating system, the main problems in static binary rewriting are generally independent from architecture and operating system. Thus, the solutions to the main problems and the methodology can be easily adapted to other platforms.

The implementation consists of following steps:

1. Finding function addresses to rewrite
2. Creating new executable section for injected code
3. Disassembly of the function for retrieving instructions and jump addresses

4. Assembly of the injected code

5. Rewriting of binary with injected code and relocated addresses.

3.1 Analyzing ELF Format

The ELF structure contains information about the binary file. In order to manipulate the binary file and do further analysis on the binary file we need to analyze and obtain some information from the ELF structure. For example, we need to know where the code and data is located in the file and where they will be loaded in the memory. Besides, file offset and virtual memory correspondence can be obtained by parsing the program header. Furthermore, the conversion between file offset and virtual memory or vice versa is used a lot in manipulating the ELF structure. For example, the debug information give the virtual address of the functions. Thus, finding the virtual address and file offset correspondence is used for finding the code of the function in the file.

In implementation we used the **elf.h** library[1]. The structures in the ELF format are defined in this library.

3.2 Finding Functions

In the proposed method we use debug information for finding function addresses and sizes. **libdwarf** [1] is used for parsing the debug information which is in DWARF format. DWARF has hierarchical tree like structure. Classes are children of namespaces. Class members are children of the classes. We use **libdwarf** for finding the function with the provided function name, class name (if it is a member function) and namespace name (if the class is under a namespace). There is an exception for overloaded and template functions. For these cases for the specific namespace and class the function name is not unique. Thus, for further clarification the parameter list or type information for templates can be provided. Another exception is for *inline functions*. The inlined functions cannot be found in the debug information or even if it is found, the execution does not flow from the function at runtime. Since it is in-

[1] `http://linux.die.net/man/5/elf`. [Last accessed on June 26, 2015].

lined, the copy of it is executed inside the caller function. For this case, the function to be rewritten has to be prevented from inlining by adding **no-inline** attribute to the function's definition.

3.3 Creating New Executable Section

In order to insert code pieces into the executable binary file, space for the extra code piece bytes is needed. Generally, there is no or a few byte long padding space between the functions in the binary file. These padding spaces can be **nop** instructions that does nothing or dummy instructions that do redundant tasks like moving a register's value into the same register. Besides, the proposed method should not depend on the assumption of pre-existing spaces in the binary. Thus, we propose creating a new executable section in the binary file.

In an executable generally there is a **LOAD** type segment for executable parts and, a **LOAD** type segment for the data parts. Table 3.1 shows an example program header output.

Table 3.1: Example Program Header

```
Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x0da1c 0x0da1c R E 0x1000
  LOAD           0x00da1c 0x08056a1c 0x08056a1c 0x00d08 0x5145c RW  0x1000
  DYNAMIC        0x00da30 0x08056a30 0x08056a30 0x000c8 0x000c8 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNUEHFRAME     0x00d880 0x08055880 0x08055880 0x0004c 0x0004c R   0x4
  GNUSTACK       0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
```

In order to the new executable section to be loaded properly, it has to be located in a **LOAD** type segment, because, executable section should be a loadable section in order to be loaded into the memory [10, Ch. 2]. Thus, in ELF format the header entries have to be modified properly. Creating new section requires three steps:

1. Opening space for the new section. The new section content should be located at the end of the file.

2. Creating a new section header entry for the new section. This entry specifies the file offset (the offset where the new section's content is located in the file), virtual address (the address the section to be loaded), alignment and read, write, execute right information of the section. Note that the new section header entry requires space too. Therefore, the section header should be moved to the end of the file and the section header also has to be located in a **LOAD** type segment.

3. Adjusting program header for locating the new section and the section header's new location in a **LOAD** type segment. In order to do this, we add a new program header entry which causes also moving the program header to the end of the file due to the space problem. Note that program header also has to be located in a **LOAD** type segment.

In executable binary files, optionally there can be a **PHDR** type program header entry which holds the information about the program header itself. For opening a new section we follow the following steps:

1. Move the program header to end of the file. If there is a **PHDR** type program header entry, change the type as **LOAD**. Otherwise add a new **LOAD** type entry. Then, modify the entry according to the new file and memory location of the program header in both cases.

2. Move the section header to end of the file. Add a new section header entry for the new section. Then, add a new **LOAD** type segment to hold the section header.

3. Add another **LOAD** type segment for the new section content. Then, write the content of the section to the file. Note that, section header entry and the program header entry information has to be consistent about the file position and memory address of the new section.

Table 3.2 shows an example output of **readelf**[2] before creating a new section and table 3.3 shows the output of **readelf** after creating a new section. The changes highlighted with bold. In ELF header the start offset of the program header table and section

[2] `http://linux.die.net/man/1/readelf`. [Last accessed on June 26, 2015].

Table 3.2: Before Creating New Section

```
1 ELF Header:
2   ... //omitted parts of ELF header information
3   Start of program headers:          52 (bytes into file)
4   Start of section headers:          111388 (bytes into file)
5   Flags:                             0x0
6   Size of this header:               52 (bytes)
7   Size of program headers:           32 (bytes)
8   Number of program headers:         8
9   Size of section headers:           40 (bytes)
10  Number of section headers:         39
11  Section header string table index: 36
12
13 Section Headers:
14  [Nr] Name              Type        Addr     Off    Size   ES Flg Lk Inf Al
15  [ 0]                   NULL        00000000 000000 000000 00      0   0  0
16  ... //omitted parts of the section header information
17  [38] .strtab           STRTAB      00000000 01d4c4 001285 00      0   0  1
18 Key to Flags:
19  W (write), A (alloc), X (execute), M (merge), S (strings)
20  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
21  O (extra OS processing required) o (OS specific), p (processor specific)
22
23 Program Headers:
24  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
25  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
26  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
27      [Requesting program interpreter: /lib/ld linux.so.2]
28  LOAD           0x000000 0x08048000 0x08048000 0x0da1c 0x0da1c R E 0x1000
29  LOAD           0x00da1c 0x08056a1c 0x08056a1c 0x00d08 0x5145c RW  0x1000
30  DYNAMIC        0x00da30 0x08056a30 0x08056a30 0x000c8 0x000c8 RW  0x4
31  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
32  GNUEHFRAME     0x00d880 0x08055880 0x08055880 0x0004c 0x0004c R   0x4
33  GNUSTACK       0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
34  //omited: rest of the output
```

header table is changed to the new offsets of them. Plus, the number of section headers and program headers are increased in ELF header. In section header table the new section is added with no name. In program header table two new program headers added (first one - line 35 - holds the section header table and the second one - line 36 - holds the new section content). In addition, the fields of the first program header (line 26) is changed according to the new position of the program header table **LOAD** and the type is changed to **LOAD** from **PHDR**.

3.4 Disassembly

Our method needs to insert machine codes for repairing relocated address references in functions. For inserting new machine codes, the function should be analyzed first.

Table 3.3: After Creating New Section

```
ELF Header:
  ... //omitted parts of ELF header information
   Start of program headers:          124745 (bytes into file)
   Start of section headers:          190281 (bytes into file)
  Flags:                              0x0
  Size of this header:                52 (bytes)
  Size of program headers:            32 (bytes)
  Number of program headers:          10
  Size of section headers:            40 (bytes)
  Number of section headers:          40
  Section header string table index: 36

Section Headers:
  [Nr] Name              Type        Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL        00000000 000000 000000 00      0   0  0
  ... //omitted parts of the section header information
  [38] .strtab           STRTAB      00000000 01d4c4 001285 00      0   0  1
  [39]                   PROGBITS    08037749 02f749 010000 00  WX  0   0  4
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x01e749 0x08035749 0x08035749 0x00500 0x00500 RWE 0x1000
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x0da1c 0x0da1c R E 0x1000
  LOAD           0x00da1c 0x08056a1c 0x08056a1c 0x00d08 0x5145c RW  0x1000
  DYNAMIC        0x00da30 0x08056a30 0x08056a30 0x000c8 0x000c8 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNUEHFRAME     0x00d880 0x08055880 0x08055880 0x0004c 0x0004c R   0x4
  GNUSTACK       0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  LOAD           0x02e749 0x08036749 0x08036749 0x01000 0x01000 RWE 0x1000
  LOAD           0x02f749 0x08037749 0x08037749 0x10000 0x10000 RWE 0x1000
  ... //omited: rest of the output
```

Thus, the function should be disassembled to extract information on each instruction and its operands precisely. The first purpose of disassembly is checking the place to insert code. Inserting bytes into an instruction would corrupt the function and can cause non-deterministic behaviour in function. If the offset is not proper, an exception is thrown. Furthermore, inserting new machine codes generally has aim of adding new functionality or changing the existing one. Thus, the insertion should be made to proper places. The second and the most important purpose is to locate the branch instructions in the function. During the binary rewriting the targets of the static and dynamic branch instructions will be corrupted. In order to fix this issue we need to locate them in the function.

