Software protection through obfuscation

Balachandran, Vivek

2014

Balachandran, V. (2015). Software protection through obfuscation. Doctoral thesis,
Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/62930>

<https://doi.org/10.32657/10356/62930>

Software Protection through Obfuscation

School of Computer Engineering

A Thesis Submitted to the Nanyang Technological University
in partial fulfillment of the requirement of the degree of
Doctor of Philosophy

by

Vivek Balachandran

under the supervision of

Prof. Ng Wee Keong

and

Prof. Sabu Emmanuel

2015

Acknowledgments

Foremost, I would like to express my sincere gratitude to my advisors Prof. Sabu Emmanuel and Prof. Ng Wee Keong for the continuous support of my Ph.D study and research, for their patience, motivation, enthusiasm, and immense knowledge.

I thank my fellow labmates in Nanyang Technological University, Singapore: Shaheen Ansari, Deepak Subrmanyam, Chia Tee Kiah for their kind support.

Many friends have helped me stay sane through these difficult years. Their support and care helped me overcome setbacks and stay focused on my graduate study. I would like to thank Aditya Venkataraman, Chitra Panchapakesan, Ganesh Bharadwaj, Giridharan Karunakaran, Karthik Raveendran, Manaswini Ramkumar, Manisha Mujumdar, Nirnaya Sarangan, Ponnu Jacob, Roshan Wahab, Shubha Nageswaran, Vidhi Patel and Vipin Pillai for making my life wonderful as a grad student.

Last but not the least, I would like to thank my family: my parents G. Balachandran and V. Santhakumari, and my brother Vishakh Balachandran who were always there for me.

Contents

Acknowledgments	ii
List of Figures	viii
List of Tables	x
Abstract	xi
1 Introduction	1
1.1 Software Security	2
1.2 Motivation	3
1.2.1 Software Obfuscation	6
1.3 Problem Definition	6
1.4 Contributions	8
1.4.1 Obfuscation by Hiding Control Flow in Stack	8
1.4.2 Obfuscation and Encryption	9
1.4.3 Function Level Control Flow Obfuscation	9
1.4.4 Return Oriented Obfuscation	10
1.5 Thesis Organization	10
2 Literature Survey	12
2.1 Security During Development	13
2.2 Security During Operation	15
2.3 Security During Maintenance	17
2.4 Security During Deployment	18
2.4.1 Software Obfuscation	18
2.4.2 Software Tamper Resistance	20
2.4.3 Software Diversity	21

2.5	Software Obfuscation	23
2.5.1	Data Obfuscation	23
2.5.2	Design Obfuscation	25
2.5.3	Control Flow Obfuscation	28
2.5.4	Evaluation Metrics for Obfuscation	38
2.6	Preliminaries	41
2.6.1	Analysis of Binary Program	41
2.6.2	Disassembly of Binary Program	42
2.6.3	Junk Byte Insertion	43
2.6.4	Self Modifying Program	43
2.7	Conclusion	44
3	Software Obfuscation by Hiding Control Flow Information in the Stack	45
3.1	Motivation	45
3.2	Threat Model	46
3.3	Proposed Method	47
3.3.1	Offline Obfuscation	48
3.3.2	Runtime De-Obfuscation	51
3.3.3	Runtime Re-Obfuscation	52
3.3.4	Junk Bytes Insertion	54
3.3.5	Conditional Instructions	54
3.3.6	Randomization to Improve Obfuscation	56
3.3.7	Example Code Snippet from IDAPro	57
3.4	Implementation	57
3.5	Performance Evaluation	60
3.5.1	Potency Against Static Analysis	61
3.5.2	Potency Against Dynamic Analysis	63
3.5.3	Cost of Obfuscation	66
3.5.4	Stealth Analysis	68
3.5.5	Comparison with Other Algorithms	69
3.6	Conclusion	70

4	Software Protection by Obfuscation and Encryption	72
4.1	Motivation	72
4.2	Threat Model	73
4.3	Protection Model	74
4.4	Proposed Algorithm	75
4.4.1	Obfuscation	76
4.4.2	Encryption	81
4.5	Mathematical Analysis	85
4.5.1	Validating Decryption Process	86
4.5.2	Validating Encryption Process	87
4.5.3	Complexity in Finding the Target Addresses	87
4.6	Experimental Evaluation	88
4.6.1	Instruction Disassembly Error	88
4.6.2	Control Flow Disassembly Errors	89
4.6.3	Time Overhead	90
4.6.4	Time Overhead with Varying Input	91
4.6.5	Program Size Overhead	93
4.6.6	Comparison with Other Algorithms	94
4.7	Obfuscation of Factorial Program	95
4.8	Conclusion	96
5	Function Level Control Flow Obfuscation	98
5.1	Motivation	98
5.2	Proposed Method	99
5.2.1	Selecting the Instructions	100
5.2.2	Creating and Inserting Labels	101
5.2.3	Displacing the Code Fragment	101
5.2.4	Hiding the Jump Instructions	102
5.2.5	Nested Obfuscation	105
5.3	Implementation	106
5.3.1	Markers with Copt	106

5.3.2	Program Translation by Python Code	107
5.4	Performance Evaluation	108
5.4.1	Instruction Disassembly Error	108
5.4.2	Control Flow Error	109
5.4.3	Function Level Obscurity	110
5.4.4	Space Overhead	112
5.4.5	Time Overhead	113
5.4.6	Time Overhead with Varying Input	113
5.4.7	Nested Obfuscation Overhead	115
5.5	Comparison with Other Algorithms	115
5.6	Obfuscation of Factorial Program	117
5.7	Conclusion	117
6	Return Oriented Obfuscation	120
6.1	Motivation	120
6.2	Proposed Method	121
6.2.1	Splitting the Function	122
6.2.2	Pushing the Return Address to the Stack	123
6.2.3	Inserting Return Instruction	124
6.2.4	Restoring the Stack	125
6.2.5	Shuffling the Code Fragments	126
6.3	Implementation	126
6.4	Overhead Analysis	129
6.4.1	Space Overhead	129
6.4.2	Time Overhead	130
6.5	Performance Evaluation	131
6.5.1	Instruction Disassembly Error	132
6.5.2	Function Level Obscurity	133
6.5.3	Space Overhead	134
6.5.4	Time Overhead	135
6.5.5	Time Overhead with Varying Input	137
6.6	Obfuscation of Factorial Program	137
6.7	Conclusion	138

7 Contributions and Future Work	140
7.1 Our Contributions	141
7.1.1 Software Obfuscation by Hiding Control Flow Information in Stack	141
7.1.2 Software Protection by Obfuscation and Encryption	142
7.1.3 Function Level Control Flow Obfuscation	142
7.1.4 Return Oriented Obfuscation	143
7.2 Future Work	144
7.2.1 Obfuscation for Distributed Programs	144
7.2.2 Malware Detection	146
List of Publications	147
References	148

List of Figures

1.1	Lifetime of a software program	2
1.2	Compilation and reverse engineering	5
2.1	Software security overview	13
2.2	Variable splitting	24
2.3	Converting static definition to procedural	24
2.4	Class coalescing	26
2.5	Type hiding	27
2.6	Opaque predicates	29
2.7	Double process based obfuscation [GCT05]	32
2.8	M-process execution [GCT05]	33
2.9	Signals in operating system	34
2.10	An example of signal based obfuscation	35
2.11	Clustered dynamic mutation	37
2.12	Self modifying code based obfuscation	38
3.1	Storing code information in stack	51
3.2	Obfuscation of <i>jmp</i> instructions	53
3.3	Junk byte insertion	55
3.4	Partial junk insertion example	56
3.5	Reverse engineered code from IDAPro	58
3.6	Jump instructions revealed during dynamic analysis	65
3.7	Time complexity of dynamic analysis	66
3.8	Mahalanobis distance of binary programs	69
3.9	Algorithm performance comparison	70

4.1	Obfuscation of jump instruction	79
4.2	Assembly program after obfuscation	83
4.3	Time complexity of linear search program	92
4.4	Time complexity of bubble sort program	93
4.5	Assembly representation of factorial program	95
4.6	Obfuscated factorial program	97
5.1	Overview of the algorithm	100
5.2	Label insertion	101
5.3	Displacing the code fragment	102
5.4	Junk byte insertion	104
5.5	Static junk byte insertion	105
5.6	Implementation overview	106
5.7	Time complexity of linear search program	114
5.8	Time complexity of bubble sort program	115
5.9	Space overhead with nested obfuscation	116
5.10	Time overhead with nested obfuscation	116
5.11	Comparison with other algorithms	117
5.12	Assembly representation of factorial program	119
6.1	Overview of the algorithm	122
6.2	Inserting labels at the splits	123
6.3	Inserting return address in the stack	124
6.4	Inserting return instruction	125
6.5	Shuffling of code fragments	127
6.6	Function call graph of nqueens program	127
6.7	Main function of nqueens program	128
6.8	Shuffling algorithm	129
6.9	Time complexity of linear search program	137
6.10	Time complexity of bubble sort program	138
6.11	Assembly representation of factorial program	139

List of Tables

3.1	Instruction disassembly error	62
3.2	Control flow errors	62
3.3	Percentage of jump instructions revealed	65
3.4	Space overhead	67
3.5	Time overhead	68
3.6	Mahalanobis distance before and after obfuscation	70
4.1	Instruction disassembly error	89
4.2	Control flow errors	90
4.3	Time overhead	91
4.4	Space overhead	93
4.5	Comparison with other algorithms	94
5.1	Instruction disassembly error	109
5.2	Control flow error	110
5.3	Function Level Obscurity	111
5.4	Space overhead	112
5.5	Time overhead	114
6.1	Instruction disassembly error	133
6.2	Function Level Obscurity	134
6.3	Space Overhead	135
6.4	Time Overhead	136

Abstract

Software, over the years, has evolved from free code given along with the hardware for free to a valuable asset, automating almost all of the electronic devices and systems. The growth in the software analyzing tools helped the software developers to analyze and better their software programs. Unfortunately, the same software analyzing tools are used to reverse engineer software systems with malicious intent such as for stealing the intellectual property of the developer, for identifying the vulnerabilities in a program and exploiting them and for unauthorized modifications of the program (tampering). The financial losses incurred by the software industry due to these are in billions.

One of the mechanisms to make software reverse engineering harder for an attacker is software obfuscation. Software obfuscation is the process of transforming a program into a semantically equivalent but hard to understand form. The primary objective of our research is to develop software obfuscation algorithms for binary programs so as to make reverse engineering harder for an attacker.

In the first part of our research we developed a new software obfuscation algorithm based on self modifying code using stack to conceal the control flow information of binary programs. This will make the reverse engineering of the binary program to assembly level representation harder. In this method, our algorithm translates the control flow instructions, like jump instructions, to normal instructions. The target address of the jump is stored in the stack and the original control flow instructions are reconstructed during runtime by reconstruction instructions.

In the next part of our research we proposed a method where encryption and obfuscation are used hand in hand to improve the security of software. In this algorithm, the obfuscation technique used is similar to our previous self modifying code approach. In this method the target addresses are stored in the static data area in an encrypted form.

This target addresses are decrypted only during runtime and is re-encrypted after the use. This makes it harder for the attacker to retrieve the target address from the data area.

Following the two control flow obfuscation techniques obscuring the control flow within functions, we developed an inter-functional control flow obfuscation technique. One disadvantage of most control flow obfuscation algorithm is that the functions are not affected and the reverse engineering tools can find the beginning and end of a function even after obfuscation. In this method code fragments from each function is stripped from the original function and is stored in another function. Each function will be having code fragments from different functions, thereby creating a function level shuffled version of the original program. Control flow is obscured between and within the function by this method

In the last part of our research, we developed and implemented an inter-functional obfuscation based on return instruction. In this method, each function is split into various function blocks, each ending with a return instruction. The function blocks are independent blocks and can be moved within the program, letting the obfuscator shuffle the function blocks, similar to our function level obfuscation technique.

A research area of interest, which we can be pursued in the future, is to develop obfuscation algorithm for distributed programs. Devising obfuscation algorithms which take advantage of the features of distributed systems to generate potent obfuscations is a promising future direction. Another research area that can be explored in the future is to use the knowledge of obfuscation to detect obfuscated variants of known malwares. The basic research challenge in this domain is to find features of a program that are invariant to obfuscation.

0.4pt

Chapter 1

Introduction

Software, over the years, has evolved from free code given along with the hardware for free to a valuable asset, automating almost all of the electronic devices and systems. The growth in the software analyzing tools helped the software developers to analyze and better their software programs. Unfortunately, the same software analyzing technologies [[Hex](#)] [[Oll](#)], are used to reverse engineer software systems with malicious intent such as for stealing the intellectual property of the program and for identifying the vulnerabilities in a program and exploiting them. Tools and documents on software reverse engineering are readily available from various websites [[Hex](#)] [[Oll](#)] [[MRB02](#)].

In this chapter, firstly we will briefly explain about the software security threats during the lifetime of a software. Then we will discuss the relevance of the study of software protection against reverse engineering and discusses about the particular protection technique, software obfuscation. We discuss about the motivation and challenges in software obfuscation as a security technique. The different evaluation metrics used for evaluating a software obfuscation method will also be discussed. Finally, we will discuss about the short comings of the current techniques, our problem definition and our contributions, so far.

1.1 Software Security

The notion of software security in this context refers to the security of application softwares. We can broadly classify the lifetime of a software into four; software development, software deployment, software operation and software sustenance as shown in Figure 1.1. Software faces different, but important, security threats at each of these levels. In this section we will briefly discuss about the threats at each level.

Figure 1.1: Lifetime of a software program

Many vulnerabilities in the software is due to bad programming practices [Sta07]. So, the security flaws, in many cases, are often embedded by the software developers. Problems like buffer overflow, memory leakage, dangling pointers are some of those vulnerabilities. This happens during the development of a software. Using good programming practices, analysing the risk factors, validating the input cases [LT08], using garbage collection [DLM⁺78] are methods to reduce these vulnerability bugs in the program.

Once software is deployed to the users, it is exposed to a hostile environment [Auc96]. The software is exposed to static and dynamic analysis. Tampering the software, reverse engineering and piracy are some of the intentions of an attacker. An attacker in this case is a user in itself and hence has the software and the right platform (his system) with complete access rights. Tamper resistance [Auc96], software diversity [vO03] and obfuscation [CTL97] are techniques which are used against these attacks. Software obfuscation which is a mechanism to make reverse engineering harder is an interesting area of research. We focus our research primarily in this area.

Security threats posed during the operation of a software programs are access right violations and malicious infections. In todays computing environments, applications written by many different authors all co-exist, sharing physical resources which include the network and file-system. When different application modules share resources among themselves the privileges for each application and the modules in each application to access certain resources should be specified properly. Access control [Bis03] is one of the ways to provide some accountability, through authentication and identification. Another threat is the malicious infections in the software [EAAT04]. Software which is infected may infect other programs and affect system performance.

Proper maintenance is an important factor for software security. Patches have to be released and applied regularly as the vulnerabilities are found. A proper patch management to ensure the authenticity of the patch and protection mechanisms against reverse engineering the patch are also necessary [SJ10].

1.2 Motivation

The 2010 annual study on software piracy [Stu] conducted by Business Software Alliance (BSA) and IDC shows that the total loss for the software industry is about 59 billion US dollars. This figure has nearly doubled in real terms from the year 2003. In many of the intellectual property thefts reported, some employ software reverse engineering techniques. Atari Games v. Nintendo [DLOa] (1992), Sega v. Accolade [DLOb] (1991), Sony v. Connectix [New] (2000), Blizzard v. bnetd [Hwa] (2002), SAS v. World Programming [ED] (2010) are some of the law suits involving software thefts using reverse engineering.

In addition, reverse engineering a binary program, to a higher level abstraction poses the threat of exposing vulnerabilities of the program. An adversary may exploit it for his advantage. Inserting Trojans, viruses and worms, denial of service are common attacks on programs. The threat by these attacks can be lethal; the recent stuxnet attack [Kus] is an

example. Stuxnet virus was designed to cause damage to Siemens industrial equipment. It caused damage to the machinery at Irans uranium enrichment facility. Iran was the most affected country by stuxnet, followed by Indonesia and India [Kus]. In this era of computerization, where software plays important role in security and business, software security is highly important.

The major threats to closed source software security are tampering [Auc96] [LCM⁺00], piracy and reverse engineering [CP06]. An intended modification to the program without the permission of the owner/distributor of the software is referred to as tampering. There are many methods [Auc96] [LCM⁺00], to obstruct tampering and software with tamper resistance algorithms is often referred to as tamper resistant software. Unauthorized copying of the software is often referred to as software piracy. However, reverse engineering is the basis of many of these issues relating to software security and copyright.

Further, software reverse engineering plays an important role in tampering and piracy. Software is released mostly in binary representation. To make a useful modification in the program, the binary program has to be reverse engineered to a higher level abstraction so that the attacker understands the logic of the program. Similarly, an adversary needs to reverse engineer the binary code, to higher level abstractions to crack the key authentication of the binary program, to generate a pirated copy of the software.

The process of compilation of a program and reverse engineering are illustrated in Figure 1.2. Compilation is the process of translating the higher level language program to machine code. Reverse engineering is the inverse of compilation where higher level structures and semantics are recovered from a machine code program. Any act of transforming a code from a lower level representation to a higher level representation is software reverse engineering. Translating binary code to assembly program, transforming assembly program to source level program, transforming java byte codes to java source program [MuAMM06] [HCT06] are all forms of reverse engineering.