Note that, disassembly with 100% accuracy is a really challenging task and still open problem. However, the problems that prevent the 100% disassembly can be eliminated in our case. First of all, we make use of the debug information which provides an advantage. Since we disassemble each function individually, the possible data or padding bytes between the functions, that cause fallacies in disassembly, do not cause problems for us. Secondly, our scope excludes the self-modifying, hand crafted and obfuscated binary files. Hence, these challenging cases are eliminated for our situation.

We use an open source disassembly library named **BeaEngine**[3]. **BeaEngine** supports x86 32 bit, x64 and some more architectures. It takes the code to disassemble and size as input. The third input parameter is the virtual address that the code is located. With this information it can calculate the virtual addresses of the instructions and virtual addresses of the targets of relative branch instructions. Using this library we disassemble functions with linear approach. It is worth to note that the compiler might append the jump table of a **switch** statement at the end of the function. In this case this jump table can be detected with further analysis [7]. In our tests we didn't come up with such a case.

3.5 Assembly

For inserting a new functionality or changing the existing one, new codes should be inserted. Since, these codes should be a machine language codes, we need a conversion to machine language. At this step, the new codes to be inserted should be written in assembly language because of the ease of conversion to machine language and ease of manipulation. Here, manipulation refers to on the fly address calculation and change. In assembly language, unlike high level programming languages, the branches, data accesses and function calls can be easily handled using relative addresses. Besides, the load address of the code can be adjusted easily so that the code fits to the address that it will be inserted.

We use **Netwide Assembler (NASM)**[4] to assemble the codes in the implementation.

[3] `http://www.beaengine.org/home`. [Last accessed on June 26, 2015].
[4] `http://www.nasm.us/`. [Last accessed on June 26, 2015].

NASM is an assembly tool which can give output in different formats. However, since we insert the output into another binary file, we get the output as bare machine language bytes.

3.6 Rewriting

The rewriting section contains the algorithm logic. The algorithm runs the following main steps for inserting one code piece, into one function, into one executable binary file:

1. Get the input. Input consists of:
    - the executable binary file and debug information if it is separated from binary file
    - the name of the function, and the namespace and class names if they are required for clarifying the function. Plus, for the overloaded functions the parameter types can be given.
    - the code to be inserted written in assembly language
    - the insertion offset relative to the beginning of the function
2. Retrieve the function beginning address and the size of the function from debug information. In debug information, the function beginning address is held as the virtual address which is the address that the function will be loaded. This virtual address is converted to file offset which gives the location of the function in the file. This conversion is done with the help of analyzing program headers in the ELF format. As we mentioned earlier, program header holds the file offset and memory address correspondence information.
3. Read the machine language codes of the function from the file to the memory.
4. Disassemble the function. Virtual address of the beginning of the function is also given to **BeaEngine** so that the instruction and branch target addresses are also computed.

5. Find all static branch instructions and store the target addresses of them. Note that these instructions' target addresses are calculated relative to their virtual addresses. Thus, after any change in these instructions' position, the target addresses should be fixed. The target address information will be used for fixing the target addresses.

6. Add a new executable section in the binary file.

7. Move the function machine language codes of the function to the new executable section in the file.

8. Create a map, named *relocation map*, of each instruction according to their memory addresses. In this map hold the old and the corresponding new virtual address of the each instruction in the function. This map will be used for bookkeeping of the instructions' new addresses.

9. Assemble and obtain the machine language codes of the code to be inserted that is written in assembly language

10. Insert the machine language codes to the given offset that is relative to the beginning of the function. Note that, this insertion corrupts the relative branches because some bytes are inserted between the instructions. Thus, the precomputed relative addresses are wrong now.

11. Handle incoming function calls. The details of this step are explained in subsection 3.6.1

12. Handle dynamic jumps in the functions and calls to the function. Note that, the function is moved to a different address. Thus, now all function calls to the function and dynamic branches will flow to a dangling address. The details of this step are explained in the subsection 3.6.2.

13. Update the relocation map according to the new addresses of the instructions.

14. Fix the static relative jumps and calls according to the last state of the relocation map. Change the operands of the static relative jumps and calls so that the targets remain unchanged. In x86 the *short jumps* have one byte operand. Hence, the short jumps can address from -128 to +127 byte relative range. Note

that, after rewriting due to the new inserted bytes, the targets of the short jumps can go beyond this range. In this case, we should change short jumps to *near* jumps which have four byte operand. Therefore, the size of the static jump instructions can change. This means byte shift in the function too. Thus, before fixing the static jumps and calls the possible size changes should be computed first.

Figure 3.1, 3.2 and 3.3 shows an example for the steps of the rewriting of a function without a dynamic jump instruction.

Figure 3.1.a shows the assembly instructions of the function obtained by disassembly and 3.2.b shows the assembly instructions of the same function after it is moved to the new address.

Figure 3.2.a shows the initial relocation map which holds the original addresses of the instructions in first column and the new address of the instruction in the second column. Figure 3.2.b shows the assembly instructions after new code pieces inserted. For simplicity only **nop** instructions are inserted in example.

Figure 3.3.a shows the updated relocation map (updated entries are written in red). Figure 3.3.b shows the assembly instructions after the static jump instructions are fixed.

3.6.1 Handling Incoming Function Calls

Moving the function to another address causes the function calls to the function to become dangling. As a solution, detecting the incoming function calls and updating them with the new address of the function can be proposed. However, these function calls can be dynamic, in which the target is determined at runtime. In such cases detecting the incoming function calls is a challenging task. For this reason, incoming function calls are handled with a different approach. Instead of detecting incoming function calls and updating them, in the proposed method the function calls are redirected to the new address of the function. This is accomplished by inserting an unconditional jump instruction, whose target is the new address of the function, to the

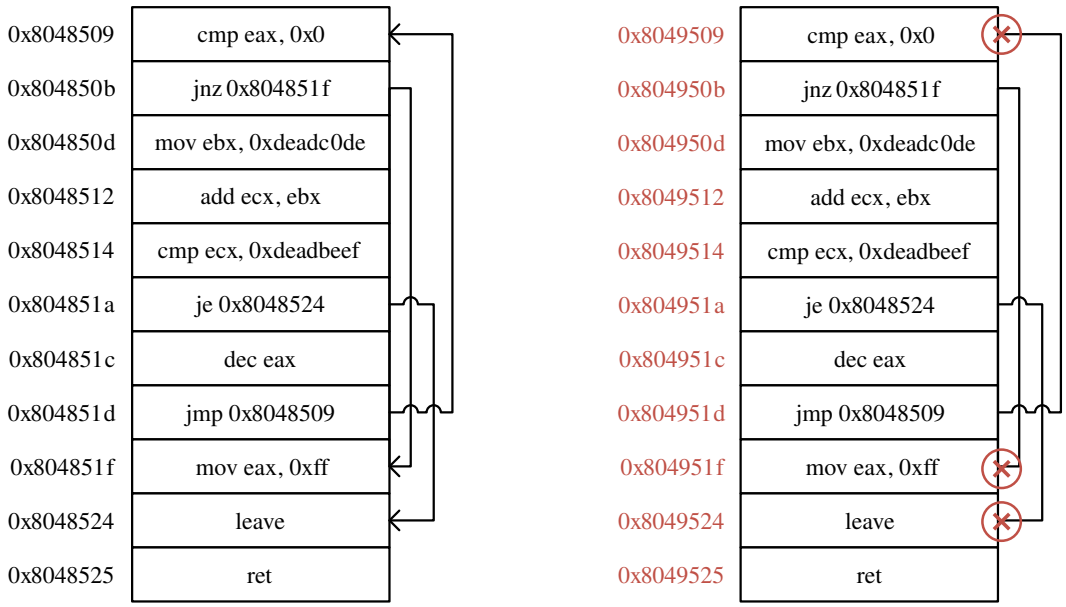


Figure 3.1: Rewriting Example: (a) Step 4 - Disassembled Function, (b) Step 7 - Function At New Address (old address + 0x1000)

original beginning address of the function [14]. Hence, whenever the moved function is called, no matter the call instruction is dynamic or static; first the execution will reach to the original address of the function. Then, the unconditional jump instruction will redirect the execution to the new address of the function. Note that, inserting the additional unconditional jump to the original beginning address of the function does not affect the stack frame. Therefore, the return from the function is not affected.

Figure 3.4 shows the inserted unconditional jump instruction at the beginning of the example function in the figure 3.1. Note that, the code of the function is moved to another address. So the space at the function's original position can be overwritten.

3.6.2 Handling Dynamic Jumps

One of the biggest challenges about static binary rewriting is handling the dynamic jumps. The targets of the dynamic jumps are determined at runtime according to state of the program. Hence, after moving the function to another address, the targets of the dynamic jumps point to dangling addresses. Unlike static jumps, the target addresses cannot be updated easily. Because, determining the target of the dynamic

0x8048509	0x8049509
0x804850b	0x804950b
0x804850d	0x804950d
0x8048512	0x8049512
0x8048514	0x8049514
0x804851a	0x804951a
0x804851c	0x804951c
0x804851d	0x804951d
0x804851f	0x804951f
0x8048524	0x8049524
0x8048525	0x8049525

(a)

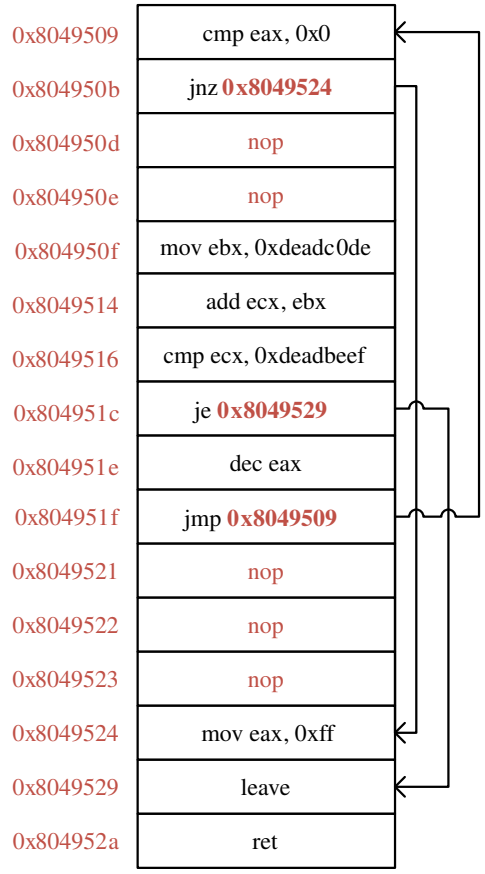
0x8049509	cmp eax, 0x0
0x804950b	jnz 0x804851f
0x804950d	**nop**
0x804950e	**nop**
0x804950f	mov ebx, 0xdeadc0de
0x8049514	add ecx, ebx
0x8049516	cmp ecx, 0xdeadbeef
0x804951c	je 0x8048524
0x804951e	dec eax
0x804951f	jmp 0x8048509
0x8049521	**nop**
0x8049522	**nop**
0x8049523	**nop**
0x8049524	mov eax, 0xff
0x8049529	leave
0x804952a	ret

(b)

Figure 3.2: Rewriting Example: (a) Step 8 - Initial Relocation Map, (b) Step 10 - New Code Pieces Inserted

jumps requires analysis of the code. In fact, in most cases the target can have different values at runtime dependent on the input, where the most typical example is the **switch** statements. Thus, even a successful analysis on the possible target values of the specific dynamic jump does not solve the problem. Instead of analyzing the target and statically redirecting it to the new address of instruction, the target should be altered at runtime so that the target becomes correct for all cases. To achieve this, instead of determining which dynamic jump instruction is targeting to which addresses, we propose first to find the addresses, which may be a possible dynamic jump target, without determining the correspondence between the dynamic jump instructions and target addresses. The found possible target addresses are actually a *superset of the possible dynamic jump targets*. The obtained set may include some addresses which are not a target address but, it must include all actual target addresses.

0x8048509	0x8049509
0x804850b	0x804950b
0x804850d	0x804950f
0x8048512	0x8049514
0x8048514	0x8049516
0x804851a	0x804951c
0x804851c	0x804951e
0x804851d	0x804951f
0x804851f	0x8049524
0x8048524	0x8049529
0x8048525	0x804952a



(a) (b)

Figure 3.3: Rewriting Example: (a) Step 13 - Updated Relocation Map, (b) Step 14 - Static Jumps Fixed

3.6.2.1 Finding Possible Dynamic Jump Targets

The dynamic jump target is obtained from data at run time. Although the target address may change according to the input or values (i.e. register values, data fields at memory etc.), the target addresses should be present in the binary file. The target addresses of the dynamic jump instructions are determined based on a lookup table like the **switch** case. Therefore, in spite of the difficulty of determining possible targets of a specific dynamic jump with perfect accuracy [31], it is possible to find a superset of all possible jump targets without analyzing the execution of the program.

In order to find the superset of all possible jump targets, we interpreted the all non-executable sections as data. Note that, we do not know the structure of these sections; hence, we do not know how to interpret the values. What we did is that, we took the

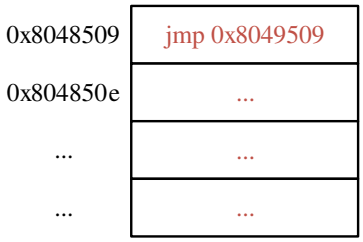


Figure 3.4: Rewriting Example: Step 11 - Redirecting Incoming Function Calls

section as a block and treated each byte as if it is a beginning of an address value. Plus, in order to cover possible data bytes between the function's codes and the immediate values in the function's code, we did the same treatment to the function code (from function begin address to end address as a block). In the end, all address values are added to the superset of the all possible jump targets. This approach includes a lot of address that is not a jump target. On the other hand, the advantage is that it includes all real dynamic jump targets. This approach for finding the possible jump targets was successful in the tests (i.e. we did not come up with a case that a dynamic jump target is not handled). Nevertheless, if it is needed, the search area for the possible jump targets can be extended.

As we mentioned, the computed superset of all possible jump target includes redundant address values. However, the redundant address values in the superset can be pruned [14]. Note that, the possible jump target must point beginning of one of the instructions in the function. Otherwise, the jump target would be in the middle of an instruction which is not a valid case if the binary file is not hand crafted or obfuscated. The other possibility is that if the target is not inside the function, the dynamic jump target may be actually beginning of a function. This case does not cause a problem even if the target function is also relocated. Since, as it is mentioned in the section 3.6.1, the incoming function calls to the relocated target function are handled by inserting a jump instruction at the beginning of the function, if the possible jump target points an address outside the function, it can be pruned. At the end, the superset includes only the addresses which are actually the begin address of an instruction inside the function.

3.6.2.2 Redirection Map For Dynamic Jump Targets

Since dynamic jump targets are determined at runtime according to the state of the program, for preventing dangling target addresses, after relocating the function, the dynamic jump targets should be redirected according to the new position of the function. This redirection cannot be done statically because, each execution of one dynamic jump instruction may lead to different target addresses.

For dynamic jump target translation at runtime, it is necessary to store the new addresses of the instructions of the function. In other words, we need a map for querying the new position of the dynamic jump targets. Therefore, the map should hold the old and the new addresses of the instructions of the function. Since this map will be accessed at runtime, it has to be stored in the binary file. Moreover, finding the new address of the target should be fast (preferably with O(1) time complexity) for reducing the overhead.

A straightforward solution is holding the map like an array. Array's zeroth index corresponds to the new address of the first instruction of the function and the following indexes correspond to the new addresses of following instructions of the function. Hence, the array holds the new address of each instruction in the indexes. However, note that in x86 the instruction sizes are not fixed so that without a secondary map for querying the beginning addresses of the instructions, we cannot find the new address of the specific instruction with O(1) time complexity. A solution can be holding an entry for each byte of the function in the array. Although, the array would be redundantly long, with this approach without secondary map for querying the beginning of the instructions, we can find the new address of each byte of the function with O(1) time complexity.

If the possible dynamic jump targets could not be determined, we would need to put an entry to the map for every byte in the function. However, by analyzing the possible jump targets, the addresses that cannot be dynamic jump target are eliminated. As a result, number of entries that should be held in the map reduces. Actually, after pruning the entries by finding the superset of possible jump targets, we can use a different approach for storing new addresses of the possible target instructions.

Since the new addresses of the instructions is only required for translating the dynamic jump targets, we only need to find the new addresses of the instructions which are included in the superset of possible jump targets. Besides, note that the code of the function is moved to another address. Therefore, the space at the function's original position is available and we can safely write the new addresses of the instructions to the possible jump targets. On each possible jump target address which is actually beginning of an instruction, we store the new address of the instruction with a one byte marker at the beginning of the address [31]. The purpose of the marker is to indicate that the following four bytes is an address value which is the new address of the instruction. This address value is used to redirect the dynamic jump targets to the correct addresses. The reasons why we use marker byte will be explained in the section 3.6.2.3. Figure 3.5.a shows an example to a function with a dynamic jump. Figure 3.5.b with assuming the found possible jump targets are 0x804850d, 0x8048513 and 0x8048519, shows how the redirection entries stored in that function after the original function is moved to another address. Suppose the numbers next to the 0xF4 byte are the new addresses of the instructions, after they moved to another address in the new executable section.