Figure 1.2: Compilation and reverse engineering

Disassembly is the process of transforming the binary code to its equivalent assembly representation. One point where software protection techniques can be applied is at the binary making it hard for an attacker to successfully disassemble the binary program. Decompilation, is the process of transforming the assembly or binary program to source code of higher level programming languages. Software protection mechanisms can be applied at the source code level so that an attacker cannot get the original source program.

Different protection mechanisms like tamper resistant software [Auc96], diversifying softwares [Coh93], patch management [SJ10] [CTT05], access right protocols [Bis03] are used to prevent modification of the software by third party attackers. But in a threat model where the attacker and the user of the software are the same, software obfuscation is the effective method to thwart reverse engineering. A brief explanation on software obfuscation is given in the following section.

1.2.1 Software Obfuscation

Software obfuscation [CTL97] is an approach, where the software developer obscures the code to a level that it is harder for the adversary to reverse engineer and make sense out of the reverse engineered program. In a perfect scenario, the obfuscator wants the program to be as obscure so that it is economical for the adversary to develop the program from scratch than reverse engineering the program.

The general obfuscating transformation definition by Collberg et al. [CTL97] is: Let $T : P \rightarrow P'$ be a transformation of a source program P into a target program P' . The $T : P \rightarrow P'$ is an obfuscating transformation, if P and P' have the same observable behavior. More precisely, in order for $T : P \rightarrow P'$ to be a valid obfuscating transformation the following condition must hold:

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

There are various software obfuscation techniques and algorithms which are discussed in chapter 2. The metrics used for measuring the quality of an obfuscation algorithm are also discussed in chapter 2.

1.3 Problem Definition

A native binary program released to the public is vulnerable to software reverse engineering attacks. Our aim, in general, is to design software obfuscation algorithms, for distributed and non distributed programs, so as to make software reverse engineering hard for an attacker. When to apply the algorithm, algorithms performing well against dynamic analysis, developing algorithms producing stealthy obfuscated programs, are specific details of our research and will be discussed in this section.

Obfuscation algorithms can be applied at different points during the compilation of a program, ranging from source level to binary level. We concentrate more on developing protection algorithms at binary level. The first step of reverse engineering a binary program to higher level representation is, disassembly. If the adversary is not able to get the correct assembly representation of the binary program, it will corrupt all further reverse engineering steps.

Also, performing obfuscation mechanisms at binary level, one does not have to worry about the compiler optimizations. The obfuscation algorithm is performed in the final binary after optimizations and hence all the changes due to obfuscations will remain in the code. If obfuscations are done at source level, the compiler optimization has to be switched off because the optimizations may remove the obfuscations. There is a trade off between security and effectiveness with the introduction of obfuscation. Obfuscation may not always lead to the most optimized code as it involves adding new instructions to confuse the reverse engineering tools. A security engineer should consider this before obfuscating a program.

In the current literature, obfuscating algorithms focus on thwarting static analysis of the program. In static analysis, an adversary uses static reverse engineering tools like IDAPro [\[Hex\]](#) to reverse engineer the program. In static analysis, the attacker tries to create the assembly representation of the binary program without executing the program. . In a dynamic analysis scheme, the execution of the program is monitored and analyzed to get the context information during runtime. One of our aims is to design obfuscation algorithm which are potent against static analysis and dynamic analysis.

Stealth is a measure of obfuscation, which evaluates how similar is an obfuscated program to the normal binary program. It is important that the obfuscated programs do not stand out and look similar to normal programs. If it is vastly different from normal binary programs then it can be easily spotted by a reverse engineer. Even though stealth

measure is important we found only one paper [LKK10] in the literature which has discussed about the stealth of their algorithm. One of our aims is to design obfuscation algorithms that generate stealthy binary programs.

Another aim of our research is to explore the possibility of combining software obfuscation techniques along with other software protection techniques. One of the approaches towards it is combining software obfuscation techniques and encryption techniques.

Extending the obfuscation techniques from intra functional obfuscation to inter functional obfuscation is an area of interest for our research. The current literature discusses mostly about obfuscating the program within functions, maintaining the inter-functional dependencies. We are interested in developing inter functional obfuscation by manipulating control flow instructions like jump and return.

1.4 Contributions

There are four major contributions in the course of our research work. The contributions of our work are the following

1.4.1 Obfuscation by Hiding Control Flow in Stack

In this work we proposed a new binary obfuscation algorithm, based on dynamic mutation. The control flow information of the binary program are removed and hidden in the stack as local variables. Reconstruction instructions are added in the program that will dynamically restore the control flow during program execution. The major conceptual contribution of this work is the hiding of control flow information in the data area, stack.

One of the challenges in this method is to maintain the semantics of the program while dynamically changing it during runtime. The dynamic changes of the program may need writing permissions of the program code area. By adding the de-obfuscation and re-obfuscation instructions in the successor and predecessor blocks enabling the permission

for a short period helps in attaining the semantic equivalence and changing the write permissions for a very small period.

Experimental evaluation shows that our algorithm has better potency against static and dynamic analysis than the state-of-the-art algorithms like signal based obfuscation [PDA07] and self modifying code obufscation [SE10]. We attained a control flow obfuscation of 61.3 % and instruction disassembly error of 79.8 %.

1.4.2 Obfuscation and Encryption

In this work we devised a dynamic obfuscation algorithm in conjunction with encryption. The control flow information removed from the program is stored in an encrypted form in this method. The control flow information, the target addresses of jump instructions, is stored in static data area in the program in encrypted form. The encryption keys are also stored in the same data area. The keys and control flow information and mixed together and it is difficult for an attacker to identify a key.

Since the encryption keys are stored along with data in the program, one of the major challenges was the security of the keys. We have stored the keys along with data and treated keys just like data so that it is indistinguishable from data. While encrypting and decrypting particular data element we encrypted and encrypted the entire data area with keys and data instead of just one data element. So, the keys and data value are dynamically changed during runtime.

Experimental evaluation shows that this method has attained better control flow obfuscation than our previous method. The time and space efficiency of this method is also better than our previous algorithm.

1.4.3 Function Level Control Flow Obfuscation

In this work we discuss a new approach which obfuscates the control flow across functions. In this method code fragments from each function is stripped from the original

function and is stored in another function. Each function will be having code fragments from different functions, thereby creating a function level shuffled version of the original program. One of the advantages of this method is that control flow of the program is obscured within the functions like other control flow obfuscations and it obscures inter-functional dependencies. This is the first obfuscation technique, working at the lower level, which is capable of doing nested obfuscation and inter functional obfuscation.

Control flow obfuscation of this method, 69.11 %, is better than our previous methods. Instruction disassembly error is 78.32 % comparable to our best result of 79.78 % of stack based obfuscation. We measured the function level obscurity by calculating the cyclomatic number of the obfuscated program. The cyclomatic number is 1.96 times the number of functions in the program.

1.4.4 Return Oriented Obfuscation

In this work we discuss a new function level obfuscation techniques based on return instructions. In this method, each function is split into various function blocks, each ending with a return instruction. The function blocks are independent blocks and can be moved within the program, letting the obfuscator shuffle the function blocks, similar to our function level obfuscation technique. The control flow of the original program is maintained by manipulating the return address in the stack.

The advantage we have in this method, compared to our previous function level obfuscation is that the code fragments are not selected based on any instruction sequence. The functions can be split into more code fragments than the previous method. The function level obscurity measure of this method is 3.25 times the number of functions in the program.

1.5 Thesis Organization

Rest of this thesis is organized as follows.

Chapter 2 gives the background theory and literature review in the areas of software protection and explains software obfuscation in detail.

Chapter 3 presents our proposed software obfuscation algorithm by hiding control flow information in the stack.

Chapter 4 presents our proposed algorithm where software obfuscation and encryption are combined together to improve the security.

Chapter 5 presents our proposed algorithm for inter functional control flow obfuscation technique. We discuss the jump instruction based obfuscation, similar to the previous two algorithms but also obscuring the functions by splitting and shuffling them.

Chapter 6 presents another function level obfuscation technique based on return instruction rather than jump instructions. In this chapter we discuss a method to split each function into separate functional units before shuffling it.

In *chapter 7*, contributions and future works directions are presented.

Chapter 2

Literature Survey

Software protection is a blend of software security protocols, cryptographic techniques and software engineering among other disciplines [vO03]. Software is exposed to completely diverse attacks and threats and this provides diverse research opportunities.

The security of software is threatened at various points of its lifecycle. The four major periods of software lifecycle are producing, deploying, operating and maintaining the software. The security concerns during each of these periods are different. In this chapter we review the various software security concerns and existing protection techniques. More emphasis is given to security threat faced by the software once it is deployed to users. Our main focus is on protection mechanisms against software reverse engineering. Then we discuss about some preliminary terminologies related to understanding the algorithms devised as a part of our research in the past two years.

As shown in figure 2.1 , the softwares security can be threatened during its,

- *Development* : A security problem is more likely to arise due to a problem in the implementation. Finding proper abuse cases and risk assessment during the development of a software will help in reducing the vulnerabilities like buffer overflow [ZH10] in the software.
- *Deployment* : Once the software is released to the public it is exposed to different kinds of attacks. Tampering the software, reverse engineering the software,

intellectual property theft etc., are some of the attacks the software get exposed to. Methods like tamper resistance, software obfuscation, software diversity, access control etc, will make it difficult for an attacker to break through.

- *Operation* : Security threats posed by the software during its execution is a valid concern. If software is infected by a virus then each time the software executes, it is raising a threat to the entire system. The privileges of the software to access different resources is also a vulnerability in itself. Access control mechanisms should be in place to make sure only the right function in a program by the right user get access to a resource.
- *Sustainment* : Proper maintenance is an important factor for software security. Patches have to be released and applied once vulnerabilities are found. A proper protection mechanism for the patch against reverse engineering is necessary.

Figure 2.1: Software security overview

2.1 Security During Development

Using methods during software development to eliminate vulnerabilities gives more resilience to the software. Software is analysed using static analysis tools for security

risks like buffer overflow, memory leakage etc. Automated input validation [ABG12] and garbage collection [YSY10][LWC⁺14] are two methods used to eliminate such vulnerabilities from the software.

Input validation makes sure the program accepts only from a range of accepted input values. If the user has the liberty to provide any input then it will often lead to finding an input which has negative security implications by an attacker [OWAb]. Some of the major software security problems like buffer overflows [Fos05], SQL injection attacks [KP11][KP12][JS12], and command-injection attacks [OWAa] are ultimately input validation problems.

Data input to a program is either valid or invalid. What defines validity can be dependent on the semantics of the program. Good security practice is to definitively identify all invalid data before any action on the data is taken. And, if data is invalid, one should act appropriately [OWAb].

Essentially input validation is the enforcement of constraints that any input must satisfy before it is accepted to raise external effects. In [LT08], the authors propose a method where input validation model is automatically recovered from the program source code. The model is then represented in a validation flow graph, which is a variant of control flow graph. By analysing the validation flow graph an understanding of the input validations features can be obtained. Test cases for testing input validations are then generated.

A form of automatic memory management which attempts to reclaim the garbage memory, the memory occupied by objects that are no longer in use by the program, is known as garbage collection [DLM⁺78]. The overhead of manual deallocation of memory is removed by using garbage collection. As a result, certain types of bugs can be substantially reduced [YSY10]:

Garbage collection frees the programmer from manually dealing with memory deallocation. As a result, certain categories of bugs are eliminated or substantially reduced:

- *Dangling pointer bugs*: This bug happens when a memory area is freed by the program and there are active pointer variables pointing to the memory. This memory area may get re-assigned by the system for another use. This will result in unpredictable results.
- *Double free bugs*: Double free bugs happen when the program tries to free a memory area which has already been freed. The memory area may have been reassigned for another use and freeing the memory area a second time might result in crashing the program.
- *Memory leaks*: When the program fails to free the memory which is not used by the program and not reached through any pointer variables, memory leak happens. The memory area is still allotted to the program but it cannot be reached. This will lead to memory exhaustion if such behaviour is repeated indefinitely.

2.2 Security During Operation

One of the security concerns of software during its execution is the environment in which the software operates. In today's computing environments, applications written by many different authors all co-exist, sharing physical resources which include the network and the system. The sharing of physical resources between many different applications brings with it many security issues which need to be dealt with. Access control [Bis03] is one of the ways to provide some accountability, through authentication and identification.

In a discretionary access control method, an individual who owns an object can either allow or deny access by others to the object. One example of this is the typical POSIX access controls on UNIX, where the owner of a file is allowed to set read, write, and execute access for other members of the group and everyone else.

In an originator controlled access control, the ability to access an object is controlled by the creator of an object [Bis03]. A common example of such an approach is digital rights management in digital media, where the creator attempts to retain control over access to the work after it is distributed to the buyer [LSNS03].

Under a mandatory access control, access policy for an object is set by an individual other than the owner or creator of the object (or same individual assuming a different role). The user, even if they own the object, cannot change the mandatory access control policy for that object. Processes that run for the user cannot modify the policy either.

History based access control considers the entire execution and the rights of the running code executed so far to decide upon the access right of the next instruction [Bar05]. The observation of any security related activity is called an event, e.g. opening a socket connection. A sequence of such events makes the history. A programmer can devise a security policy in which a certain instruction is given execution rights based on the access control of its history. This will enforce that certain actions can only be performed through certain paths [SS94].

Another threat to security by an executing program is that the program may be malicious, infecting other programs and the system[PYY14][FSR09]. Anomaly-based detection[AK08][WCW⁺13] is one of the methods of detecting malwares. It usually occurs in two phases; a training (learning) phase and a detection (monitoring) phase. During the training phase the detector attempts to learn the normal behaviour. The detector could be learning the behaviour of the host or the program or a combination of both during the training phase. PAYL [WS04] is a tool based on dynamic anomaly based detection, which calculates the expected payload in each socket. A byte frequency distribution is created which allows for a centroid model to be developed for each of the hosts services. The detector compares incoming payloads with the centroid model and if the difference is more than the threshold it flags it as malicious. In [HCS09] another

method is proposed, where existing malwares are indexed using function call graphs. If a new program has a function call graph similar to any of the indexed malwares then it is flagged as malware.

Another method is signature-based detection attempts to model the malicious behaviour of malware and uses this model in the detection of malware. The collection of all of these models represents signature-based detection knowledge. This model of malicious behaviour is often referred to as the signature and any executable following the model is flagged as a malware [EAAT04] [IKA95].

2.3 Security During Maintenance

A few years ago, patch management[LKR09][BS03] was not considered so seriously. Once deployed, many systems were infrequently or never updated. The rise of widespread worms and malicious code targeting known vulnerabilities on unpatched systems, and the resultant downtime and expense they bring, makes patch management significant [SJ10] [CTT05].

An organization needs a point person or team that is responsible for keeping up to date on newly released patches and security issues that affect the systems and applications deployed in its environment. This team can also take the lead in alerting administrators and users of security issues or updates to the applications and systems they support and use.

Another threat is that the patch can get reverse engineered. It is much easier to wait for a patch to be released and just look at the code in the patch than tearing the whole program apart. An attacker can reverse engineer the patch to determine what it does, thereby identifying where the flaw exists in unpatched systems.

2.4 Security During Deployment

Software once distributed out to the world, it is exposed to a hostile environment. An attacker can statically analyse the software program using static analysis tools like IDAPro [Hex] for reverse engineering the program. An attacker can also analyse the software dynamically using a virtual machine. The security measures against these attacks are the focus of our study. Tamper resistance[Hei11][CKY09][CFS⁺09], software obfuscation[RM13][KM14] and software diversity[HNL⁺13][PPSS14] are some of the methods used in opposing these threats.

2.4.1 Software Obfuscation

Software obfuscation [CTL97] [CPN⁺09], is a security method to harden the process of reverse engineering. Software obfuscation is a practical approach, where the software developer obscures the code to a level such that it is harder for the adversary to reverse engineer and make sense out of the reverse engineered program. In a perfect scenario, the obfuscator wants the program to be as obscure so that it is economical for the adversary to develop the program from scratch than reverse engineering the program.

Software obfuscation can be broadly classified into layout [Han03], design [SNM03b], data [CTL97] [CTL], and control [WDHK01] [PDA07] [SE10] obfuscations.

Layout obfuscation [Han03], refers to obscuring the layout of the program. For example deleting comments, removing debugging information, renaming variables and changing the source code formatting falls under the category of layout obfuscation.

Design obfuscation [SNM03b][GZ10] tries to obscure the design of the software systems. For example, in the case of object oriented programs, obfuscations such as splitting classes, hiding type information and merging classes will obscure the design intend of the program.

Data obfuscation [CTL97] [CTL], is deployed to prevent the adversary from extracting information from the data used in the program. Data structures used in the program and the data values can give out information regarding the nature of the program. Obfuscation techniques like array splitting, data to procedure conversion, variable splitting changing variable life time are examples for data obfuscation.

Control flow obfuscation [WDHK01] [PG05] obscures the control flow information of the program. Control flow of a program gives logical meaning to the program. Control flow flattening [WDHK01], obfuscation using opaque predicates [CTL], are some methods which give control flow obfuscations.

Another classification of obfuscation is based on the time point in compilation during which the obfuscation is performed. Obfuscation can be done at different points of program compilation. Obfuscation is done either at source code level [CTL] or at binary level [PDA07] [SE10][DL13][WGXW10].

At source code level, the program is obscured by methods like class merging, class splitting, and variable renaming. Before the compilation of the program the obfuscator obscures the program in such a way that it is hard to understand the logic of the program. The program is then compiled to and the machine code is released to public. The basic concept of source code level obfuscation is that even if the attacker managed to reverse engineer the code, it will be difficult for the attacker to understand the logic of the program. Design obfuscation is an example for source code level obfuscation. To attain the security of the design of an object oriented program source code level obfuscations like class splitting [CTL97], inserting bogus classes and class merging [CTL97] are used.