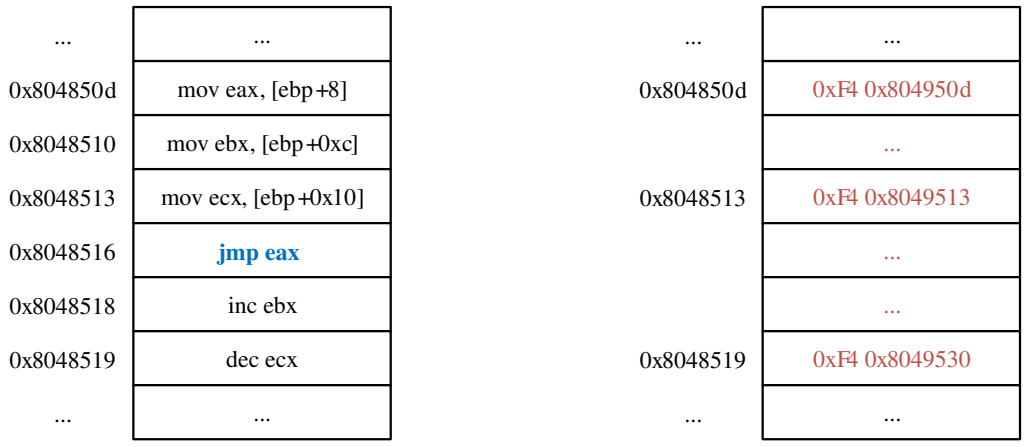


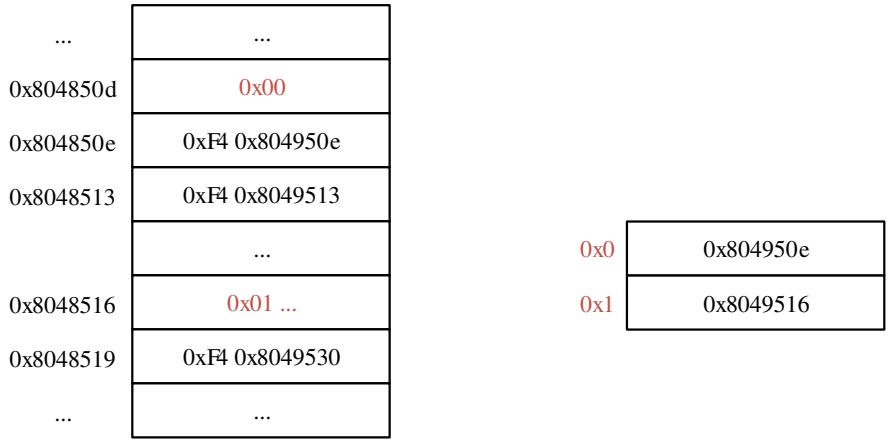
Figure 3.5: Rewriting Example: (a) Function With A Dynamic Jump, (b) Redirection Entries For Possible Jump Targets 0x804850d, 0x8048513 and 0x8048519

0xF4 is used as marker byte. In x86 it is the opcode of the **halt** instruction which is not a valid instruction [31]. Note that, one byte marker and four bytes address value is written to the possible target instruction. Therefore it requires five bytes

space to store the new addresses of the instructions. Nevertheless, the problem is that there can be multiple possible target in five bytes range, because, in x86 there are instructions shorter than five bytes. Furthermore, the space for storing the new addresses is restricted with the size of the function. Hence, possible jump targets that are in the last four bytes of the function cause the same problem. Plus, in order to handle the incoming function calls to the relocated function, as we mentioned earlier, a jump instruction, whose length is also five bytes, is written at the beginning of the function. Therefore, if there is a possible jump target in the first five bytes of the function, we cannot overwrite five byte on the jump instruction.

In order to handle such cases where we cannot write five bytes due to space restriction, we write one byte array index indicator to the possible jump target address. Then, the new address is stored in an array at the corresponding index. Then, the array is stored in the binary for accessing at runtime. Except the case that there is a possible jump target in the first five bytes of the function, the array index can be assigned incrementally from 0 to 0xFF and skipping 0xF4. If there is a possible jump target in the first five bytes of the function, we use the byte that is already written in the jump instruction as array index indicator. At the end, for one function we can store up to 255 new addresses for the possible jump targets. During the tests we didn't come up with such a case that 255 index bytes weren't enough. However, if such a case occurs, one of the previously mentioned approaches can be used such as storing the new address of each byte of the function in an array. Figure 3.6.a with assuming the found possible jump targets are 0x804850d, 0x804850e, 0x80485013, 0x8048516 and 0x8048519, shows the redirection entries stored in the function 3.5.a. Figure 3.6.b shows the array holding the extra redirection entries. Since we cannot insert five bytes between 0x804850d and 0x804850e, we used an array index indicator at 0x804850d. Again, there is not five bytes space between 0x8048516 and 0x8048519, so we used another array index indicator at 0x8048516.

One remaining exception about storing the new addresses of possible jump targets occurs when the jump instruction written at the beginning of the function has a 0xF4 byte in it and there is a possible jump target corresponding that byte. In this case we cannot use that byte as array index indicator because it is the marker byte. By moving the function to a different address, the problem can be solved. Since the operand bytes



(a) (b)

Figure 3.6: Rewriting Example: (a) Redirection Entries For Possible Jump Targets 0x804850d, 0x804850e, 0x80485013, 0x8048516, 0x8048519, (b) Redirection Entries In Extra Array

of the jump instruction will change according to its target address, if the new address of the function is changed properly, it will change the bytes in the instruction.

3.6.2.3 Dynamic Jump Target Translation At Runtime

The target address of the dynamic jumps should be updated at runtime. According to the original target address, it has to be translated to its corresponding new address with the help of the map explained in the section 3.6.2.2. In order to accomplish this task, we need to insert a code piece just before the dynamic jump instruction. The code piece should look the original target address, retrieve the new position of the target and update the target of dynamic jump accordingly.

We categorize the dynamic jumps into two categories according to their operands' format. In one category it is simpler to do the translation. In this category the operand of the dynamic jump is just a register like **eax**, **ebx**, etc. The target of the dynamic jump is directly the value in the operand register. Therefore, in order to handle this case, we just need to update the value in the register. On the other hand, the other category uses memory reference. In this category, the target of the dynamic jump is stored in a memory location and the operand shows the address of the memory location. Moreover, the address of memory location might be resolved with a computation in the

operand. Table 3.4 shows example instructions from these categories. For the second category, we split the instruction. First, we insert an instruction which does the computation - if there is any - and obtain the actual target address from the memory location that holds the actual target address. Next, we insert a code piece which updates the target address according to our map. Finally, we replace the dynamic jump with a dynamic jump instruction from the first category.

Table 3.4: Example Dynamic Jump With and Without Computation

Dynamic jump without computation	Dynamic jump with computation
`1 jmp eax`	`1 jmp [0xdeadc0de+eax 4]`

The code piece that does the translation differs depending on whether we need an array to store the new addresses or not. If there is no need for the array, which means for all possible jump targets there are five bytes space, the translation code piece is simpler. It just has to check if the first byte of the target address is 0xF4 or not. Table 3.5 shows the code pieces for this case. If the extra array is used for storing some of the new addresses of the possible jump targets, the code piece has to look into array to retrieve the new address from the array. Table 3.6 shows the code pieces for this case.

The marker byte is used to discriminate whether the new address is in the extra array or not. If 0xF4 was written at the first byte of the target address, this indicates that the following four byte is the new address of the target. On the other hand, other bytes mean that the new address of the target is stored in the array. For the case that extra array is not needed, another purpose of the marker byte is to indicate if the dynamic jump target leads into the function that is actually relocated or it leads to another function. Since, in x86 no valid instruction can start with 0xF4 byte, if there is a 0xF4 byte at the beginning of an instruction, we can be sure that the byte was written by us. Hence, we can be sure the target address actually leads into the function that was relocated. For the case that extra array is needed, in order to be sure the target address actually leads into the function, we check if the target address is between the function's original begin and end address.

One important issue is that saving the values of the used registers in these code pieces.

Table 3.5: Translation Examples (Without Extra Array)

Translation of **jmp eax**	Translation of **jmp [0xdeadc0de+eax*4]**

`cmp byte [eax], 0xF4`
`cmovz eax , [eax+1]`
`jmp eax`
	`sub esp, 4`
	`push eax`
	`mov eax, [0xdeadc0de+eax 4]`
	`cmp byte [eax], 0xF4`
	`cmovz eax , [eax+1]`
	`mov [esp+4], eax`
	`pop eax`
	`ret`

Since the code pieces inserted after compilation, if the value of any register is changed due to the code piece inserted by us, it will cause change in the logic of the program. Moreover, the stack frame has to remain unchanged. Most of the operations in the binary is dependent on the stack frame like function calls, function returns, local variables etc. In order to preserve values of the used register by the code piece and the values in the stack frame we used some extra instructions.

In the second columns of the table 3.5 and 3.6, the first instruction is opening space in the stack frame to store the new address of the jump target. The found new address of the jump target is stored in that space by the instruction **mov [esp+4], eax**. The second instruction pushes the value of the **eax** register so, after our task is done, the old value is restored by **pop eax** instruction. Note that the last instructions are **ret** however, it is not used for returning from a function. As we mentioned before, the **ret** instruction pops an address from top of the stack into the instruction pointer. We used this fact, for redirecting the execution to the new address of the jump target and restoring the stack frame which is modified at the first instruction by opening space.

3.6.3 Handling PIC

Position independent code (PIC) is the code which executes independent from where it is loaded into memory [21]. PIC is not tied to a fixed address. Since it includes no absolute address, it can execute correctly wherever it is loaded to memory. However, it uses relative addresses in order to access to memory. PIC retrieves the current value of the instruction pointer and computes the address to be accessed relative to

Table 3.6: Translation Examples (With Extra Array)

Translation of **jmp eax**

```
1   cmp byte [eax], 0xF4
2   je notExtra
3   cmp dword eax,  funcAddr
4   jl out
5   cmp dword eax,  funcEndAddr
6   jge out
7   mov byte al, [eax]
8   and eax, 0xFF
9   mov eax, [arrayAddr + eax 4]
10  jmp out
11 notExtra:
12  mov eax, [eax+1]
13 out:
14  jmp eax
```

Translation of **jmp [0xdeadc0de+eax*4]**

```
1   sub esb, 4
2   push eax
3   mov eax, [0xdeadc0de+eax 4]
4   cmp byte [eax], 0xF4
5   je notExtra
6   cmp dword eax, funcAddr
7   jl out
8   cmp dword eax, funcEndAddr
9   jge out
10  mov byte al, [eax]
11  and eax, 0xFF
12  mov eax, [arrayAddr + eax 4]
13  jmp out
14 notExtra:
15  mov eax, [eax+1]
16 out:
17  mov [esp+4], eax
18  pop eax
19  ret
```

the instruction pointer. Therefore, changing the position of the PIC in the binary will make the relative addresses be computed wrong.

In x86 direct accesses to instruction pointer is not allowed. Therefore, value of instruction pointer is retrieved with the help of **call** instruction. As we mentioned earlier, the **call** instruction implicitly pushes the address of the instruction following the **call** instruction to the stack. In other words, it pushes the return address to the stack. With the help of this fact, in PIC the return address is retrieved from stack by the **pop** instruction or by reading the return value from stack. After relocating the function the corrupted PICs are fixed by searching this PIC pattern in the function code and updating the register value that holds the instruction pointer value according to the new position of the function code. Table 3.7 shows an example to PIC and with assuming the address of the **add ebx, 0x1234** instruction is 0x8048ABC, how it is modified during rewriting.

Table 3.7: Example PIC

Example Original PIC

```
1 getEIPValue:
2   mov ebx, [esp]
3   ret
4   ...
5   call getEIPValue
6   add ebx, 0x1234
7   call [ebx+0x1c]
8   ...
```

Example Rewritten PIC

```
1 getEIPValue:
2   mov ebx, [esp]
3   ret
4   ...
5   call getEIPValue
6   mov ebx, 0x8048ABC
7   add ebx, 0x1234
8   call [ebx+0x1c]
9   ...
```

CHAPTER 4

TESTS AND RESULTS

We implemented a standalone function based static binary rewriting framework for executable binary files, Linux based operating systems and x86 32 bit architectures. It takes an executable file, code insertion description including the function and the offset relative to the beginning of the function to which the code piece will be inserted. The framework gives the new executable binary file as output, after the code insertion process is complete.

The tests were run on virtual machine with an Ubuntu 14.04 guest operating system. The virtual machine has a single core 2.83 GHz processor and 2 GB of RAM.

We tested our implementation against nine applications. Seven of them are written in C programming language and two of them are written in C++. The names of the applications and the small description of them are listed below:

base64: it does base64 encoding/decoding of the given input[1]

md5sum: it computes and verifies the md5 hash of the given input[2]

sha1sum: it computes and verifies the sha1 hash of the given input[3]

tsort: it performs topological sort on the given input[4]

gzip: it compresses the given file using Lempel-Ziv coding[5]

[1] `http://linux.die.net/man/1/base64`. [Last accessed on June 26, 2015].
[2] `http://linux.die.net/man/1/md5sum`. [Last accessed on June 26, 2015].
[3] `http://linux.die.net/man/1/sha1sum`. [Last accessed on June 26, 2015].
[4] `http://linux.die.net/man/1/tsort`. [Last accessed on June 26, 2015].
[5] `http://linux.die.net/man/1/gzip`. [Last accessed on June 26, 2015].

diff: it compares the given files line by line[6]

readelf: it parses and displays the information in the given ELF files[7]

cppcheck: it is a static analysis tool for C/C++ code[8]

doxygen: it is a tool for generating documentation from annotated C++ source codes[9]

Those applications are all open source. They are compiled with gcc (Gnu Compiler Collection)[10] with **-pg -g** flags[11]. The **-pg** flag is used for enabling profiling and **-g** flag is to include the debug information to the file. These executable binary files are our control binaries.

After compilation, the applications are run with their inputs and profiled with **gprof**[12] [16]. **gprof** outputs the list of all executed functions that takes measurable time (some functions takes very small execution time so that the profiler cannot measure). The functions that are in the list, are the functions that will be processed by our framework in our tests. Besides, for the functions in the list **gprof** outputs how many times that each function is executed. Plus, the total execution time of each function in the list and the total execution time of the program are available in the output of the **gprof**.

We inserted four bytes of **nop** (each **nop** is one byte) instructions to the beginning of all the functions in the list using our framework. The resulting executable binary files are our experimental binaries. Table 4.1 shows how many function was rewritten by our framework and total numbers of the static and dynamic jump instructions in the rewritten functions.

In our tests, we ran both control binaries and experimental binaries with the same inputs which are actually the inputs we used to obtain the function list. Both control and experimental binaries were run 31 times. The first runs were ignored in order to eliminate the effect of hard disk cache. The first run usually takes dramatically longer

[6] `http://linux.die.net/man/1/diff`. [Last accessed on June 26, 2015].
[7] `http://linux.die.net/man/1/readelf`. [Last accessed on June 26, 2015].
[8] `http://linux.die.net/man/1/cppcheck`. [Last accessed on June 26, 2015].
[9] `http://www.doxygen.org/`. [Last accessed on June 26, 2015].
[10] `https://gcc.gnu.org/`. [Last accessed on June 26, 2015].
[11] `https://gcc.gnu.org/onlinedocs/gcc/Debugging-Options.html`. [Last accessed on June 26, 2015].
[12] `http://linux.die.net/man/1/gprof`.. [Last accessed on June 26, 2015].

Table 4.1: Numbers About The Tested Binaries

	# of rewritten functions	# of static jumps	# of dynamic jumps
base64	4	54	0
md5sum	10	48	0
sha1sum	10	51	0
tsort	15	149	0
gzip	43	480	0
diff	40	729	1
readelf	38	2439	8
cppcheck	609	28620	7
doxygen	1695	1982	103

execution time. The following runs take less execution time because the input file is cached.

For all applications, the execution times of 30 runs for both control and experimental binaries are measured with the help of **gprof**. Then the average, minimum, maximum and standard deviation of the execution times were calculated. Table 4.2 shows the results for each application (o stands for original executable file, r stands for the rewritten executable file). Plus, the figure 4.1 shows the graph of the results.

In order to validate that the experimental binaries provide same functionality correctly with the control binaries, for each application the hash of the output of the each run was computed by **md5sum**[13] shell tool. Then, each hash were checked if it same for control binary and experimental binary. In the tests the output hash checks were successful for all binaries for all runs.