In lower level obfuscation binary program is obscured so that it is hard for an attacker to disassemble the binary to assembly. Various binary level obfuscations are discussed in [PDA07] [SE10]. Binary level obfuscation obscures the binary program and makes it difficult for an adversary to disassemble the binary to correct assembly level representation. Reverse engineering of a binary program starts with the disassembly of binary

program to assembly level representation. Obfuscation at the binary level thus means corruption during disassembly process and giving corrupted assembly program. This further corrupts the programs in higher level representations in the progressive steps of reverse engineering to get the source code [CG95] [oQ].

A detailed explanation of different types of software obfuscation algorithms in different obfuscations types are discussed in section 2.5.

2.4.2 Software Tamper Resistance

Tampering of software refers to unauthorized modification of the program. Tamper resistance offers authenticity to the program. The goal of tamper resistance software is to identify if the software is modified by an attacker or an automated program.

In [Auc96], Aucsmith discusses a method to develop tamper resistant software. The two parts of the tamper resistant software architecture discussed are: Integrity Verification Kernel and Interlocking Trust Mechanism.

Integrity verification kernels are small code segments added to the program to verify the integrity of the code segments of the program. The integrity verification kernels are made robust by encryption. They are encrypted in parts and decrypted on demand. The integrity verification kernel is split into parts and each part is encrypted with a key. Each part contains the key to decrypt the part that follows it in the control flow of the program. During runtime at the end of execution of one part, the part which follows is decrypted and the current cell is encrypted. Thus only one part of the code is exposed at a time.

The interlocking trust mechanism uses the inherent strength of the integrity verification kernel (IVK), that it is robust, and checks other IVKs. This mutual checking will help in increasing the resistance to tampering greatly.

Another method to resist software tampering is explained in [GHD10]. In this method the program is embedded with a virtual machine and extra code section called guards

are packaged together and distributed. The purpose of guards is to check for tamper resistance and the virtual machine enables just in time decryption of the program.

Each guard makes use of a light weight hash functions to check a range of memory addresses. A guard calculates the checksum of the allotted memory area to makes sure the program is not tampered.

The encrypted application is decrypted just in time by the virtual machine. The decrypted code and the guards are cached, so as to avoid excessive time overhead.

In [TYM04], a bulk encryption technique is discussed where the entire program is encrypted and during runtime the entire program is decrypted at once. The decryption procedure is placed at the entry point of the program. One of the advantages of this method is that statically no details of the program can be known. Another advantage is that no meaningful modification of bits is possible, statically. However, this method has a disadvantage as the program is completely visible after the process of decryption. An attacker can dump the image of the program after decryption and there by getting the original program.

Lie et al. discusses a novel partial encryption scheme with the help of a hardware implementation of a form of execute only memory (XOM) in [LCM⁺00]. It allows instructions stored in memory to be executed but not otherwise manipulated. The processes in the XOM machine are compartmentalized, where each process is allowed a compartment and not allowed to access data out of its compartment.

2.4.3 Software Diversity

Software diversity is comparatively a new idea in software protection. The fundamental idea is similar to the diversity in nature. The diversity in genes makes sure a single virus or disease will not wipe out entire species. The same logic can be applied in terms of resisting the exploitation of software vulnerabilities and program based attack. The

idea of software diversity was documented by Cohen in [Coh93]. The software obfuscation techniques can be used to attain software diversity. Instead of one obfuscation, make many different obfuscation copies of the same program, which are all functionally equivalent but syntactically different.

The advantage of having software diversity is that success in exploiting one instance of a program will not necessarily work on another instance. Various approaches to attain software diversity are discussed in [FAA97]. Rearranging the basic blocks of a program is one of the method explained in [FAA97]. When the basic blocks are shuffled, it will cause the instructions to be stored in different locations. The dependency of the basic blocks is still maintained. This may help in disrupting some injection attacks where viruses insert a single jump instruction to the virus code, stored at the end of the program, and then return control to the original program.

Similarly, rearrangement of independent instructions within a basic block will help in making a diverse software.

Another technique explained in [FAA97] is padding of the stack frame pointer. By padding with random amount of bytes, the return addresses stored in the stack are not anymore in predictable locations. The amount of padding can be dynamically changing or fixed statically for one compilation.

Treating the stack like a heap is another technique to attain software diversity. Every new stack allocation is done at a random location than assigning the next contiguous location. This method will help against stack overflow attacks. Randomizing the locations of global variables and stack offsets for local variables is another technique.

Most programs will be using dynamic libraries and system calls. By varying the names and shuffling the arguments will confuse an attacker. Randomized checks, which perform dynamic array bound check, will help to prevent buffer overflow attacks. It can be specified during compilation time, how often and how many checks each program should do in a run.

2.5 Software Obfuscation

The general obfuscating transformation definition by Collberg et al. [CTL97] is: Let $T : P \rightarrow P'$ be a transformation of a source program P into a target program P' . $T : P \rightarrow P'$ is an obfuscating transformation, if P and P' have the same observable behavior. More precisely, in order for $T : P \rightarrow P'$ to be a legal obfuscating transformation the following condition must hold:

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

In simple terms software obfuscation is the process of transforming a program into a semantically equivalent but harder to understand form. In this section we will discuss about various types of obfuscations and methods and algorithms in each of them.

2.5.1 Data Obfuscation

Hiding data and data structures from the adversary is the objective of data obfuscation. Data obfuscation can be classified into two; based on storage and encoding, Aggregation and re-ordering [CTL97].

Variable splitting [CTL], is a data obfuscation technique that is based on storage and encoding. The basic method is to split a single variable into two or more variables. A combination of split variables will be used in place of the original variable in the program. This will obscure the program data and data structures more and makes it difficult for an attacker to understand the logic. The disadvantage of this method is the time overhead due to the extra computations in the program. Figure 2.2 shows how variable splitting is done.

Figure 2.2: Variable splitting

Converting static variables into procedures [CTL97] is another storage and encoding based data obfuscation. In most programs static data are assigned directly to variables. Instead of assigning raw data directly to static variables, a procedure call is assigned to the static variables. The procedure will return the correct data to the variable. Figure 2.3 shows, how the static definition is replaced by procedure call.

Figure 2.3: Converting static definition to procedural

Array merging, flattening, splitting, and folding [CTL97] [CTL], are aggregation and re-ordering based data obfuscation. In array merging two or more arrays are merged to-

gether into one array and used as a single array in the program. The proper classification of the arrays is thus obscured. Array splitting is the inverse of array merging. In this process a single array is split into two different arrays. How to split the array or how to merge the array can be decided by the obfuscator as long as he makes sure that the semantics of the program is preserved.

Array flattening is a mechanism in which the arrays of higher degrees (2D array, 3D array) are flattened to lower degrees. A two dimensional matrix will be flattened to a vector array in this mechanism. The inverse of this process is array folding, where an array of lower dimension is folded to a higher dimensional array. A vector getting converted to a matrix is an example of array folding.

2.5.2 Design Obfuscation

Design obfuscation design deals with obscuring the design of the software. Many of the higher level languages like java are released in bytecode which are isomorphic to their source code [CTL97]. So, even if other obfuscations like data, layout and control obfuscations are performed, an attacker may understand the class information and the inter dependency between classes. This gives out information regarding the design of the software. Design obfuscation techniques make it harder for an attacker to get this information.

Class coalescing [SNM03a] is a design obfuscation method where the two or more classes from the program is replaced by a single class. Two classes are coalesced by combining the fields and methods of both the classes into a single class. Subsequently all variable declarations using the initial classes have to be changed to declarations using the new class. Figure 2.4 shows an example of class coalescing.

Class splitting [SNM03b] is the inverse process of coalescing. In this method a single class will be split into two different classes. In the general case, splitting a class using an

Figure 2.4: Class coalescing

arbitrary split function is not possible because of dependencies among methods and fields of the original class. So, a conservative dependency analysis has to be done before class splitting. Decisions about the relationship between the new classes and placing method in the new class have to be decided according to the analysis [SNM03a] .

Type hiding [SNM03a] is another design obfuscation method. In this method, the type of the data used during the declarations is hidden using interfaces and macros. In C-program macros are used to alias basic built in data types to user specified names. In

higher programming languages like Java, interfaces are used to hide the type information.

Unlike classes, interfaces do not have instance fields and do not provide implementations for methods that they declare. Interfaces are legitimate types, and as such they can be used in declaration of variables, fields, and methods. Interfaces also can be used in casting operations. In type hiding using interfaces methods, a class has a set of interfaces and each interface will include only a small subset of the public methods of the original class. Variable declarations using the class will be changed with interfaces. Instructions where the variable makes a call to a method in the class, typecasting instructions to typecast the variable to the right interface are added as a part of obfuscation. The example shown in figure 2.5 shows how type hiding is done using interfaces.

Figure 2.5: Type hiding

2.5.3 Control Flow Obfuscation

Many of the obfuscation algorithms designed for software protection focus on control flow obfuscation [WDHK01] [THW09]. Control flow of a program is crucial in understanding the logic of a program. Concealing this information will make it difficult for an adversary to reverse engineer a program successfully. The techniques such as opaque predicates [CTL], double-process obfuscation [GCT05] and signal based obfuscation [PDA07] are examples of control flow obfuscation. This is our major area of focus.

2.5.3.1 Opaque Predicates

Opaque predicates [CTL], is one of the first techniques used for control flow obfuscation. A complex boolean expression or a pointer is used to construct a predicate whose truth value is hard to deduce for an automated de-obfuscator is an opaque predicate. Opaque predicates have a definite truth value. Therefore, if a control flow decision instruction is made by the truth value of an opaque predicate, it will always take one path. Since the truth value is hard to deduce by a de-obfuscator the wrong path will be considered as a valid path while creating control flow graph. Figure 2.1 shows an example of how opaque predicates are used. The opaque predicate is constructed based on the well known fact from number theory that for any integers x, y , $7y^2 - 1 \neq x^2$.

2.5.3.2 Opaque Predicates in Distributed Systems

Obfuscation of distributed systems using opaque predicates were implemented by Majumdar in [MT06]. The system communication available in distributed system has been exploited in this method. The method makes use of the state machine of the system. A doubly circular linked list maintains a number for every systems state. The opaque predicate becomes true when the sum of these numbers add up to a constant. Obfuscation-specific messages are sent and received to stimulate the state machine. The impossibility

Figure 2.6: Opaque predicates

of predicting the message delivery and the order of message transmissions and receptions makes it difficult for an adversary to statically find the truth value of the predicate. The difficulty with this method is that unlike non distributed obfuscation algorithms this cannot be applied all over the software. The time delay caused due to the communications between systems accumulate to an unacceptable level when the method is applied widely throughout the system.

In this technique attacker has a NP-complete complexity to find the value of the opaque predicate statically. An instance of Knapsack problem is used for the creation of opaque predicates in this method. Knapsack problem can be stated as follows:

Given a set $S = \{a_1, a_2, \dots, a_n\}$ of positive integers and a sum $T = \sum_{i=1}^n x_i a_i$ where each $x_i \in \{0, 1\}$, find x_i . This is a decision problem and is in NP-complete. The advantage of the obfuscation method explained in [MT06] is that they map knapsack problem to manufacturing distributed opaque predicate. Every participating process will be initialized with the initial set, S . These are stored in their local data structure, which is a doubly circular linked list. An arbitrary sum formed from the combination of elements in the set S is then used as sum. Then the processes are initialised with values from S which

sums up to T . They transfer these values dynamically through messages and satisfy the opaque predicate during execution.

Let us see it with an illustration. Let the set be,

$$S = \{3, 7, 6, 8, 1, 2, 12, 21\}$$

Let P_1 be the protected process and P_2, P_3 be the participating processes in the opaque predicate creation. Let $T = 17$, be the arbitrary sum. A solution vector corresponding to that sum is

$$x = \{0, 1, 0, 1, 0, 1, 0, 0\}$$

For the illustration with three processes, the list is initialised with three pointers: $p_1.v$ for P_1 , $p_2.v$ for P_2 , and $p_3.v$ for process P_3 . So node initialisation with $p_1.v = 7$ for P_1 , $p_2.v = 8$ for P_2 and $p_3.v = 2$ for P_3 are done. P_1, P_2, P_3 communicates the value with each other during runtime to solve the opaque predicate,

$$\phi : p_1.v + p_2.v + p_3.v = 17$$

2.5.3.3 Double Process Obfuscation

Another control flow obfuscation technique based on two-process scheme is discussed in [GCT05]. The basic idea is to have a small monitor process, M-process, which communicate with the original program, P-process, using inter-process communication methods. The control flow is removed from the P-process and is stored in M-process as a jump table. During program execution P-process will query the M-process for the target address for control flow change. When M-process receives the query, it will send the right target address from the jump table to P-process. The difficulty of this method is to save the M-process from attacks once the algorithm is known to the adversary.

The first step in M-process based obfuscation is to combine the basic blocks into nodes. More than one consecutive basic block is combined together to a node. Control flow of basic blocks within a node is conserved as it is. The control flow between the

nodes gets completely mixed up in the next step of the obfuscation. The nodes are shuffled randomly, so that the control flow between them is not sequential. Once the shuffling is complete, the program control flow is completely lost and hence need to be recovered. The control flow information is stored in a smaller light weight process called M-process. Additional codes are added at the end of each node in P-process. These codes are to query the M-process for the correct target address. During execution, once a node finishes the execution, the query instruction at the end will asks the M-process for the next node to which the control should flow and M-process checks the jump table and replies back the correct address of the node to which the control should flow. Figure 2.7 shows the working of the method.

M-process has all the information regarding the control flow of the program and hence it is important to have some protection mechanism for M-process. M-process is stored in an encrypted form to secure the content of the M-process. M-process is split into different cells and each cell is encrypted using a key. The encryption keys are stored in the cells of M-process. A cell will have the key that is used to encrypt and decrypt the next cell. The algorithm is designed such that only one cell (that is getting executed) will be open at a time during the execution of the program. The first cell of M-process will be in decrypted form when the program starts executing. When the first call o M-process is made from P-process, M-process starts executing. Since the first cell is in decrypted format, M-process starts execution. At the end of first cell, there are instructions to decrypt the cell which is to be executed next and encrypt the first cell. In this way, when the control moves to the next cell, it is in decrypted format. Figure 2.8 explains the M-process execution scheme.

2.5.3.4 Obfuscation by Signals

In [PDA07] Popov et al. introduced obfuscation based on signals. Signals are used for asynchronous communications between processes in operating system. One can have user

Figure 2.7: Double process based obfuscation [GCT05]

An Execution of the Protected Program, with the Control Flow of
 Start → Cell 0 → Cell j → Cell k → → Cell i → End

Figure 2.8: M-process execution [GCT05]

defined signal handlers. During the execution of a program, when an exception happens a trap signal is raised and the trap handler of the operating system kernel is activated. The kernel gives the control to user defined signal handler which gets executed. After the execution, the control is given back to the kernel which in turns returns the control to the point of original execution. Figure 2.9 depicts the signal handling process.

This obfuscation takes leverage on the fact that the control flow of the program is transferred from the normal program execution to the signal handler. The basis of every control flow obfuscation algorithm is to replace the trivial control flow obfuscation mechanism with non trivial mechanisms.

In the signals based obfuscation method, the control flow instructions are replaced by

Figure 2.9: Signals in operating system

trap instructions like divide by zero instructions. Trap instruction will raise an exception only during the program execution. When statically analysed these instructions will look like normal instructions. This will fool a standard static disassembler. So, if a *jmp label-l* instruction is replaced by a trap instruction then the disassembler will assume the control flows directly to the next address instead of *label-l*.

However, changing the control flow instruction to trap instruction has changed the meaning of the program and if the program is executed it will crash. User defined signal handler is introduced in the algorithm here. The obfuscation algorithm installs a user defined signal handler installed to handle these special trap instructions. Setup code is written ahead of the trap instruction which stores the original control flow address in the table and the table is indexed by the hash value of the trap instruction address. During execution when the trap instruction is executed instead of an original control flow instruction. The control then goes to the user defined signal handler through the kernel. The user defined signal handler overwrites the kernel restore functions return address field with user restore functions address. So, after the execution of user defined signal handler and the kernel restore function, the user restore function gets activated. This will calculate the hash value of the original target address and searches the table.

The address is pushed into the stack and is returned to that address from user defined function. Figure 2.10 shows an example of signal based obfuscation.

Figure 2.10: An example of signal based obfuscation

2.5.3.5 Dynamic Code Mutation

Madou et al. discusses about attaining control flow obfuscation through dynamic code mutation in [MAM⁺06]. The basic technique used is to dynamically mutate the program as it executes. As a result the same memory area will be occupied by different code sequences.

This method is based on two basic concepts: an editing engine and edit scripts. The editing engine is the obfuscation engine which generates the code dynamically and edit scripts are scripts which help the edit engine generate the right code.

When a procedure, say f , is decided to be obfuscated by this algorithm then it means the procedure f is to be generated at runtime. The first step of the obfuscation is to statically replace the procedure f with a template. The template is a copy of the procedure f where some of the instructions are removed. The instructions that are

removed are replaced by a stub to the editing engine. The inputs to the editing engine from the stub are the location of edit scripts and the entry point of the procedure. The editing engine will now reconstruct the procedure according to the information in edit scripts. Once the reconstruction is over then it jumps to the entry point of the procedure f .

All the information required to reconstruct the procedure from the template will be there in the edit script. The necessary information are the location of the template, a specification of bytes to be changed and the new byte values that replaces the old template.

[MAM⁺06] discusses two methods of implementation: one pass mutations and clustered mutations. In one pass mutation, each procedure has a separate template and hence each procedure is obscured separately. During program execution, the changes are nullified by a single round of editing for each procedure. The stub is placed at the beginning of the procedure. During the first execution of a procedure the stub is invoked and the edit engine runs, replacing the template with the procedure. The stub is also replaced by the editing engine.

In a clustered mutation scheme a group of procedures are clustered together to have a single template. The procedures will be selected such that the instruction sequences are sufficiently similar. The procedures in the same cluster will thus be mapped to the same memory area, the cluster template. A call to the procedure is replaced by a stub that invokes the editing engine, which reconstructs the process. Figure 2.11 shows the process of clustered mutation scheme.

2.5.3.6 Self Modifying Code based Obfuscation

Another obfuscation algorithm using self modifying code was proposed by Lian et al. in [SE10]. The basic idea of this method is that the program dynamically modifies itself

Figure 2.11: Clustered dynamic mutation

during program execution and there are no extra modules unlike the previous methods. The self modification engine is fused into the program and is a part of the program.

This method concentrates more to increase the disassembly error of automated disassemblers. The possibility of adding partial junk bytes in the presence of unconditional jump instructions is discussed in [LD03a]. When you add partial junk bytes the disassembler associates the junk byte with neighbouring legal bytes to generate assembly instructions. The instructions thus generated are not valid. To achieve this, the disassembler should be fooled to believe that the junk byte is a valid byte. The location just after an unconditional jump instruction is such a location. So, if more unconditional jump instructions are available, the more the technique confuses the attacker.

In this method basic blocks are split into two and *jmp* (unconditional jump) instruction is inserted at the end of first part, pointing to the second part. Thus, the semantics of the program is conserved. But the addition of unconditional jump instruction gives scope for inserting junk bytes into the program code. This will increase disassembly errors while an automated disassemble is used.

The *jmp* instructions are camouflaged with normal instructions like *mov* instruction. This will give control flow obfuscation. Removing jump instructions will affect the

semantics. Hence, the camouflaged instructions are reconstructed back to jump instructions during runtime by modification instructions. Modification instructions are added above the camouflaged instructions. This makes sure that the modification instructions are executed before the camouflaged instruction. Figure 2.12 shows the working of self modifying based obfuscation.

Figure 2.12: Self modifying code based obfuscation

2.5.4 Evaluation Metrics for Obfuscation

Software obfuscation can only harden the process of reverse engineering and cannot prevent it [BGI⁺12]. In this section we will discuss about the different evaluation metrics used to measure the obscurity achieved by an obfuscation algorithm. The strength of the obfuscation algorithm to resist automated attacks, the ability of an obfuscated program to pass as a normal program and the overheads due to obfuscation are discussed in this section.

Recursive traversal is the main technique used for reverse engineering tools to translate binary code to assembly level representation. Our algorithms are designed to beat

the technique of recursive traversal to achieve control flow obfuscation. Hence the evaluation metrics find the obscurity achieved when a reverse engineering tool using recursive traversal is used to disassemble an obfuscated binary program. We use IDAPRo [Hex] for analysis as it is one of the best reverse engineering tool based on recursive traversal.

2.5.4.1 Potency Measure

Potency measures the performance of the obfuscation algorithms [CTL97]. It measures how well the obfuscation performed against automatic reverse engineering tools. The two methods used for measuring potency are instruction disassembly error [PDA07] and control flow error [SE10].

Instruction Disassembly Error

We evaluate the instruction disassembly error with confusion factor. Confusion factor is the fraction of instruction address that the disassembler fails to identify [SE10]. T_{total} is the total number of actual instruction addresses before obfuscation and T_{disasm} is the total number of instruction addresses properly recognized by the disassembler, then the confusion factor is defined by the following,

$$CF_{instr} = \frac{|T_{total} - T_{disasm}|}{T_{total}} \quad (\text{Eq. 2.1})$$

Control Flow Error

The number of conditional and unconditional jump instructions in the program before and after the obfuscation is calculated. If CFG_{before} is the total number of conditional and unconditional jump instructions in the program and CFG_{after} is the total number of jump instructions in the obfuscated program. CFG_{cfg} is the confusion factor in the control flow of the program,

$$CFG_{cfg} = \frac{|CFG_{before} - CFG_{after}|}{CFG_{before}} \quad (\text{Eq. 2.2})$$

The ratio gives the control flow confusion caused by the obfuscation. The error in identifying the control flow instructions in the program gives us an account of the control flow obfuscation attained by the algorithm.

2.5.4.2 Stealth Measure

The stealth of obfuscation measures the difficulty to identify whether a binary is obfuscated or not [CTL97]. An obfuscated program is considered stealthy if it is hard to distinguish it from a normal binary file.

The method we use for stealth measure is by calculating the Mahalanobis [LKK10] distance between the obfuscated program and normal binary samples as discussed in [LKK10].

2.5.4.3 Cost Measure

In this section we will discuss how overheads caused by the obfuscation to the program. The two overheads are, increase in the space complexity and time complexity of the program [CTL].

$Space_{ovh}$ defines the variation in the size of the program due to obfuscation.

$$Space_{ovh} = \frac{S_{code1} + S_{data1}}{S_{code0} + S_{data0}} \quad (\text{Eq. 2.3})$$

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

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

$$Time_{ovh} = \frac{T_{obf}}{T_{ori}} \quad (\text{Eq. 2.4})$$

T_{ori} refers to the execution time of the original file and T_{obf} refers to that of the obfuscated code under the same input.

2.5.4.4 Time complexity due to obfuscation

If the time complexity of a program is $T(n)$ it means there are instructions in the program which executed in the order of $T(n)$. In the worst case all the instructions in the program are executed in the order of $T(n)$. So if there are a instructions in the program, then the time to execute the program is $aT(n)$.

The obfuscation algorithms in the thesis, introduces new instructions to the program. The introduced instructions are straight line code which gets executed one after another. The instructions by itself is of the time complexity $O(1)$.

However, when the instructions are introduced into a program with a time complexity $T(n)$, in the worst case all the newly added instructions gets executed in the order of $T(n)$. If the program has a instructions and x obfuscation instructions are added to the program, then the time to execute the program will be $(a+x)T(n)$.

If $x=a$, then the time complexity is $2aT(n)$. So, the time complexity increases 2 times that of the original time complexity.

2.5.4.5 Resilience Measure

Resilience is the measure of the amount of time required for creating an automated deobfuscator [RL07] [SSM05] that will reduce the potency of the obfuscation [CTL97]. Resilience also measures the time and space complexity of such an automated deobfuscator.

2.6 Preliminaries

2.6.1 Analysis of Binary Program

Reverse engineering of binary can be classified broadly into two: static analysis based [CJ03] [HCS09], and dynamic analysis based [NS]. In a static analysis based scheme, the binary program is analyzed and disassembled statically. It tries to create the assembly

representation of the binary program without executing the program. There are various tools, open source [[ALD](#)] [[Oll](#)] [[GNU](#)] and proprietary [[Hex](#)] [[Cor](#)], which helps in the process.

In a dynamic scheme, the execution of the program is monitored and analyzed to get the context information during runtime. As a result it only covers only those regions of the program which are executed, and there can be infinite number of execution paths for a program.

Static analysis is mostly employed because it gives a complete overview of the program than a single execution path given by dynamic analysis. Research works usually measure the potency of their algorithm against static analysis. We have measured the efficiency of our algorithm against static and dynamic analysis.

2.6.2 Disassembly of Binary Program

The first step of reverse engineering of a binary program is disassembly. It is the process of creating the assembly representation of the binary program. Linear sweep and recursive travel are the most widely used disassembly algorithms [[EC05](#)]. Linear sweep begins disassembly at the programs first executable byte, and sweeps through the program, disassembling instructions sequentially one after the other. But this method has a weakness, that it misinterprets data bytes as instructions if the data is embedded between instructions in the code section. The popular tools using this method are GNU objdump [[GNU](#)] and Microsofts DumpBin [[Cor](#)].

The problem in linear sweep is solved by recursive traversal algorithm. Recursive traversal takes control flow of the program into account. Control flow of a program is the order in which the basic blocks of a program are executed. In recursive traversal, it follows the execution flow recursively and disassembles the instructions. However, the assumption that recursive traversal can precisely find the control transfer location may

not hold in the case of conditional jumps and calls. Disassemblers implementing this algorithm are the IDAPro [Hex] and OllyDbg [Oil]. All these disassemblers are used for static disassembly of binary programs.

Assembly language debugger [ALD], *ald*, can be used for dynamic disassembly. One can execute the program and disassemble instruction by instruction while execution using *ald*.

2.6.3 Junk Byte Insertion

Junk byte insertion [LD03a] is a method used to confuse the automated disassembler and forcing them to give wrong disassembly results. The idea is to add junk bytes in areas where disassembler expects code. The junk bytes are partial instructions added at locations which will not change the semantics of the program. For instance, junk bytes can be inserted into a basic block immediately after a block ending with an unconditional jump. Junk bytes are added to the beginning of this block, which is in fact unreachable code during runtime. But a static disassembler sees it as a valid instruction and tries to disassemble the partial instruction bytes. Since the instructions are partial, the disassembler joins it with the next valid instruction bytes, to create a valid assembly instruction. This corrupts the valid instruction in the basic block.

2.6.4 Self Modifying Program

Self modifying program is one which modifies itself while executing. This method is used in different binary obfuscation techniques in different form. The basic idea of this method is that, parts of the programs are removed or replaced by other instructions thus statically the program looks different.

During runtime the program is transformed back to its original form. Different methods are adopted to achieve this as presented in [KMNM03] [SE10] [MAM⁺06]. The basis

of all the methods is to add extra code modules to the program which know exactly which area of the program is to be modified and when to be modified.

The advantage of using self modifying programs is that it obscures the programs really well and makes it difficult for the static disassemblers to correctly disassemble the program. Statically program looks completely different and it gets fixed dynamically through self modifying code. Self modifying code can also have self obfuscation code which obfuscates the real code, dynamically changed after its execution. So, the period in which the code is in its true form is during its execution. So, even if an adversary decides to run the program and break at some point and dynamically disassemble the program, his/her chance to get the program in its true form is low.

2.7 Conclusion

In this chapter, we discussed various software protection techniques at each level of software lifecycle from software development to software maintenance. We discussed about the different types of software obfuscation and also saw different obfuscation algorithms in each of them. Preliminary knowledge in techniques used in software obfuscation were also discussed in this chapter.

Chapter 3

Software Obfuscation by Hiding Control Flow Information in the Stack

In this chapter we propose a new software protection algorithm based on self modifying code and using stack to conceal the control flow information. The performance of this algorithm against automated disassembly attacks is better than the state-of-the-art algorithms [PDA07] [SE10]. The time and space overhead due to the obfuscation are comparable with other algorithms.

3.1 Motivation

In the control flow obfuscation algorithms in literature, the basic idea is to remove the direct reference of control flow instructions. These control transfers are then achieved using alternative techniques like signals [PDA07], edit engine [MAM⁺06] or self modifying code [SE10]. One disadvantage of these methods is that the removed control transfer address locations are stored in the program code area. The reconstruction module will be having the correct addresses to which the control transfer has to be made. Another disadvantage is the addition of an extra module in the program as a part of the obfuscation. The possibilities of using the data area to conceal the target addresses are not

explored. In this chapter we discuss a method which uses data area to hide control flow instructions. Also, our method does not add an extra module to fix the obfuscations dynamically. The dynamic reconstruction instructions are instead a part of the program itself.

Further, many of the obfuscation algorithms are designed to work well against static analysis. In most cases dynamic attacks are not discussed and the performance of algorithms against them is not measured. It is important to consider the dynamic nature of the attacks and to design protection algorithms against those. We have discussed the performance of our algorithm against dynamic attacks in this chapter.

3.2 Threat Model

It is necessary to understand the hostile environment in which the software is going to get exposed to before deciding on the protection scheme. Our assumption is that the attacker is trying to disassemble the binary program and trying to get the assembly level control flow graph representation. The threat posed by the attacker is modeled by considering three factors:

- Origin of the threat
- Tools available for the adversary
- Knowledge of the adversary

Origin of threat is classified into three categories, category 1, category 2 and category 3 as explained in [PDA07]. In category 1, the malicious threat is originated outside the computer. The adversary will be working from a remote computer breaching the communication access controls. In category 2, the attack happens while the software is running on the system. The attacker was successful in planting malicious code to the

system which tries to manipulate the software during its execution. In category 3, the attacker has complete control over the system. The attacker can analyze the program, modify it and execute it. This attack has no security perimeters as the owner of the system is the attacker. We are interested in category 3 attack model. In our threat model we assume the origin of the threat as category 3.

The tools available for the attacker define the strength of the attack. There are various tools for disassembling binary programs [GNU] [Cor] [Hex]. Some are based on linear sweep approach which begins disassembly at the programs first executable byte, and sweeps through the program, disassembling instructions sequentially one after other. A better disassembly technique is recursive traversal which takes control flow of the program into account while reverse engineering. We assume the attacker uses a recursive traversal based commercial disassembler, IDAPro [Hex].

We also assume that the attacker has the knowledge that the program is obfuscated. We do not assume that the attacker knows the exact steps of the obfuscation algorithm. To conclude about the threat model, we assume an attacker with complete access to the computer and binary program, with the knowledge that the program is obfuscated, is trying to disassemble the binary program to assembly level representation using IDAPro to reverse engineer and analyse the program.

3.3 Proposed Method

A program consists of code area and data area. Different data areas are global, local and dynamic. Stack is an example for local data area and heap for dynamic. Our method is basically built on the fact that reverse engineering tools and methods consider data area and code area separately. Reverse engineers and reverse engineering tools try to extract programming information from the code segments of the software and extracts data values and information about the data structures from the data segments and symbol tables.

The basic idea of our obfuscation is to hide the code information like jump instructions, in the data area, stack, with other data elements thus obscuring the program code. The process of hiding code information in data area is done at the obfuscation time. The information is stored in stack and hence it looks like ordinary variables defined in the function. It is harder for an attacker to distinguish this from ordinary variables by just analyzing the stack. Removing instructions from the code area or camouflaging it with other instructions makes the program semantically different. The code information stored in the data area is used to reconstruct the original code at runtime and thereby the execution of the program is semantically equivalent. This is achieved by inserting reconstruction instructions just above the original location. This will result in reconstructing the original instruction at runtime. We further explain our algorithm in detail.

3.3.1 Offline Obfuscation

This is the first phase of our obfuscation algorithm. The binary program is converted to its equivalent assembly program using PLTO (Pentium Link Time Optimizer) [SDAL01]. It is then analyzed to find suitable instructions to be obfuscated. Once the obfuscation is done, the assembly program is assembled back to binary.

3.3.1.1 Selecting Instruction to be Obfuscated

The first step of the algorithm is to identify which all instructions have to be camouflaged. The trivial method is randomly picking instructions from the code area. But, in our method the jump instructions are chosen to be camouflaged for the following reasons.

Jump instructions decide the control flow of a procedure in the program. By obscuring the jump instructions in the procedure we are thus obfuscating the control flow of the program. Instructions which give information about the control flow of the program will help the adversary to easily understand the logic of the program.

Another motivation for considering jump instructions to be camouflaged is the scope it provides for inserting junk bytes in the program. Camouflaging jump instructions obscures control flow of the program. This will lead in confusing the disassembly tool to assume wrong control flow to the program and makes it possible to add junk bytes between code blocks which are unreachable. This will increase the errors while an adversary tries to reverse engineer the binary program.

3.3.1.2 Storing Target Address in the Stack

With the instructions to be camouflaged are known, the space required in the stack to store the target addresses of camouflaged instruction can also be calculated. In the method proposed, for each instruction to be camouflaged in a procedure space is allocated in the stack. The counts of instructions in the function which are going to be camouflaged are calculated and then the stack is expanded accordingly.

The expansion of the stack is possible with a small tweak in the assembly program. In the calling convention of the ELF (Extended Linker Format) programs in x86 platforms, the stack allocation for a function is done by the function itself. All the functions start with the following instructions:

```
push ebp  
mov ebp, esp  
sub esp, 8
```

Once the function is called the base pointer of the caller function is pushed onto the stack. Then the current stack pointer is stored as the new base pointer (for the called function). The first two assembly instructions in the code segment are essentially doing that. The third instruction is where the allocation of the stack for the particular function happens. The size of the stack needed by the function in this particular case is 8 bytes. By modifying the value in the third instruction, the size of the stack for that particular function can be changed.

Once, the instructions that are going to be obfuscated and their count are known, the stack is expanded accordingly as mentioned in the previous paragraph.

Since we know that we are moving *jmp* instructions, the target address to which jump happens constitutes the code information. This target address is what we store in the data area.

Selecting stack area to store the code information has an advantage over global data area. The code information in stack area is stored in a way similar to that of local variable definition. When instructions use this information to find the real target address of a jump, it just appears like instructions manipulating ordinary variables. Loading the value from the variable to a register and analyzing the value. The variables of that function are used only by the instructions of that function. On the contrary, if global data area was used to store the code information, then the code information is stored in the global data area and each local function will use only those variables which are used to store the control flow information of the particular function. An adversary can easily notice that, certain global variables are exclusively used by certain local functions, which information can then be used by the adversary.

Figure 3.1 shows how the *jmp* instructions target address is stored in the stack area. The target address *xxxx* of the *jmp* instruction in the first block is stored in a stack variable.