In order to observe the effect of handling dynamic jumps and scaling of our method over the size of data, we did an additional test. In **doxygen** application we picked two different functions. One of the functions, which is named **HtmlGenerator::codify**, includes one dynamic jump instruction. The other function, which is named **QGDict::look_ascii**, does not include dynamic jump instruction. In one case we inserted four bytes of **nop** instructions only to the beginning of the function with a dynamic jump. In the other case we did the same operation to only the function without dynamic jump.

[13] `http://linux.die.net/man/1/md5sum`. [Last accessed on June 26, 2015].

Table 4.2: Execution times in seconds; o: original, r: rewritten

	average		minimum		maximum		stdev	
	o	r	o	r	o	r	o	r
base64	2.76	2.66	2.51	2.38	3.04	3.17	0.15	0.21
md5sum	2.09	2.08	1.73	1.50	2.67	2.59	0.25	0.29
sha1sum	7.01	6.80	6.68	6.57	7.38	7.03	0.19	0.15
tsort	8.55	7.46	5.28	4.74	12.70	11.15	2.40	1.94
gzip	42.43	44.28	39.64	43.44	44.01	45.66	1.06	0.58
diff	21.91	22.45	19.78	20.66	24.10	25.61	1.23	1.38
readelf	3.81	7.67	3.13	6.83	4.20	8.24	0.44	0.4
cppcheck	71.57	78.88	68.87	76.77	72.75	81.00	0.99	1.15
doxygen	3.52	4.10	3.06	3.63	3.96	4.60	0.21	0.27

In both cases we used eight inputs with the size range of 200mb to 1.6 GB with increase amount of 200mb. In both cases, for each input the application were run 21 times with ignoring the first one. Then, for each input the average execution time of the altered function was measured for both cases. Moreover, as in the previous test the output hash check was done and the checks were successful. Figure 4.2 shows the graph of the execution times of the function with a dynamic jump instruction. Figure 4.3 shows the graph of the execution times of the function without dynamic jump instruction.

As the graphs indicate the main cause of the overhead comes from handling the dynamic jump instructions. If there is no dynamic jump instruction in the function, the execution time of the original and the altered function are approximately same. The overhead of the handling dynamic jump instructions grows approximatively linear with the increasing input size.

As it can be seen from the table 4.2 and graph in the figure 4.1, the performance overhead in the **readelf** is much more higher than the performance overhead in the other applications. The reason for this high performance overhead in the **readelf** application is caused by the handling the dynamic jump instructions. In **readelf** application there is a function with a dynamic jump instruction whose name is **byte_get_little_endian**. This function takes 79.3% of the total execution time in the

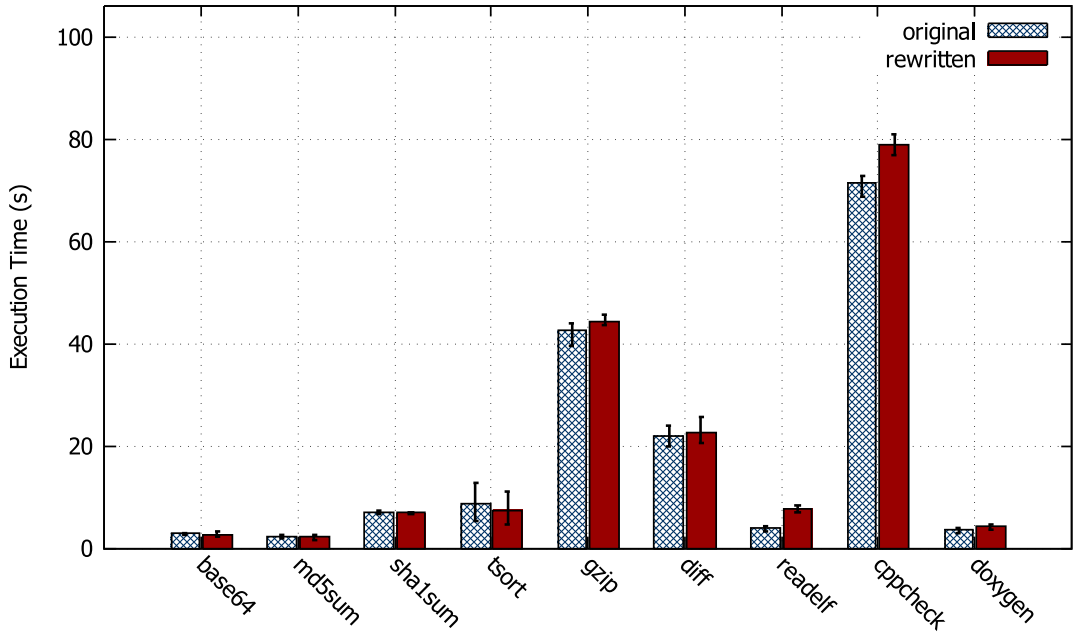


Figure 4.1: Tested Binaries and Execution Times

original executable file. Therefore, considering this fact and the fact that handling dynamic jumps is the main cause of performance overhead (as the figure 4.2 and 4.3 indicates), shows us the reason of this high performance overhead in **readelf**.

Note that, after inserting the **nop** instructions, the function's position is changed. This causes the failure of the profiling by **gprof**, because, it depends on the symbol table entries in the binary file which becomes wrong after relocation of the function. For the tests, after the code insertion process, we fixed the corresponding symbol table entries in the binary. Nevertheless, apart from this special case, after the rewriting of the binary the debug information become wrong due to the relocation of the functions.

4.1 Case Study

Our main motivation behind static binary rewriting study is applying software protection methods without source code modification. For the sake of showing this use case we applied a known obfuscation method to an executable binary file without making any change in source code using our binary rewriting framework.

As we mentioned earlier, disassemblers can generate the control flow graph of a pro-

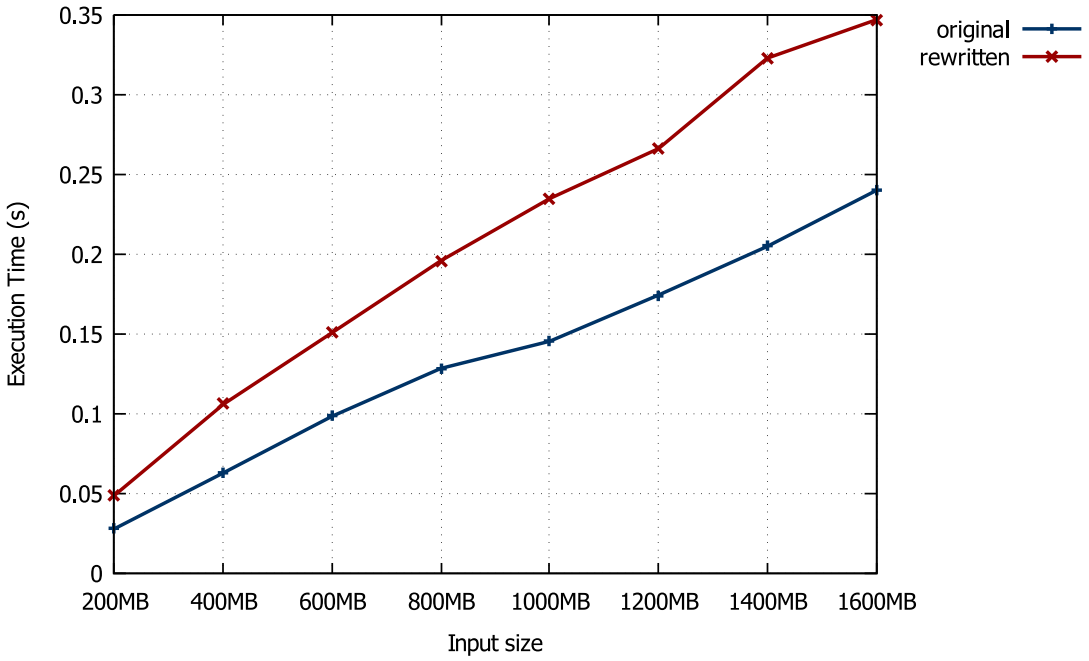


Figure 4.2: Execution Time of A Function With A Dynamic Jump (for a Doxygen function)

gram with a pretty good accuracy. This information is important for a reverse engineer, because, the control flow graph reveals how the program executes. With further analysis, based on the control flow graph a reverse engineer can obtain the if-else and loop mechanisms in a program. Therefore, this kind of information can be a good hint for learning the secret algorithm in the program.

For hardening the control flow graph analysis we implemented a method named as *jump through branch functions* [24, Ch. 4.3.5]. This method replaces static jump instructions with a function call. Note that the **ret** instruction pops the return address from the stack and execution continues from the popped address. Using this fact, in the stack the return point of those functions are changed with the target address of the corresponding static jump instruction. Therefore, instead of returning to the instruction next to the **call** instruction, the functions return to the target addresses of the jump instructions. This method relies on the fact that the disassemblers assume that the **call** instruction returns to the following instruction of **call** instruction. Furthermore, targets of the static jump instructions can be determined intuitively (the operand is the relative distance to the target or the absolute address). When these static jump instructions are replaced with function calls, it becomes harder to determine the targets.