3.3.1.3 Obfuscating the *jmp* Instructions

The *jmp* instruction is ready to be obfuscated as the target address of the *jmp* has already been stored in the stack. The *jmp* instructions are replaced with another instruction instead of removing. The *jmp* instructions are replaced by the following instruction,

mov eax, 0

The replacement of *jmp* instruction with *mov* results in the loss of control flow information. The new instruction, *mov*, is an ordinary instruction and does not have a say in

Figure 3.1: Storing code information in stack

the control flow of the program. When an automated disassembler tries to disassemble the program, it assumes the control flows just to the next address location after *mov*.

We decided on the instruction *mov* to be used to replace *jmp* instructions owing to the fact that it is the most used instruction in a program. It is possible to use other instructions instead of *mov* to camouflage the *jmp* instructions. The logic remains the same. Randomizing the selection of instruction to be used to replace *jmp* instruction will increase the challenge posed by the method to an adversary.

3.3.2 Runtime De-Obfuscation

Camouflaging the instructions in the program as explained in the previous section changes the semantics of the program. Running this program in this form will give erroneous results and most probably crash the program. And hence, the program has to be changed back to its original form before it gets executed. In our method we do this dynamically at runtime with the help of self modifying code.

Reconstruction instructions which reconstruct *jmp* instruction at runtime are inserted in a block that precedes the *jmp* instruction. The block should be a dominator block, which means it should precedes the *jmp* instruction in all execution paths. The insertion of reconstruction instructions are shown in Figure 3.2.

The first step is to change the opcode of *mov* instruction to that of *jmp* instruction. The opcode of *jmp* instruction is 0xE9 and that of *mov* instruction 0xB8. We add an instruction to XOR the address location of *mov* instruction with 0x00000051. This changes the instruction to *jmp* offset 0. Now the next step is to add the address offset stored in the data area to the instruction. We add an instruction to add the value in the local variable to the instruction address. Now the exact *jmp* instruction is created at the address location of *mov* instruction.

In Figure 3.2, the camouflaged *jmp* instruction is at address location A1 in basic block B1. The *jmp* instruction is camouflaged into *mov* instruction and the reconstruction instructions are added before the camouflaged instruction.

3.3.3 Runtime Re-Obfuscation

With the reconstruction instructions in place, the program semantics are restored and program works perfectly well. Now, the instructions which are obfuscated are restored and in its original form. An adversary tracking the image of the program at regular intervals will be able to find the de-obfuscated instructions. A core dump of the image of the program will give the instructions in its true form if it is done after the re-construction operations.

A method to address this problem is by re-obfuscating the instruction at runtime after its execution. This is achieved by adding extra re-obfuscation instructions in the succeeding blocks to re-obfuscate *jmp* instruction back to *mov*. Note that, the re-obfuscation instruction should be inserted in all the successor blocks as the execution path is chosen dynamically at runtime. Re-obfuscation is done by XOR-ing the *jmp* instruction with 0x00000051 to get the instruction:

mov eax, 0

Figure 3.2: Obfuscation of *jmp* instructions

According to the control flow of the example in Figure 3.2, the basic block B3 follows after the execution of the `jmp` instruction. The re-obfuscation instructions for the program are hence added in the beginning of the basic block B3.

3.3.4 Junk Bytes Insertion

The replacement of `jmp` instruction with `mov` instruction opens space for inserting junk bytes into the code section. Insertions of junk bytes introduce more confusion to the disassembler [LD03a]. This will result in making more instruction disassembly error during disassembly process.

Since the `jmp` instruction is replaced by `mov` instruction the disassembler will think that control flows directly after the `mov` instruction to the next instruction. This lets us introduce junk bytes after the `mov` instruction. Partial junk bytes are introduced as discussed in [LD03a] to achieve maximum confusion.

Another effect of insertion of junk bytes is that there can be wrong `jmp` instructions in the junk byte region, which will confuse the disassembler further.

Figure 3.3, shows how the junk bytes are introduced in the program. The existence of junk bytes corrupts the original code in the program too, since partial junk bytes of an instruction are added.

3.3.5 Conditional Instructions

Conditional jump instructions like, `jle` (jump if less than or equal), `jge` (jump if greater than or equal), `jz` (jump if zero), `jg` (jump is greater than) etc., also adds to the control flow of the procedures in a program. Obfuscation of these instructions can be done similar to the unconditional jump instructions. These conditional jump instructions can be camouflaged using other ordinary instructions and the target address can be stored in the stack. The problem is that the insertion of junk bytes, which is responsible for confusing the disassembler and increasing the instruction disassembly error can't be done with conditional instructions.

The basic reason that junk byte insertion not being possible with conditional instruction is that the instruction followed by the conditional jump instruction is a valid instruction point. Adding junk bytes there will corrupt the program.

Figure 3.3: Junk byte insertion

To take care of this condition our method deals with conditional jumps in a different manner, so as to get better obfuscation.

In this method a junk byte is added just above the conditional jump instruction. This junk byte should be a partial byte of an instruction as explained in [LD03a]. In the example is shown in the Figure 3.4, a partial junk byte of 10 is added to the program. The junk byte is partial because it is part of an instruction but the byte itself is not an instruction. So when a disassembler encounters such a byte it will have to join the instructions following the byte to create an assembly instruction. In this example, the disassembler will join *10* and *EB* to create the instruction *adc bh, cl*.

The semantics of the program will be affected by this insertion of the junk byte and the junk byte should not get executed. Re-construction instructions are added just like in the case of unconditional jump in this case. Instructions are added before the junk

byte to convert *10* to *90*. The value *90* is the opcode for *nop* instruction. Thus the semantics of the program remains the same during runtime.

Machine code	Assembly	Machine code	Reverse Engineered Assembly
10	<i>Inserted partial junk byte</i>	10 EB	<i>adc bl, ch</i>
EB 90	<i>jmp offset 90h</i>	90	<i>nop</i>

During obfuscation

During disassembly

Figure 3.4: Partial junk insertion example

Similar to the case of unconditional instructions, re-obfuscation instructions are added in all the successor blocks. In this case, the re-obfuscation instructions obfuscate the *nop* instruction back to the junk byte.

3.3.6 Randomization to Improve Obfuscation

To improve the performance of the algorithm against an intelligent adversary, randomization can be used. XOR instruction is used in the re-construction and re-obfuscation process in the method. Other logical and arithmetic instructions can be used to achieve the same result. An adversary who is trying to find the obfuscated points by filtering based on a specific instruction, such as XOR in this case, will not work if randomly chosen instructions are used to re-construct and re-obfuscate the program.

An equivalent operation which gives the effect of the XOR (*xor (A1), 0x00000051*) instruction during dynamic reconstruction of jmp instruction, is the following,

And (A1), 0x00000000

Add (A1), 0x000000E9

The *mov* instruction (0x000000B8) at the address location A1 will be converted to 0 by the *and* instruction and then adding 0x000000E9 at the address location will reconstruct back the jmp instruction. Using an OR instruction instead of *add* will also give the same result.

And (A1), 0x00000000

Or (A1), 0x000000E9

Similarly during junk byte insertion, it is better to select a junk byte at random from the set of junk bytes rather than using the same junk byte for all the insertions.

Another randomization that can be done is the selection of the basic blocks in which the re-construction and re-obfuscation instructions are inserted. The only condition for the block in which re-construction instructions are added is that the block should be a dominator [ALSU06] to the block containing the obfuscated instruction. Similarly the re-obfuscation instruction should be in a post-dominator [ALSU06] block. A basic block A is a dominator to basic block B if in all execution paths A gets executed before B . Similarly, a basic block A is a post-dominator to basic block B if in all execution paths A gets executed after B .

3.3.7 Example Code Snippet from IDAPro

Fig. 3.5 shows how our obfuscation obscures control flow of the program. The initial configuration of the blocks is in such a way that block-1 and block-2 have an edge to block-3. The transformation of the jump instruction in block-1 to move instruction by obfuscation makes the reverse engineering tool assume that the control flow of the program flows from block-1 to block-2.

3.4 Implementation

The proposed obfuscation is carried out at link time of the compilation process. The implementation expects a binary program as input. The input program is then reverse engineered to binary and the obfuscation instructions are added to the program. The obfuscated assembly is compiled and the obfuscated binary program is given out as output. The development platform used is GNU Linux operating system and the input binary

Figure 3.5: Reverse engineered code from IDAPro

files are expected to be in the extended linker format (ELF). For the implementation of our algorithm at link time, PLTO, Pentium Link Time Optimizer [SDAL01] was used.

The input binary program is fed to PLTO which creates the control flow graph of the program. The control flow graph thus generated is scanned through to find possible candidate instructions to be obfuscated. Each function of the program is scanned through block by block to find the unconditional jump instructions.

Once the count of the jump instructions that are going to be obfuscated is finalized then the size of the stack is expanded. The local variables of each function are stored in the stack. The activation record for each function will be of constant size defined in the beginning of a function. It has the space required for storing local variables, parameters and return value. Every time a function is called this constant space in stack is allotted for the function. Since our method stores the code information as variables in the stack,

this stack size has to be expanded. The code in the function which defines the stack size required is modified according to the requirement to expand the stack size. With this modification, when the function is called it pushes the stack pointer further and thus incorporating the space for the new local variables used to store the control flow information.

For each function in the program, obfuscation is done in three rounds. In the first round all the unconditional jumps are handled. Junk byte insertions at locations after unconditional jumps are done in the second round. Conditional jumps are handled in the third round of the algorithm. The process repeats for all the functions.

The exact sequence of implementation in the first round is as follows. The target address of each unconditional jump instruction is extracted from the instruction and is stored in the local variable. The `jmp` instruction is then replaced with `mov` instruction. The basic blocks in which the re-construction instructions and re-obfuscation instructions have to be inserted are calculated. Re-construction instructions and re-obfuscation, which use the variable where the address is stored, are inserted in the respective basic blocks. The successor block of the `jmp` instruction is flagged as candidate block for junk byte insertion.

In the second round, all the basic blocks which are flagged as candidate blocks for junk byte insertion are visited and from the set of junk bytes, which are partial instructions, randomly chosen junk byte is added to the beginning of the basic block.

The third round in the implementation is similar to the first round. Each basic block with unconditional jump instructions are visited. The junk byte to be inserted is randomly chosen and is stored in the variable in the stack to be used for re-construction and re-obfuscation instructions. The junk byte is then inserted just above the unconditional jump instruction. The basic blocks in which the re-construction instructions and re-obfuscation instructions have to be inserted are calculated. Instructions which convert the junk bytes to `nop` instructions are inserted in the basic block for re-construction

instructions. The instructions for converting *nop* back to the junk byte are inserted in the basic blocks for re-obfuscation instructions.

The whole program, which is in the intermediate control flow representation in the PLTO framework is then re-compiled to binary executable.

3.5 Performance Evaluation

In this section we evaluate the performance of the proposed algorithm against static and dynamic analysis. The three performance measures used are potency, cost and stealth as explained in [CTL]. Potency is a measure of the strength of the obfuscation algorithm. It measures how well the obfuscation performs under automatic de-obfuscators. Instruction disassembly error and the control flow errors in the de-obfuscated assembly program analyzed statically gives the potency measure against static analysis. The percentage of original instructions that were not obtained by static analysis but gained through dynamic analysis, gives the potency of the algorithm against dynamic analysis. The cost of obfuscation can be measured in terms of program size overhead and execution time overhead due to obfuscation. Stealth of obfuscation measures the difficulty to identify whether the program is obfuscated or not. We measure the stealth of the obfuscation as Mahalanobis distance between the original program and the obfuscated program [LKK10].

We evaluated the efficacy of the obfuscation with programs from the *SPECint-2006* benchmark suites. The evaluation results are similar when applied on other C programs. Our evaluation platform is 2.6GHz Pentium system with a 2 GB internal main memory. The operating system used is *Ubuntu* distribution of GNU Linux. The compiler used is gcc version 3.4 at optimization level -O3. The disassembly results from IDAPro [Hex], version 5.2.0.911, is used for the performance evaluation.

3.5.1 Potency Against Static Analysis

Potency measures the performance of the obfuscation algorithms. It measures how well the obfuscation performed against automatic reverse engineering tools. For statically analyzing the binary program we used IDAPro [Hex], a professional reverse engineering tool. The binary program is disassembled using IDAPro. The program thus disassembled is compared with the original program to find the instruction disassembly error and control flow error caused by our obfuscation.

3.5.1.1 Instruction Disassembly Errors

We evaluate the instruction disassembly error with confusion factor. Confusion factor is the fraction of instruction address that the disassembler fails to identify [SE10]. T_{total} is the total number of actual instruction addresses before obfuscation and T_{disasm} is the total number of instruction addresses properly recognized by the disassembler, then the confusion factor is defined by the following,

$$CF_{instr} = \frac{|T_{total} - T_{disasm}|}{T_{total}} \quad (\text{Eq. 3.1})$$

Table 3.1 shows the instruction confusion factor of the *SPECint-2006* programs. The average instruction disassembly error of the *SPECint-2006* programs is 79.78 %. This means that the disassembler succeeds in recovering only 20.22% of the instructions properly, on an average

3.5.1.2 Control Flow Disassembly Errors

We calculated the number of conditional and unconditional jump instructions in the program before and after the obfuscation. If CFG_{before} is the total number of conditional and unconditional jump instructions in the program and CFG_{after} is the total number

Table 3.1: Instruction disassembly error

Programs	T_{total}	$T_{total} - T_{disasm}$	CF_{instr}
bzip2	980149	787157	80.03 %
hmmer	1608118	1326325	82.47 %
lbm	870411	640529	73.58 %
mcf	901763	681301	75.52 %
sjeng	1105023	964533	87.28 %
Mean			79.78 %

Table 3.2: Control flow errors

Programs	T_{total}	$T_{total} - T_{disasm}$	CF_{instr}
bzip2	16883	9516	56.36 %
hmmer	24183	17729	73.31 %
lbm	14766	7660	51.87 %
Mcf	15217	8095	53.19 %
sjeng	19281	12421	64.42 %
Mean			61.35 %

of jump instructions in the obfuscated program. CFG_{cfg} is the confusion factor in the control flow of the program,

$$CFG_{cfg} = \frac{|CFG_{before} - CFG_{after}|}{CFG_{before}} \quad (\text{Eq. 3.2})$$

The ratio gives the control flow confusion caused by the obfuscation.

The error in identifying the control flow instructions in the program gives us an account of the control flow obfuscation attained by the algorithm. Table 3.2 shows the number of control flow instructions in the original program and the obfuscated program and their ratio.

3.5.2 Potency Against Dynamic Analysis

In this section we will discuss one of the dynamic analyses that can be done against our obfuscation technique and how effective our method works against it.

One of the methods of dynamic analysis of any program is executing the program and breaking after each instruction execution and tracking the instruction. This will definitely give the correct instructions, in its true form to the adversary but only those instructions in that execution path. Just disassembling one execution path of the program is the major hurdle of this method of dynamic analysis. There can be infinite number of execution paths in a computer program due to the loops and to track all the execution paths is really difficult.

To analyze the program containing self modifying code, another approach is to core dump the image of the program while executing. This approach is more realistic than the earlier one. Adversary statically disassembles the program first and then run the program. The program is stopped in between and the image of the program is dumped. The dynamic binary image of the program is disassembled by a static disassembler. The statically disassembled binary and the disassembly of dynamic binary image are then compared to see if there are any differences. If the program has self modified during the execution of the program, those modified regions will be exposed in the dynamic binary image. An adversary can easily point out those changed areas in the code by simply comparing it with the statically disassembled program.

We chose the latter method for dynamic analysis as our obfuscation method has self modifying code and it is more realistic. We implemented the dynamic analyzer using python and *ald*- assembly language debugger [[ALD](#)]. Python was the framework used to call *ald* and give instructions to the *ald* tool. The binary program is disassembled using *ald* before it starts executing and is stored in a file. The section addresses of the program are available through *ald's file secinfo* command. The address location of code area is

thus known to the automated software. Python randomly picks an address location from this area. It then gives *ald* to run through the binary program and break at the randomly chosen address location. Once the *ald* successfully breaks, the image of the program in *ald* is disassembled again to get the new assembly code. This is stored in another file and is compared with the first file for differences.

We tried this method in *SPECint-2006* programs starting from 5000 breaks to 100,000 breaks. When we say 5000 breaks, it means that the python program forces the *ald* to break at 5000 random locations while running and disassembles the program after each break and compares it with the statically disassembled program. So, if an obfuscated jump instruction is revealed during this process then it is added information to the adversary along with the static analysis information which he has.

The dynamic analysis results of *SPECint-2006* programs are shown in Figure 3.6. Number of breaks during the dynamic analysis and the number of jump instructions revealed forms the axis of the graph.

The graph give the absolute value of the number of the jump instructions revealed during dynamic analysis of the *SPECint 2006* programs. For example 52 obfuscated jump instructions of *sjeng* were revealed by dynamic analysis with 100,000 breaks.

Table 3.3, gives a relative perspective of dynamic analysis unlike the absolute information in Figure 3.6. It is clear from Figure 3.6 that the number of jump instructions revealed increases as the number of breaks used in the dynamic analysis increases. Table III, shows the percentage of obfuscated jump instructions revealed by dynamic analysis with 100,000 breaks i.e., the number of jump instructions that are revealed out of the total number of jump instructions that are obfuscated.

The percentage of obfuscated jump instructions by dynamic analysis is less than 1%. The dynamic analysis with 100,000 breaks yields only small additional information than static analysis.