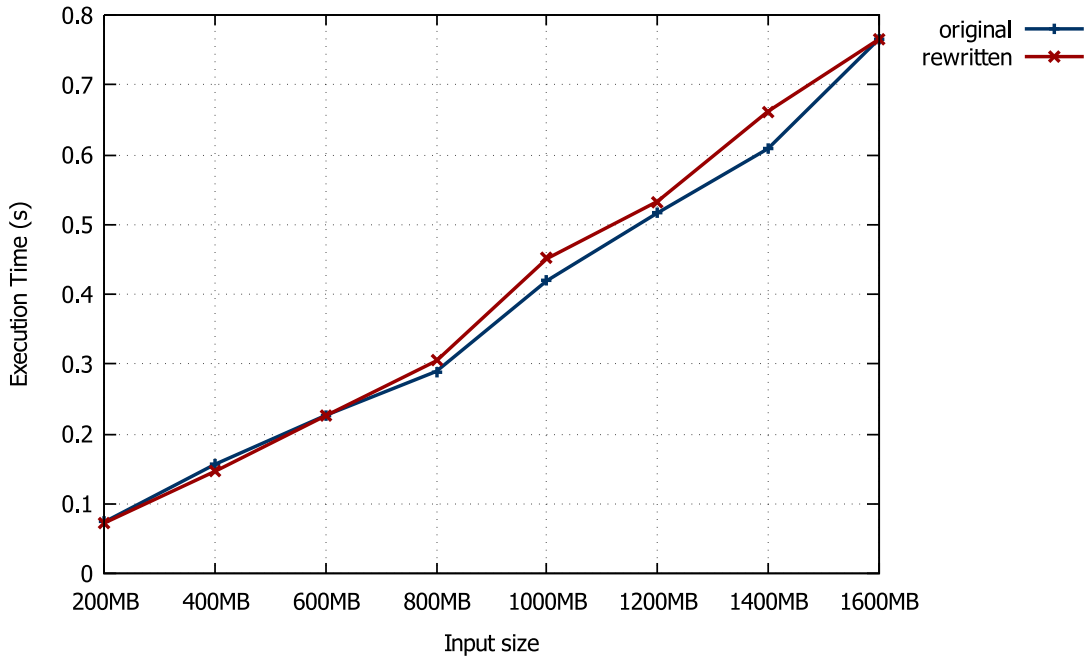


Figure 4.3: Execution Time of A Function Without Dynamic Jump (for a Doxygen function)

Hence, the disassemblers' accuracy at constructing the correct control flow decreases.

In this method for each conditional and unconditional static jump instruction, a new function is defined and the jump instruction is replaced with a **call** instruction to its corresponding function. For conditional jump instructions, the condition is checked in the corresponding function and if the condition does not hold, the return address is not modified so the behaviour remains unchanged. The **call** instruction does not modify the flags so the condition checks can be done safely after the function call.

We applied this method to a function named **gen_codes** in **gzip**[14]. Figure 4.4 shows the original control flow graph of the function and 4.5 shows the control flow graph of the function after the method applied to the function. The control flow graphs are generated by IDA-Pro [26]. As it can be seen from the generated control flow graphs, before the method is applied IDA-Pro is successfully generated the control flow graph including the decision making mechanism (i.e. if-else statements). On the other hand, after the method is applied the control flow graph becomes *flat*. In this case the decision making mechanism and the algorithm of the function is harder to

[14] `http://linux.die.net/man/1/gzip`. [Last accessed on June 26, 2015].

understand. The adversary should trace the function calls in the function in order to understand the algorithm. Plus, for making even harder to understand, these functions can be obfuscated with other methods as well.