Figure 3.6: Jump instructions revealed during dynamic analysis

Table 3.3: Percentage of jump instructions revealed

Programs	bzip2	hmmmer	lbm	mcf	sjeng
% of jump instr. revealed	0.75	0.28	0.64	0.36	0.76

Dynamic analysis is time consuming than static analysis. When static disassembly of *SPECint-2006* programs take minutes to complete its action, dynamic analysis takes hours. Figure 3.7, shows the time taken for dynamic disassembly of *SPECint-2006* and how it varies as the number of breaks varies.

The time taken for dynamic analysis is comparatively larger than that of static analysis. For instance, in the case of 5000 breaks *ald* has to disassemble the program 5000 times. For dynamic analysis of *bzip2* program with 100,000 breaks it takes 38.4 hours.

Figure 3.7: Time complexity of dynamic analysis

3.5.3 Cost of Obfuscation

In this section we will discuss the overheads caused by the obfuscation to the program. The two overheads are, increase in the space complexity and time complexity of the program.

3.5.3.1 Program Size Overhead

Obfuscation will have effect on the size of the program. $Space_{ovh}$ defines the increase in the size of the program.

$$Space_{ovh} = \frac{S_{code1} + S_{data1}}{S_{code0} + S_{data0}} \quad (\text{Eq. 3.3})$$

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

One advantage of our algorithm is that the program space has not bloated up too much after obfuscation. Table 3.4 shows the space overhead caused by our algorithm.

Table 3.4: Space overhead

Programs	Space _{before} (in bytes)	Space _{after} (in bytes)	Space _{ovh}
bzip2	589489	1296240	2.19
hmmer	862922	2045808	2.37
lbm	527128	1103728	2.09
mcf	533107	1140592	2.14
sjeng	707023	1562480	2.21
Mean			2.22

The average increase of programs after obfuscation is 2.2 times of the original size. The increase in size is due to two reasons. The reconstruction instructions added in the program contributes to increasing the size of the program. For each jump instruction removed from the program, additional instructions are added to reconstruct the instruction at runtime. Similarly instructions are added to dynamically obfuscate the de-obfuscated jump instruction.

Another reason is the insertion of junk bytes to achieve more instruction disassembly errors. Junk bytes are added in the succeeding block of the removed jmp instruction. This also accounts to the increase in the size of the program.

3.5.3.2 Time Overhead

Obfuscation will have effect on the time complexity of the program. With the insertion of new instructions more instructions are computed during runtime. In this section we will discuss the increase in the time complexity due to obfuscation. We evaluate the effect of obfuscation on execution speed with $Time_{ovh}$ defined as,

$$Time_{ovh} = \frac{T_{obf}}{T_{ori}} \quad (\text{Eq. 3.4})$$

T_{ori} refers to the execution time of the original file and T_{obf} refers to that of the obfuscated code. Table 3.5, shows the time complexity overhead created by our algorithm.

Table 3.5: Time overhead

Programs	Original (T_{ori})	Obfuscated (T_{obf})	Slowdown (T_{obf} / T_{ori})
bzip2	215	312	1.45
hammer	1276	1660	1.30
lbm	1310	1578	1.21
mcf	468	621	1.32
sjeng	980	1323	1.35
Mean			1.29

3.5.4 Stealth Analysis

The stealth of obfuscation measures the difficulty to identify whether a binary is obfuscated or not. To measure the stealth of our obfuscation we considered measuring Mahalanobis distance between the obfuscated program and normal binary samples as discussed in [LKK10]. Mahalanobis distance is a common measure used for statistics based malware detection [SKF08] [SWL07]. It is the dissimilarity measure between two random vectors whose components are scalar-valued on the same probability space [TK08].

To measure the stealth of our obfuscation we calculated the Mahalanobis distance (M_{dist}) of the obfuscated binary and 248 normal binaries:

$$M_{dist}(X, O) = \sqrt{(X - O)S(X - O)} \quad (\text{Eq. 3.5})$$

X is a vector that represents the probability mass function (pmf) of each opcode's mean occurrence in the normal binary programs. S is the diagonal matrix which is made up of variance of mean values of each opcode for the normal binary programs. Mahalanobis distance calculates the distance of a programs opcode distribution from the mean vector X . O represents the pmf of each opcode's occurrence in the program whose distance from the mean is calculated. After calculating X and S from 248 binary programs we measured the M_{dist} of each of the normal binaries from the mean and is

plotted as shown in Figure 3.8. The M_{dist} of the obfuscated binaries were then calculated and plotted. We observe that the M_{dist} of the obfuscated binaries is in the range of normal binaries.

Figure 3.8: Mahalanobis distance of binary programs

Further, we calculated the mahalanobis distance of binary programs without obfuscation and compared it with their equivalent obfuscated binaries. The mahalanobis distance of the obfuscated binary programs is not far from the original binary program. Table 3.6 shows the mahalanobis distance of binary programs before and after obfuscation.

3.5.5 Comparison with Other Algorithms

The performance of the algorithm is compared with two algorithms, namely signal-based obfuscation [PDA07] (SBC) and self modifying code based algorithm [SE10] (SMC). Figure 3.9, shows the comparison on the basis of instruction disassembly error, control flow error and space and time overhead of our algorithm with the other two.

Table 3.6: Mahalanobis distance before and after obfuscation

Programs	Original(MD_{or})	Obfuscated(MD_{ob})	MD_{ob}/MD_{or}
bzip2	0.096	0.107	1.115
hmmer	0.101	0.103	1.019
lbm	0.103	0.106	1.029
mcf	0.104	0.106	1.019
sjeng	0.099	0.108	1.091

Figure 3.9: Algorithm performance comparison

Instruction disassembly error and control flow errors achieved by our algorithm, is better than the other two algorithms. The size and time overhead created by our algorithm is comparable to the other two algorithms.

3.6 Conclusion

In this chapter we proposed a software obfuscation algorithm to increase the difficulty in reverse engineering binary programs. The control flow information from the program are removed from the code area and stored in the data area, stack, and reconstructed

dynamically on demand. The concept of adding junk bytes and randomization are used to make the disassembly process harder. Potency, stealth and cost overheads of the algorithm are measured against static and dynamic analysis. The evaluation results show that the proposed method is effective in confusing professional disassemblers like IDAPro, potent against dynamic analysis and give stealthy obfuscation. Comparing with other algorithms like signal based approach [PDA07], the proposed algorithm has better potency and cost effectiveness.

One of the shortcomings of this method is that, even though we managed to move the target addresses to data area from code area, it is represented in its raw form. In the next algorithm in chapter 4, we improve this by encrypting the target addresses stored in the data area.

Another limitation of our method is that we are attaining control flow obfuscation by self modification of the programs at runtime. Program area has to be made write permissible to attain our obfuscation which will make the program vulnerable. However, our obfuscation methods explained in chapter 5 and 6 does not have this limitation and can attain control flow obfuscation without changing any write permissions.

Chapter 4

Software Protection by Obfuscation and Encryption

In this chapter we discuss a new approach which uses obfuscation and encryption hand in hand to provide better security to the binary programs. We use dynamic code mutation to attain control flow obfuscation. The control flow obfuscation is attained using dynamic code mutation similar to the method discussed in chapter 3. The main difference in this algorithm is that the control flow addresses are hidden in the static data area than the volatile stack. This gives the scope of encrypting the information stored in the data area thereby providing better protection.

The chapter is organized as follows. Section 4.1 discusses the motivation behind designing this algorithm. Section 4.2 discusses the threat model assumption we make. Section 4.3, explains the protection model, followed by our algorithm in section 4.4. Mathematical analysis of our algorithm is discussed in section 4.5. Section 4.6 discusses the experimental evaluation of our method.

4.1 Motivation

Many of the control flow obfuscation algorithms works in the concept of removing the control flow instructions from the binary program. There will be extra code or modules

added by the obfuscation algorithm to the program code section to re establish the control flow during runtime. The control flow addresses will thus be available in these extra modules and code snippets. An attacker with the knowledge of the algorithm can analyze these modules to find out the control flow addresses. In the method described in chapter 3, we made it more difficult for an attacker by hiding the control flow addresses in the volatile stack. Even then, the control flow addresses are exposed in the stack when a function is getting executed. In this chapter we discuss how we can encrypt the store the control flow addresses so that the attacker has difficulty in finding the control flow addresses in its true form.

We compare this algorithm with the signal based algorithm [PDA07], self modifying based algorithm [SE10], stack based algorithm explained in chapter 3 and M-process based algorithm [GCT05]. M-process based algorithm [GCT05], is an algorithm where combination of obfuscation and encryption is discussed. The basic method is to replace the control flow jumps with a function call to another procedure (M-process). M-process acts as a jump table and is encrypted in parts and decrypted on demand. The advantage of this method is that only a small part of the M-process will be open at a time.

The disadvantage of this method is that, given the knowledge of the algorithm, an adversary can decrypt the jump table procedure statically. The encryption keys are stored in M-process and the part which is open has the key to the next encrypted part. Thus with the knowledge of the algorithm an attacker can decrypt M-process statically.

4.2 Threat Model

The thread model assumptions we make for this algorithm is similar to the threat model defined in chapter 3 with slight modifications. Our assumption is that the attacker is trying to disassemble the binary program and trying to get the assembly level control

flow graph representation. The threat posed by the attacker is modelled by considering three factors:

- Origin of the threat
- Tools available for the adversary
- Knowledge of the adversary

We assume a category 3 origin of threat in our threat model. In category 3, the attacker has complete control over the system. The attacker can analyze the program, modify it and execute it.

We assume a recursive traversal based tool is used by the attacker to disassemble. Commercial disassembly tool, IDAPro [[Hex](#)] is assumed to be the tool used by the attacker.

The knowledgeable attacker will be able to disassemble the binary program more effectively than an attacker who blindly disassembles using a disassembler. If the attacker knows if the program is obfuscated and the obfuscation algorithm, then the attacker can fine tune the disassembly process for the particular algorithm. We assume that the attacker knows that the program is obfuscated but does not know about the exact obfuscation algorithm used.

Hence, the threat model can be concluded as an attacker who knows that the program is obfuscated, and who has complete access to the program, trying to disassemble the binary program using a reverse engineering tool, based on recursive traversal technique, like IDAPro.

4.3 Protection Model

Our premise is to design an obfuscation algorithm, which will obscure the programs control flow, making it difficult for an attacker to disassemble the program. We use

the concept of self modifying code, control flow obfuscation and junk byte insertion to obscures the program and use encryption to protect the obfuscation information in the program.

The basic idea of our obfuscation is similar to the method in chapter 3. We extract the target addresses of control flow instructions like jump instructions and store it in the global data area. Since the control flow addresses are in global data area we can encrypt them. The target addresses are encrypted and stored in the data area. During runtime the target address needed is decrypted and used for reconstructing the jump instruction. We introduce reconstruction instruction in the program which will use the target address information stored in the data area to reconstruct the jump instructions during runtime. This will restore the semantics of the program at runtime. We further explain our algorithm in detail.

4.4 Proposed Algorithm

Our algorithm takes an assembly program as input and gives the binary program as output. The algorithm has two phases. The first phase is obfuscation. In this phase control flow obfuscation is achieved using self modifying code obfuscation technique. This obfuscation will change the semantics of the program which is corrected dynamically by self modifying code. The second phase of the algorithm is encryption, which encrypt the information required for self modification of the program at runtime. In the first pass we analyze the program to find the suitable instructions for obfuscation. Obfuscation, the first phase of our algorithm, as shown in algorithm 1, is described in detail in the following subsection:

Algorithm 1 Obfuscation pass-1

Input: Assembly program as input**Output:** Control flow obfuscation*D* $\leftarrow$ *New Data Area**Func_List* $\leftarrow$ *Find User Defined Functions***for** *each Function in Func_List* **do***Jmp_List* $\leftarrow$ *Find Jump Instructions to be Obfuscated***for** *each Jump in Jmp_List* **do***Tgt_Add* $\leftarrow$ *Extract Target Address**Store Target Address in D**S* $\leftarrow$ *Size of Jump Instruction**N* $\leftarrow$ *Normal Instruction of Size S**Replace Jump Instruction with N**Add Re – construction Instructions**Add Re – obfuscation Instructions***end for****end for**

4.4.1 Obfuscation

4.4.1.1 Create New Data Area

The first step is to create a new data area to store the target address of the jump instructions that are going to be obfuscated. The assembly code for this has to be added to the input program, during obfuscation time. This data area is also used to store the encryption keys, used to encrypt and decrypt the target addresses.

4.4.1.2 Find Functions to Obfuscate

Once the data area is created the algorithm will scan the assembly program to find suitable functions for obfuscations. A program will have a set of system procedures and external library calls associated with it. There is no point in obfuscating these functions as an attacker can always get the unobfuscated binary version of these functions. Hence, in our implementation we consider the user defined functions as our targeted functions for obfuscation.

4.4.1.3 Finding Instructions to Camouflage

An important step of the algorithm is to identify which all instructions have to be camouflaged. The trivial method is, randomly picking instructions from the code area. But, in our method the jump instructions are chosen to be camouflaged for the following reasons.

Jump instructions decide the control flow of a procedure in the program. By obscuring the jump instructions in the procedure we are thus obfuscating the control flow of the program. Instructions which give information about the control flow of the program will help the adversary to easily understand the logic of the program. Another motivation for considering jump instructions to be camouflaged is the scope it provides for inserting junk bytes in the program as explained in [\[LD03a\]](#).

4.4.1.4 Storing Target Address to Data Area

Once the jump instruction to be obfuscated is decided, the next step is to extract the target address from the jump instruction. In the data area created, space will be allotted to store the target address. The assembly program is modified in such a manner that the target address is then stored in the data area.

4.4.1.5 Obfuscating Jump Instructions

The jump instruction can be obfuscated after storing the target address to the data area. The size of the jump instruction is calculated and the jump instruction is replaced by one or more instructions to fill up the void. The easiest approach is to replace the jump instruction with *nop* no operation, instructions. If the jump instruction is of size 2 bytes then, the jump instruction is replaced by two *nop* instructions. We can use other normal instructions like *mov*, *add*, *mul*, etc. The key factor is that the camouflaging instruction(s) should have size equal to or more than of that of the jump instruction. This is to ensure that there are enough bytes in the program that can be modified to re-construct the jump instruction during runtime.

By replacing jump instruction with other normal instruction, the program loses its control flow information. When an automated disassembler tries to disassemble the program, it assumes the control flows just to the next address location after the normal instruction. In Figure 4.1, the camouflaged *jmp* instruction is at address location A1 in basic block B1. The *jmp* instruction is camouflaged into *mov* instruction and the reconstruction instructions are added before the camouflaged instruction

4.4.1.6 Adding Re-construction Instructions

We cannot execute the program with the camouflages in place as it changes the semantics of the program. The program semantics should be restored before it gets executed. We use self modifying code, similar to our previous technique, to restore the semantics.

Reconstruction instructions which reconstruct jump instruction at runtime are inserted in a block that precedes the jump instruction. The block in which the reconstruction instructions are added should be a dominator block. Block A is a dominator block to block B if and only if block A precedes block B in all execution paths. The insertion of reconstruction instructions are shown in Figure 4.1.

In the example shown in Figure 4.1, the *jmp* instruction is replaced by *mov* instruction. The opcode of *jmp* instruction is 0xE9 and that of *mov* instruction 0xB8. We add an instruction to XOR the address location of *mov* instruction with 0x00000051. This changes the instruction to *jmp offset 0*. Now the next step is to add the address offset stored in the data area to the instruction. We add an instruction to add the value in the global variable to the instruction address. Now the exact *jmp* instruction is created at the address location of *mov* instruction.

In Figure 4.1, the camouflaged *jmp* instruction is at address location A1 in basic block B1. The *jmp* instruction is camouflaged into *mov* instruction and the reconstruction instructions are added before the camouflaged instruction. The only difference in this

method compared to our previous technique is that the target address is stored in the data segment instead of stack.

Figure 4.1: Obfuscation of jump instruction

4.4.1.7 Adding Re-obfuscation Instructions

The addition of re-construction instructions makes sure that the obfuscated program is semantically equivalent to the original program. Now, after the execution of the re-construction instructions the camouflaged instruction is in its original form. An adversary tracks the image of the program at regular intervals will be able to find the de-obfuscated instructions.

To address this problem, we introduce the concept of dynamic re-obfuscation. The jump instruction is dynamically camouflaged at runtime. This can be achieved by adding extra re-obfuscation instructions in the succeeding blocks of the jump instruction to camouflage it back to ordinary instructions. Note that, the re-obfuscation instruction should be inserted in all the successor blocks as the execution path is chosen dynamically at runtime.

In the example shown in Figure 4.1, re-obfuscation is done by XOR-ing the *jmp* instruction with 0x00000051 to get the instruction: *mov eax, 0*

According to the control flow of the example in Figure 4.1, the basic block B3 follows after the execution of the *jmp* instruction. The re-obfuscation instructions for the program are hence added in the beginning of the basic block B3.

4.4.1.8 Randomization of Re-construction Instructions

Randomization of the re-construction instructions are used so that the attacker wont be able to infer any knowledge by searching for particular pattern of instructions. In the example shown in the thesis we use *xor* and *add*. We can use various combinations of arithmetic and logical operations to achieve the same result. The selection of these set of instructions happens at random during obfuscation time. So, an adversary cant look at specific instructions alone to sort out obfuscation points.

4.4.1.9 Randomization of Re-construction Locations

The only condition for the block in which the reconstruction instructions are added is that the block should dominate [ALSU06] the instruction to be camouflaged. The reconstruction instruction can be added in any of the dominators [ALSU06] of the camouflaged instruction. The selection of the block from the list of dominators happens at random during obfuscation time.

4.4.2 Encryption

Encryption is the second phase of our algorithm. In this phase the stored target addresses in the newly created data area will get encrypted. In this pass the target addresses in the newly created data area are encrypted using randomly generated symmetric keys at obfuscation time. The symmetric keys are stored in the data area and decryption and re-encryption functions are added to the assembly program to decipher the target addresses on demand. The output of this pass in the algorithm is the final obfuscated binary program. Algorithm 2, is the algorithm for second pass. The detailed steps of the algorithm are as follows

Algorithm 2 Obfuscation pass-2

Input: Assembly program from pass-1, D_i - Data area created in pass 1

Output: Control flow obfuscation with encryption

```

for each Tgt_Add in D do
    K  $\leftarrow$  Create Random Key
    Add K to D
    Encrypt Tgt_Add with Key
    Instr  $\leftarrow$  Call Decrypt
    Insert Instr before Camouflaged Instruction
    Instr  $\leftarrow$  Call ENcrypt
    Insert Instr after Camouflaged Instruction
    Jmp_List  $\leftarrow$  Find Jump Instructions to be Obfuscated
end for
Do Randomization
Assemble to Binary Program

```

4.4.2.1 Create and Store Random Symmetric Key

For each target address stored in the newly created data area, we assign one symmetric key. The symmetric key is randomly assigned during obfuscation time. These keys are used to encrypt the target addresses stored in the newly created data area. The keys are then stored along with the encrypted target addresses in the newly created data area. The position of the key and the target address in the data area are randomly assigned.

4.4.2.2 Encrypting the Target Address

Once the keys are generated, they are used for encrypting the target address in the newly created data area. During obfuscation time, the target addresses get encrypted using the corresponding key. The encryption method we use is XOR. The target address is XOR-ed with the key and is then stored in the data area as shown in Figure 4.2. The target addresses does not share keys, each target address use a separate key, and hence the strength of the encryption is as good [SB]. Another advantage of using XOR is its simplicity.

4.4.2.3 Adding Decryption Method

With the encrypted target address in the newly created data area if you try to run the program, it will crash. This is because the correct target address is required for reconstructing the camouflaged jump instruction. Therefore the target address should be decrypted to its original values before the reconstruction instructions use it. The instruction to call the decryption function is thus added before the camouflaged instruction as shown in Figure 4.2, during obfuscation time. The decryption function just takes one input parameter, which is the location of key, in the data area.

3.1)Runtime decryption method: We have designed decryption algorithm in such a manner that not much information about the decrypted data is understood by analyzing

Figure 4.2: Assembly program after obfuscation

the decryption function. The input to the decryption function is a location in the data area, where the key to be used for decryption is stored. The decryption algorithm reads the key from the data area. Then, each element in the newly created data area is XOR-ed using the key. Since the decryption and encryption functions are both XOR, this will decrypt the target address. So, the specific target address to which the key was associated gets decrypted. All other target addresses and keys stored in the newly created data area changes due to the XOR-ing.

One of the advantages of this method is that an attacker will not be able to know which specific target address the key is associated to. Another advantage is that during a decryption process every other target address gets modified along with their keys. Since, every element in the data area is XOR-ed; the other keys are also XOR-ed. Hence both the keys and the target addresses are XOR-ed with the current key. This changes both

of them in such a way that the new keys can decrypt their target addresses when used by the decryption algorithm. Algorithm for decryption is shown in algorithm 3.

Algorithm 3 Decryption

Input: Key location - K, Data Area - D

Output: Decryption of the target address

$KEY \leftarrow D[K]$

for each X in D **do**

$X \leftarrow X \text{ xor } KEY$

end for

4.4.2.4 Adding Encryption Method

Call to encryption function is added to the successive blocks of the camouflaged instruction as shown in Figure 4.2, during obfuscation time. During program execution, decryption of the target address happens before the execution of the reconstructed camouflaged jump instruction. At the point of jump instruction, the target address is decrypted and is in the true form. So after the execution of the jump instruction, the re-obfuscation instructions in the successive blocks camouflage the jump instruction again. The encryption function is called immediately after the jump. The encryption function randomly generates a new key and encrypts the target address with the new key, at runtime.

4.1)Runtime encryption method: Encryption method randomly generates a key at runtime. This key is used to encrypt the newly created data area, entirely. In our case, the new key is XOR-ed with every element in the newly created data area. This will change the keys and the target addresses dynamically. Algorithm 4, shows the encryption algorithm.

4.4.2.5 Randomization in Data Area

We use the newly created data area to store both the keys and the target addresses. There is no specific pattern or location for storage of keys and the target addresses.

Algorithm 4 Encryption

Input: Data Area - D **Output:** Decryption of the target address $KEY \leftarrow \text{Random}()$ **for each** X **in** D **do** $X \leftarrow X \text{ xor } KEY$ **end for**

Completely random locations are assigned for each key and target address. The random locations are decided during obfuscation time and hence the obfuscation algorithm will know the location of the keys. Looking just at the pattern of storage one cannot conclude which is a key and which is target address. Similarly, one cannot find the relationship between a key and a target address by just looking at the data storage pattern.

4.4.2.6 Junk Decrypt and Encrypt Call Insertions

The argument to the decryption function call is the location to a key stored in the data area. So, if an adversary sees a decrypt call in the program he can infer that the argument used in that call is a location for the key. This is not desirable. Hence, decryption and encryption calls with wrong key locations, are inserted in the program. These insertions will be done at unreachable code area so that it will not affect the semantics of the program.

4.5 Mathematical Analysis

In the section we investigate the validity of decryption and encryption processes in our algorithm and the complexity for an adversary to crack the obfuscation-encryption scheme so that he gets the target addresses in its original form.

Let the number of target addresses in the newly created data area be N , and then the number of keys used will also be N . Let d_i , and k_i for $i = 1$ to N , represent the N target addresses and the corresponding keys respectively. Key k_i , is used to encrypt and

decrypt the target address d_i , for all $i = 1$ to N . Let e_i , for $i = 1$ to N , represent the encrypted form of target address stored in the data area.

4.5.1 Validating Decryption Process

The relationship between the key k_i , d_i , and e_i in the data area before the execution of the program is the following:

$$e_i = k_i \oplus d_i \quad (\text{Eq. 4.1})$$

When a decryption function is called, we expect the relationship between the keys, target address and the encrypted target address to remain the same. Another expectation is the decryption of the intended target address. When a decryption function call of $decryption(k_m)$ happens, key k_m is XOR-ed with every element in the data area. This gives the following,

$$k_i' = k_i \oplus k_m \quad (\text{Eq. 4.2})$$

$$e_i' = e_i \oplus k_m \quad (\text{Eq. 4.3})$$

When $m = i$,

$$e_i' = e_i \oplus k_i$$

$$e_i' = d_i \oplus k_i \oplus k_i \text{ (from eq.4.1)}$$

$$e_i' = d_i$$

ie., the intended target address get decrypted. Now, for all other keys the new key (k_i') and the new encrypted (e_i') should have the same relationship as in eq. 4.1..

$$e_i' = e_i \oplus k_m$$

$$e_i' = d_i \oplus k_i \oplus k_m$$

$$e_i' = d_i \oplus k_m' \text{ (from eq. 4.2)}$$

4.5.2 Validating Encryption Process

In encryption function, a random key is generated and is xor-ed with every element in the data area. This is similar to the decryption algorithm except that none of the encrypted target addresses are decrypted. So, the proof that the relationship between the keys, target addresses and the encrypted target addresses are maintained at the end of execution of encryption is same as that of the decryption algorithm.

4.5.3 Complexity in Finding the Target Addresses

In this section we will show how hard it is for an adversary to find the target addresses from the data area. The keys and target addresses are mixed together and stored in the data area. For an adversary it is hard to differentiate the key and the target address from the data section due to the randomizations in the obfuscation. And to get the right target address the adversary should find the right key as the target addresses are encrypted and stored.

One approach the adversary can follow is to guess the key from the elements in the data area. If there are n target addresses there are $2n$ elements in the data area. Only one in the n combinations attacker guesses from $2n$ elements is the right set of keys.

Since, there are n keys in the $2n$ elements in the data area, an attacker can get a key in one guess with a probability of $1/2$. Once the key is found the probability of finding the right data element of the key is $1/(2n-1)$. Probability of finding one key-data pair is then,

$$P(\text{finding first key} - \text{datapair}) = \frac{1}{2(2n-1)}$$

After one key-data pair is found, the total number of elements in the data area is reduced to $2n-2$. Therefore, after i key-data pair is found the probability of finding a key data pair is,

$$P(\text{finding key - data pair after } i \text{ iteration}) = \frac{1}{2(2(n-i)-1)}$$

The probability of finding all the keys is the product of finding individual keys and hence,

$$P(\text{finding all key - data pair}) = \prod_{i=0}^n \frac{1}{2(2(n-i)-1)}$$

$$P(\text{finding all key - data pair}) \leq \prod_{i=0}^n \frac{1}{2}$$

$$P(\text{finding all key - data pair}) \leq \frac{1}{2^n}$$

If the probability of finding the all key-data pair is atleast $\frac{1}{2^n}$ then the experiment has to be repeated atleast 2^n to find all the key-data pairs. Therefore the number of times the guesign has to be done is greater than or equal to 2^n . The asymptotic lower bound for finding all the key-data pairs is thus $\Omega(2^n)$ [CLRS]. Thus the time complexity to find the key-data pairs are at least exponential.

4.6 Experimental Evaluation

In this section we evaluate the performance of the proposed algorithm against automated attacks. We tested the potency of our algorithm with IDAPro 6.2 [Hex], and measured the instruction disassembly error and control flow error caused by the obfuscation. The increase in the size and time complexity due to the addition of self modifying code and encryption/decryption functions are also measured. Microsoft Visual Studio 10.0 is used to generate the assembly programs for obfuscation.

4.6.1 Instruction Disassembly Error

We evaluate the instruction disassembly error with confusion factor. Confusion factor is the fraction of instruction address that the disassembler fails to identify [LD03a]. T_{total}

Table 4.1: Instruction disassembly error

Programs	T_{total}	T_{disasm}	CF_{instr}
Qsort	199	28	85.9%
Mergesort	397	114	71.3%
Huffman Encoding	1904	449	73.6%
Gauss Jordan	348	30	91.4%
Mean			78.2%

is the total number of actual instruction addresses before obfuscation and T_{disasm} is the total number of instruction addresses properly recognized by the disassembler, then the confusion factor is defined by the following,

$$CF_{instr} = \frac{|T_{total} - T_{disasm}|}{T_{total}} \quad (\text{Eq. 4.4})$$

Table 4.1 shows the instruction confusion factor of assembly programs generated by Microsoft Visual Studio and the obfuscated assembly. The average instruction disassembly error of the test programs is 78.2%. This means that the disassembler succeeds in recovering only 22

4.6.2 Control Flow Disassembly Errors

We calculated the number of conditional and unconditional jump instructions in the program before and after the obfuscation. If CFG_{before} is the total number of conditional and unconditional jump instructions in the program and CFG_{after} is the total number of jump instructions in the obfuscated program. CF_{cfg} is the confusion factor in the control flow of the program,

$$CF_{cfg} = \frac{|CFG_{before} - CFG_{after}|}{CFG_{before}} \quad (\text{Eq. 4.5})$$

Table 4.2: Control flow errors

Programs	CFG_{before}	CFG_{after}	CF_{cfg}
Qsort	11	5	54.4%
Mergesort	29	10	65.5%
Huffman Encoding	191	69	63.9%
Gauss Jordan	40	16	60%
Mean			63.1%

The ratio gives the control flow confusion caused by the obfuscation as shown in Table 4.2.

4.6.3 Time Overhead

Obfuscation will have effect on the time complexity of the program. With the insertion of new instructions more instructions are computed during runtime. In this section we will discuss the increase in the time complexity due to obfuscation. We evaluate the effect of obfuscation on execution speed with $Time_{ovh}$ defined as,

$$Time_{ovh} = \frac{Time_{after}}{Time_{before}} \quad (\text{Eq. 4.6})$$

T_{before} refers to the execution time of the original file and T_{after} refers to that of the obfuscated code. An average time overhead of 1.26 is caused by our algorithm.

In the test programs, Qsort and Mergesort programs are run on an input set of 1000 element array, whose values are stored in reverse sorted order. Huffman encoding program is run on an input text document with thousand words. Gauss Jordan program is run on an input of 8×8 matrix.

Table 4.3: Time overhead

Programs	Original (Time _{before})	Obfuscated (Time _{after})	Slowdown (T _{ovh})
Qsort	310	420	1.35
Mergesort	770	1100	1.42
Huffman Encoding	1320	1550	1.17
Gauss Jordan	890	1090	1.22
Mean			1.26

4.6.4 Time Overhead with Varying Input

The time complexity of a program is dependent on the input size of the program. For instance, the time complexity for searching an element in an array of length n is $O(n)$. So as the length of the array increases the time overhead increases linearly. This means that the instructions in the program are executed in the order of n . The number of instructions in the program also affects the time complexity. If the program has a instructions then $aO(n)$ is the time complexity.

In our obfuscation technique, all the new instructions added are straight line code and has a time complexity of $O(1)$, when they are executed independently. However, the effect of the newly added instruction on the time complexity also depends on the point at which they are inserted. For example if a jump instruction within a loop is obfuscated, then the new instructions are added within the loop and hence the new instructions will be executed n -times if the loop size is n . Similarly, if the instructions are added in a 2-nested loop, they get executed n^2 times.

So, if the worst case time complexity is $T(n)$ then the newly added instructions will get executed $T(n)$ times. If the program has a instructions and x obfuscation instructions are added to the program, then the time to execute the program will be $(a+x)T(n)$. Hence, the time complexity of the obfuscated program is also of the same order $T(n)$.

We have tested this in two programs of complexity $O(n)$ and $O(n^2)$. A linear search in an array to find an element is the first program used for the test. We designed the array in such a manner that the searched value is the last element of the array. We changed the size of the array from 10000 to 100000 and calculated the time needed for the execution. The same program is then obfuscated which obfuscates the jump instruction inside the loop. The time required for the execution of the obfuscated program is also calculated and the results are shown in Fig. 4.3.

Bubble sort program with array size varying from 1000 to 10000 was also used to test our results. We designed the array in such a way that it is arranged in descending order of values thereby making the complexity of the program to be $O(n^2)$. The jump instruction inside the two loops was obfuscated by our obfuscation algorithm. We calculated the time overhead of original and obfuscated programs for varying inputs and are plotted in Fig. 4.4

In both the cases the time overhead of the obfuscated program is sandwiched between $T(n)$ and $2T(n)$.

Figure 4.3: Time complexity of linear search program

Figure 4.4: TIme complexity of bubble sort program

Table 4.4: Space overhead

Programs	$Space_{before}$ (in kilobytes)	$Space_{after}$ (in kilobytes)	$Space_{ovh}$
Qsort	44	44.5	1.01
Mergesort	74	75	1.01
Huffman Encoding	60	66.5	1.10
Gauss Jordan	67	69	1.02
Mean			1.04

4.6.5 Program Size Overhead

Obfuscation will have effect on the size of the program. $Space_{ovh}$ defines the increase in the size of the program.

$$Space_{ovh} = \frac{Space_{after}}{Space_{before}} \quad (\text{Eq. 4.7})$$

The average increase of binary programs after obfuscation is 1.04 times of the original size as shown in Table 4.4.

Table 4.5: Comparison with other algorithms

Comparison Item	Signal based	SMC based	M Process	Stack based	Proposed Algorithm
Instr. disas.error	57.3%	75.3%	not available	79.78%	78.2%
Control. flow error	41.2%	52.2%	not available	61.35%	63.1%
Size	2.39	2.05	1.7	2.22	1.04
Time	1.38	1.21	1.32	1.29	1.26

4.6.6 Comparison with Other Algorithms

The performance of our algorithm is compared with three algorithms, namely signal-based obfuscation [PDA07] (SBC), self modifying code based algorithm [MAM⁺06] (SMC) and M-Process based obfuscation [GCT05] and our method in chapter 3. Table 4.5, shows the comparison on the basis of instruction disassembly error, control flow error and space and time efficiency of our algorithm with the other two. Our algorithm has better instruction disassembly error, control flow error, space efficiency and comparable time efficiency.

Our method has better instruction disassembly error and control flow error and space efficiency than signal based algorithm and M-process based algorithm. The instruction disassembly of this algorithm is slightly lesser than our previous stack based algorithm. Time overhead of our algorithm is comparable with the other algorithms. We compared our algorithm with M-Process based obfuscation because it is the only other obfuscation algorithm, we found, which uses encryption and obfuscation hand in hand. Our algorithm has a better size and time overhead compared to [GCT05] and the other three algorithms. The instruction disassembly error and control flow error of [GCT05] are not available for comparison.