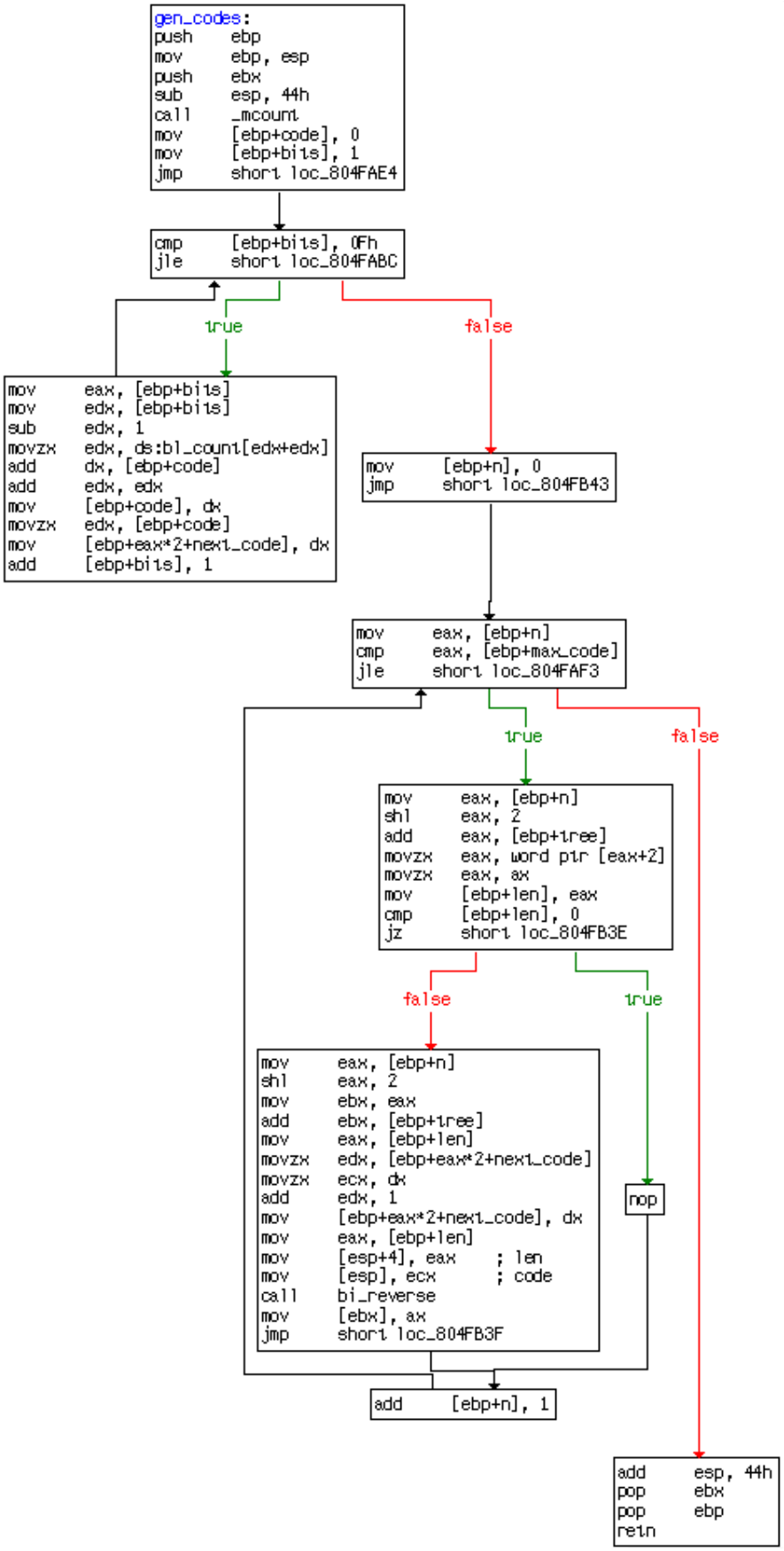


Figure 4.4: Case Study: Original Control Flow Graph

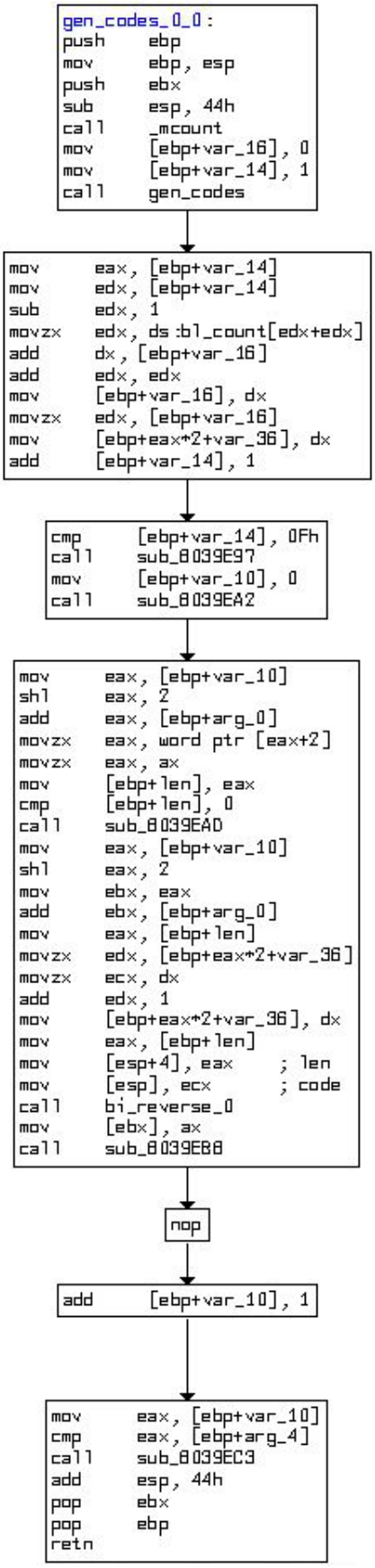


Figure 4.5: Case Study: Control Flow Graph After Branch Function Method

CHAPTER 5

CONCLUSION

In this thesis we studied on static binary rewriting. Static binary rewriting provides capability of inserting some code pieces to executable binary files. Inherently, it is usually used as a subcomponent in a bigger task such as applying software protection methods to executable binary files without source code modification. Therefore, it is important to have standalone implementation for using the static binary rewriting as a subcomponent in other products.

Some prior studies use a special compiler and/or linker in their approaches. Some others use a third party commercial disassembly tool (mostly IDA Pro) to make analysis on the binary file. Unlike these studies, our implementation can work directly on the output of the compiler without any third party disassembly tool or special compiler/linker dependency. To remove the disassembly tool dependency, we used an open source disassembly library which translates the given byte sequence to x86 representation. Then, the required analysis is done by our framework.

In our solution we use the debug information. This brings us two advantages. One of them is the disassembly accuracy. Since the functions begin and end addresses is known, the code data interleaving, which reduces the disassembly accuracy, is much less problematic. The other advantage is the flexibility to select functions from the input program and insert different code pieces to different functions. This is important for applying software protection methods in terms of performance and provided protection. Since, the software protection methods have some performance overhead, instead of applying to all functions, applying to some critic functions reduces the overall performance overhead.

According to the test results, the main cause of our implementation's overhead is caused by handling dynamic jump instructions. As we mentioned, we insert some extra code pieces and redirection maps for translating the dynamic jump targets to their new addresses. That's why handling dynamic jump instructions brings some overhead. On the other hand, the execution time of the function without dynamic jump is nearly not affected.

The most important challenge was handling the dynamic jump instructions in static binary rewriting. We proposed a runtime translation based solution to this problem. As a solution we searched the possible jump targets of the dynamic jump instructions in the binary. Then, the possible jump targets are used at runtime translation. This translation should be fast in order to reduce performance overhead. In the framework the runtime translation of dynamic jump targets are done in constant time algorithmic complexity.

5.1 Future Work

In our study the implementation is done for Linux based operating systems and x86 32 bit architecture. As a future work, the implementation can be adapted to other platforms such as 64 bit architecture and windows operating systems.

Another remaining work is handling the case where the binaries can be loaded into different addresses each time they are executed. *Address space layout randomization* (ASLR)[29] property of the operating systems, libraries and shared objects are the instances of this case. In these cases, since the binary file can be loaded into different address each time it's loaded, the accesses to memory using direct address values are problematic. The address values will be changed when the binary file loaded into different address. These direct accesses to memory addresses normally handled with the help of relocation information in the binary file [22]. The relocation information stores the direct address values that should be updated according to the load address. Then, the address update operation is done while loading the binary file to memory using the relocation information in the file. Therefore, the direct addresses will show the correct addresses after the file is loaded into memory.

In our implementation we insert code pieces that use direct addresses to access data (for example, the address of the extra array which is used at runtime translation of dynamic jump targets). This is problematic when ASLR property of the operating system is active or the binary file is a library or shared object. As a solution the direct addresses which are used should be added to relocation information of the binary file. Therefore, during loading of the binary file the addresses will be updated.

As we mentioned earlier, during rewriting process the rewritten functions (i.e. code inserted functions) are moved to another address. Hence, the debug information of the executable binary file become wrong. In order to fix this issue, after the rewriting process the related debug information can bu updated.

REFERENCES

[1] David A. Libdwarf and dwarfdump. `http://www.prevanders.net/dwarf.html`. [Last accessed on June 26, 2015].

[2] Philippe Biondi and Fabrice Desclaux. Presentation at black hat europe: Silver needle in the skype, 2006. `https://www.blackhat.com/presentations/bh-europe-06/bh-eu-06-biondi/bh-eu-06-biondi-up.pdf`. [Last accessed on June 26, 2015].

[3] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but not academical overview of malware anti-debugging, anti-disassembly and anti-vm technologies. *Black Hat USA*, 2012.

[4] Derek Lane Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[5] Randal Bryant, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective*, volume 2, chapter 3.6–3.7, pages 138–171. Prentice Hall Upper Saddle River, 2003.

[6] Hoi Chang and Mikhail J Atallah. Protecting software code by guards. In *Security and privacy in digital rights management*, pages 160–175. Springer, 2002.

[7] Cristina Cifuentes and Mike Van Emmerik. Recovery of jump table case statements from binary code. In *Program Comprehension, 1999. Proceedings. Seventh International Workshop on*, pages 192–199. IEEE, 1999.

[8] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[9] DWARF Debugging Information Format Committee et al. Dwarf debugging information format, version 4. *Free Standards Group*, 2010.

[10] Tool Interface Standards Committee et al. Executable and linkable format (elf). *Specification, Unix System Laboratories*, 2001.

[11] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture*, chapter 2–3. 2015.

http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf. [Last accessed on June 26, 2015].

[12] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, A-Z*, chapter 3–4. 2015. http://www.intel.com.tr/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf. [Last accessed on June 26, 2015].

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

[14] Zhui Deng, Xiangyu Zhang, and Dongyan Xu. Bistro: Binary component extraction and embedding for software security applications. In *Computer Security–ESORICS 2013*, pages 200–218. Springer, 2013.

[15] James P Farwell and Rafal Rohozinski. Stuxnet and the future of cyber war. *Survival*, 53(1):23–40, 2011.

[16] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, number 6, pages 120–126. ACM, 1982.

[17] Galen Hunt and Doug Brubacher. Detours: Binaryinterception ofwin 3 2 functions. In *Usenix Windows NT Symposium*, pages 135–143, 1999.

[18] Tımea László and Ákos Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30:3–19, 2009.

[19] Michael A Laurenzano, Mustafa M Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. IEEE, 2010.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Acm Sigplan Notices*, volume 40, number 6, pages 190–200. ACM, 2005.

[21] Sun Microsystems. *Linker and Libraries Guide*, pages 115–116. 2004. http://docs.oracle.com/cd/E19683-01/817-3677/817-3677.pdf. [Last accessed on June 26, 2015].

[22] Sun Microsystems. *Linker and Libraries Guide*, chapter 3. 2004. `http://docs.oracle.com/cd/E19683-01/817-3677/817-3677.pdf`. [Last accessed on June 26, 2015].

[23] Robert Muth, Saumya K Debray, Scott Watterson, and Koen De Bosschere. alto: a link-time optimizer for the compaq alpha. *Software: Practice and Experience*, 31(1):67–101, 2001.

[24] Jasvir Nagra and Christian Collberg. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*, chapter 4–9, pages 201–601. Pearson Education, 2009.

[25] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, number 6, pages 89–100. ACM, 2007.

[26] Hex Rays. Ida pro disassembler and debugger. `https://www.hex-rays.com/products/ida/`. [Last accessed on June 26, 2015].

[27] Ted Romer, Geoff Voelker, Dennis Lee, Alec Wolman, Wayne Wong, Hank Levy, Brian Bershad, and Brad Chen. Instrumentation and optimization of win32/intel executables using etch. In *Proceedings of the USENIX Windows NT Workshop*, volume 1997, pages 1–8, 1997.

[28] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. Disassembly of executable code revisited. In *Reverse engineering, 2002. Proceedings. Ninth working conference on*, pages 45–54. IEEE, 2002.

[29] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[30] Serhat Toktamışoğlu. Last man standing or self defensive software. *Defence Turkey*, 8(48):46, 2013.

[31] Richard Wartell, Vishwath Mohan, Kevin W Hamlen, and Zhiqiang Lin. Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 157–168. ACM, 2012.