4.7 Obfuscation of Factorial Program

Figure 4.5 shows the original assembly program of factorial program generated by Microsoft Visual Studio 10.0. The two jump instructions in the program are obfuscated by our algorithm. Figure 4.6 shows the obfuscated assembly representation. The new function *decrypt* is introduced into the program. We use the same function *decrypt* for decryption and encryption. A new data segment to store the keys and data is also introduced into the program. The code introduced for the obfuscation of the first jump instruction is coloured in the example. The jump instruction is translated to two *nop* instructions in the example.

```
PUBLIC  _factorial
_TEXT  SEGMENT
_n$ = 8
_factorial PROC
    push    ebp
    mov     ebp, esp
    cmp     DWORD PTR _n$[ebp], 0
    jne     SHORT $LN1@factorial
    mov     eax, 1
    jmp     SHORT $LN2@factorial
$LN1@factorial:
    mov     eax, DWORD PTR _n$[ebp]
    sub     eax, 1
    push    eax
    call    _factorial
    add     esp, 4
    imul    eax, DWORD PTR _n$[ebp]
$LN2@factorial:
    pop     ebp
    ret     0
_factorial ENDP
_TEXT  ENDS
END
```

Figure 4.5: Assembly representation of factorial program

4.8 Conclusion

In this chapter we proposed an algorithm to increase the difficulty in reverse engineering binary programs by combining both software obfuscation and encryption so that it gives a better security. The control flow information from the program are removed from the code area and stored in the data area and reconstructed dynamically on demand. The control flow information is stored in an encrypted form in the data area making it hard for an attacker to infer about the control flow by statically analyzing the data area. The concept of adding junk bytes and randomization are used to make the disassembly process harder. The evaluation results show that the proposed method is effective in confusing professional disassemblers like IDAPro better than the competing algorithms. An average instruction disassembly error of 78.2% and control flow error of 63.1% are obtained by our method with the overhead in time of 1.26 and space of 1.04. The control flow error obtained by this algorithm is slightly better than the stack based algorithm explained in chapter 3, which has a control flow error of 61.3 %. The instruction disassembly error of this algorithm is slightly lesser than the stack based algorithm. The time and space complexity of this algorithm is better than the stack based algorithm. A future work direction is to extend the obfuscation technique using better encryption mechanisms. A limitation of our method is that the keys are stored in the program and is not completely inaccessible to an attacker. Another future work direction is to store and manage the key from a different location than the program itself. A server side key management for obfuscating a client program can store the keys in the server program.

```
_DATA SEGMENT
control0key DB 222
control1key DB 113
control0 DB 246, 00h
control1 DB 98, 00h
_DATA ENDS

PUBLIC _factorial
_TEXT SEGMENT
_n$ = 8
_factorial PROC
    push ebp
    movebp, esp
    cmp DWORD PTR _n$[ebp], 0
    pushf
    push 0
    call decrypt
    add esp, 4
    push eax
    mov al,117
    mov byte ptr[obflabel0],al
    mov al,control0+0
    mov byte ptr[obflabel0+1],al
    pop eax
    popf
obflabel0:
    nop
    nop
    pushf
    push 1
    call decrypt
    add esp, 4
    popf
    moveax, 1
    pushf
    push 1
    call decrypt
    add esp, 4
    push eax
    mov al,235
    mov byte ptr[obflabel1],al
    mov al,control1+0
    mov byte ptr[obflabel1+1],al
    pop eax
    popf
obflabel1:
    nop
    nop
$LN1@factorial:
    pushf
    push 1
    call decrypt
    add esp, 4
    popf
    moveax, DWORD PTR _n$[ebp]
    sub eax, 1
    push eax
    call _factorial
    add esp, 4
    imuleax, DWORD PTR _n$[ebp]

$LN2@factorial:
    pushf
    push 0
    call decrypt
    add esp, 4
    popf
    pop ebp
    ret 0
_factorial ENDP
_TEXT ENDS

PUBLIC decrypt
_TEXT SEGMENT
_temp$ = -5
_i$ = -4
_k$ = 8
decrypt PROC
    push ebp
    movebp, esp
    sub esp, 8
    moveax, DWORD PTR _k$[ebp]
    mov cl, BYTE PTR control0key[eax]
    mov BYTE PTR _temp$[ebp], cl
    mov DWORD PTR _i$[ebp], 0
    jmp SHORT $LN3@fun
$LN2@fun:
    movedx, DWORD PTR _i$[ebp]
    add edx, 1
    mov DWORD PTR _i$[ebp], edx
$LN3@fun:
    cmp DWORD PTR _i$[ebp], 6
    jge SHORT $LN4@fun
    moveax, DWORD PTR _i$[ebp]
    movsxecx, BYTE PTR control0key[eax]
    movsxdx, BYTE PTR _temp$[ebp]
    xorecx, edx
    moveax, DWORD PTR _i$[ebp]
    mov BYTE PTR control0key[eax], cl
    jmp SHORT $LN2@fun
$LN4@fun:
    movesp, ebp
    pop ebp
    ret 0
decrypt ENDP
_TEXT ENDS
END
```

Figure 4.6: Obfuscated factorial program

Chapter 5

Function Level Control Flow Obfuscation

In this chapter we discuss a new approach which obfuscates the control flow across functions. In this method code fragments from each function is stripped from the original function and is stored in another function. Each function will be having code fragments from different functions, thereby creating a function level shuffled version of the original program. Control flow is obscured between and within the function by this method.

The chapter is organized as follows. Section 5.1 discusses the motivation behind designing this algorithm. Section 5.2, explains the proposed model of obfuscation, followed by our algorithm implementation in section 5.3. Section 5.4 discusses the experimental evaluation of our method, followed by section 5.5 where we compare our algorithm with other algorithms.

5.1 Motivation

Most control flow obfuscation algorithms concentrate on obfuscating the control flow within a function. The instruction flow within a function is thoroughly obfuscated by these algorithms but the functions remain intact. Even after the obfuscation the reverse engineering tools can differentiate instructions from one function over another. The obfuscation algorithms explained in the previous chapters also follow this principle while

obfuscating the control flow. To limit the visibility of the functions and the linear order in which they are stored, an inter functional obfuscation is necessary. In this chapter we propose an algorithm to address this issue. Our algorithm performs a function level obfuscation and shuffles the code segments from different functions. This will result in an obfuscated program where, each function block contains code segments from different functions thus confusing the attacker. We compare this algorithm with the signal based algorithm [PDA07], self modifying based algorithm [SE10], stack based algorithm explained in chapter 3. One of the advantages of our algorithm over other algorithms is that it is possible to perform nested obfuscation. The same obfuscation algorithm can be applied over and over on the program. The effect it has on program performance has been analysed in our experimental evaluation section.

5.2 Proposed Method

In this section we discuss our new obfuscation method that can be used against software reverse engineering. Our aim is to attain control flow obfuscation at the function level. Our obfuscator takes assembly level representation of the program as input. The obfuscation algorithm converts the assembly program into a semantically equivalent but obscure assembly program. The obfuscation algorithm analyses the functions in the program, selects suitable code fragments from each function and interlaces them with other functions. Fig. 6.1, represents the overview of the obfuscation algorithm.

The basic idea of our algorithm is to fragment the program and thereby disturb the control flow of the program. Sequence of assembly instructions from different functions of the program are removed and are stored in other functions. At the location of removed assembly instruction, jump instruction is added to change the control flow to the new location of the code fragment. The meaning of the program is thus not clear without the missing instructions. The new instruction fragments added to the function from other

Figure 5.1: Overview of the algorithm

functions also helps in confusing the attacker. The jump instructions to and from the new segment will be camouflaged with the help of dynamic junk byte insertions [BE13]. Further in this section our algorithm is explained in detail.

5.2.1 Selecting the Instructions

The first step of our algorithm is to choose which instructions have to be removed from each function. An obvious method is to select instructions at random from each function. In our algorithm we are following a different strategy. We select sequence of instruction followed by a jump instruction as a candidate code fragment to be moved to another function. The advantage we have by such a selection is that additional control flow obfuscation within the function can be obtained. When we remove a jump instruction from a function, the control flow of that function gets obscured. Without the jump instructions, a function looks like a single block of instructions. This will make it hard for an adversary to analyze and understand the program.

In this step, the obfuscator scans through the functions in the program to find suitable instructions. When it finds three instructions followed by a jump instruction, both conditional and unconditional, it is marked. The beginning and end of the code fragment is recorded by the obfuscator.

5.2.2 Creating and Inserting Labels

Once the obfuscator knows the code fragments to be obfuscated, these code fragments should be made as a basic block [16] by inserting labels. There should be labels to refer to the entry point to the code fragment and exit point of the code fragment. Two unique labels are generated for each code fragment selected for obfuscation by the obfuscation algorithm. These labels are inserted before and after the selected code fragment. Fig. 5.2, shows how the labels are inserted in the program code.

Figure 5.2: Label insertion

5.2.3 Displacing the Code Fragment

The labels *obf_label_enter* and *obf_label_exit* are randomly generated and are unique labels generated at obfuscation time in our implementation. These two label names are used in the thesis for the ease of explanation of the algorithm. The selected code fragment is sandwiched between these two labels.

The code fragments should now be displaced to another function. The obfuscator keeps track of the number of functions in the program. For each code fragment in each function, the source function, the obfuscator does the following. It randomly selects one destination function from all the remaining functions to displace the code fragment to that function. The code fragment along with the entry label, *obf_label_enter* in the example, is replaced from the source function with a jump instruction,

```
jmp obf_label_enter
```

This jump instruction will make sure that the semantics of the program is secure by maintaining the control flow of the program.

The code fragment along with the entry label, *obf_label_enter* in the example, is inserted at the end of the destination function, just after the return instruction. Another jump instruction is added after the code fragment to maintain the control flow of the program.

```
jmp obf_label_exit
```

This jump instruction makes sure the programs control flow is properly maintained. The Fig. 5.3, shows how the selected instructions are displaced into another function without disturbing the semantics of the program. The selected instruction from function *Foo* is displaced to function *Bar* in the example shown in Fig. 5.3.

Figure 5.3: Displacing the code fragment

5.2.4 Hiding the Jump Instructions

Displacing the code fragments from one function to another makes the program complex to follow but the control flow of the entire program will still be shown by powerful reverse

engineering tools like IDAPro [Hex]. So it is necessary to hide the control flow between functions using the newly inserted jump instructions. Our obfuscation algorithm conceals these newly created jump instruction by a technique called dynamic junk byte insertion [BE13] and static junk byte insertion [LD03b].

5.2.4.1 Dynamic Junk Byte Insertion

For each set of instructions that has been removed from a source function, there are two jump instructions that are added to the program. One jump instruction transfers the control flow from source function to destination function, where the code fragment resides. Another jump instruction transfers the control back to the source function.

In the example shown in Fig. 5.3, the jump instruction,

jmp obf_label_enter

transfers control to the code fragment. While the jump instruction

jmp obf_label_exit

transfers control back to the source function.

In our algorithm these two jumps will be concealed using dynamic junk code insertion. In dynamic junk code insertion we will add a junk byte just above the jump instruction as shown in Fig. 5.4. Hexadecimal value *0x10*, is a partial instruction, it is added just above the jump instruction. The machine code for *jmp original_location* is *EB 90*. When the assembly code with the junk byte is assembled the binary program will have the hexadecimal code *10 EB 90* instead of *EB 90*.

The effect of this is that when a disassembling tool tries to reverse engineer the instruction, it will join *0x10* and *0xEB* together to make one instruction, namely *adc bl, ch* as shown in Fig. 5.4. And the hexadecimal byte *0x90* becomes instruction *nop*. So, by adding a junk byte the original jump instruction vanishes from the program. This conceals the control flow of the program. The semantics of the program changes

because of the insertion of this junk byte. In our obfuscation algorithm, self modifying instructions are added to the program that will convert the junk byte to hexadecimal value *0x90* which is the opcode for *nop* instruction.

Figure 5.4: Junk byte insertion

5.2.4.2 Static Junk Byte Insertion

Static junk byte insertion is also used along with dynamic junk byte insertion. In static junk byte insertion the junk byte is not changed dynamically at run time to *nop* instruction. The junk byte remains in its original form throughout the execution of the program. Since the junk byte is not changed, it is important that it should not get executed as it will change the behavior of the program. In this section we will explain how static junk bytes are added to the program execution path without changing the behavior of the program.

Similar to dynamic junk byte insertion, a junk byte is added just above the newly added jump instructions. One jump instruction that transfers control to the displaced code fragment and the jump instruction that transfers control back to source function as shown in Fig. 5.4. From the example shown in Fig. 5.3, the jump instructions that are hidden by static junk byte insertion are *jmp obf_label_enter* and *jmp obf_label_exit*.

To avoid the execution of the junk byte during runtime we add two jump instructions just above the junk byte as shown in Fig. 5.5. These two jump instructions are

complementary to each other. These jump instructions makes sure that the control flow is transferred to the valid address location, *static_junk_label* in the example shown in Fig. 5.5, evading the junk byte.

We use both dynamic and static junk byte insertions to hide the jump instructions in our obfuscation algorithm, if the obfuscation is not nested. In nested obfuscation, explained in Section II.E, we use only static junk byte insertions.

Assembly instruction	Remarks
<i>jz static_junk_label</i> <i>jnz static_junk_label</i> DB 10h <i>static_junk_label:</i> <i>jmp obf_label_enter</i>	Evade the junk byte during runtime Junk Byte

Figure 5.5: Static junk byte insertion

5.2.5 Nested Obfuscation

One of the advantages of our algorithm over other control flow obfuscation is that the obfuscation can be nested. We can reapply the same obfuscation on an obfuscated program. When the obfuscation algorithm is applied to a program, four new jump instructions are added for one displaced code fragments. The jump instruction in the code fragment also exists in the program, but in a different subsection of a different function. This provides scope for further obfuscation.

Obfuscating the obfuscated program will help in concealing the jump instructions used in junk byte addition. The inter-functional shuffling also improves due to nested obfuscation. A disadvantage of nested obfuscation is that the time and space overhead increases. The obfuscator has to decide the number of levels of obfuscation based on the execution time and space requirements.

We do not use dynamic junk insertion for nested obfuscation. Self modifying code requires specific information about the address location to modify the instructions. The

self modifying codes are added to the program at the end of the obfuscation algorithm by calculating the address information. If an obfuscated program with self modifying code is re obfuscated then the address information changes and it affects the instructions responsible for the self modifications.

5.3 Implementation

The proposed obfuscation technique is implemented using python programming language and copt general-purpose peephole optimizer [Fraa] in Ubuntu 13.04 (64 bit) operating system. Intel Core 2 Quad CPU was used for developing and testing the obfuscation algorithm. The obfuscation algorithm expects assembly programs generated by gcc 4.7.3 as input. Obfuscation algorithm generates modified assembly program which is then assembled to binary using the gcc assembler. We explain the implementation of the algorithm in detail in this section.

Figure 5.6: Implemetation overview

5.3.1 Markers with Copt

The copt general-purpose peephole optimizer is used to mark the beginning and end of functions in the program. It is also used for selecting the instructions to be obfuscated. Rules are added to add the comment *#function starts* at the beginning of the function

and the comment *#function ends* at end of the function. Rules are also added to mark the beginning and end of the code fragment that is suitable for obfuscation. A sequence of 3 normal instructions followed by a jump instruction is considered as code fragment suitable for obfuscation. The comments suitable *#obfuscation starts* and *#obfuscation ends* are added by the rules to the program. The copt rules will generate an intermediate assembly file with these markers inserted.

5.3.2 Program Translation by Python Code

Python code analyses the marked assembly file. It keeps track of number of functions and assigns numeric value to each function for assigning it to a different subsection. At the beginning of the function 0 the directive *.text 0* is added by the python program, for function 1 the directive *.text 1* and so on. Thus, each function resides in its own subsection.

When the marker for the obfuscation is seen by the python program, it understands that the code fragment should be assigned to the subsection of another function. The python program randomly picks a subsection number from the subsection numbers of other functions. The directive *.text selected number* is added above the block, so that when the assembler assembles the program the code fragment is appended to the subsection of a different function.

For all the obfuscation points labels are generated by the python program. The labels are added before and after the selected code fragments. The jump instructions between the subsections are also added at this point. A random junk byte is added before the jump instruction. Self modifying code to convert the junk byte to *0x90*, the opcode of *nop* instruction is also added above the junk byte.

The modified program is saved into a new file and the intermediate file created by copt general-purpose peephole optimizer is deleted.

5.4 Performance Evaluation

In this section we evaluate performance of the proposed obfuscation technique against automated reverse engineering attacks. We used IDAPro [Hex] to assess the potency of our obfuscation against automated reverse engineering attacks. Instruction disassembly error and control flow error are two measures we used to evaluate the strength of our obfuscation. Instruction disassembly error refers to the error in disassembling the correct instructions by the automated reverse engineering tool. The total number of control flow instructions displaced by our algorithm is measured by the control flow error measure. Space and time complexity of the program increases, due to the introduction of instructions, by the obfuscation algorithm. These overheads are also discussed in this section. We used the test programs from the lcc 4.2 [Frab] compiler source as input test programs for our obfuscation algorithm.

5.4.1 Instruction Disassembly Error

Instruction disassembly error measures the potency of the obfuscation to confuse an automated reverse engineering tool. We used IDAPro [Hex] to reverse engineer the obfuscated binary program. Confusion factor measures the ratio of instructions that the reverse engineering tool fail to reverse engineer. The total number of instruction addresses before obfuscation is represented by T_{total} . The number of instruction addresses recognized by IDAPro after disassembling the obfuscated binary program is represented by T_{disasm} . The confusion factor is defined by the following equation,

$$CF_{instr} = \frac{|T_{total} - T_{disasm}|}{T_{total}} \quad (\text{Eq. 5.1})$$

Table 5.1 shows the confusion factor while disassembling obfuscated test programs by IDAPro. Mean instruction disassembly error of 78.32% is obtained by our obfuscation.

Table 5.1: Instruction disassembly error

Programs	T_{total}	T_{disasm}	CF_{instr}
8q	283	74	73.85%
array	341	97	71.55%
cf	184	39	78.80%
cq	13786	2799	79.69%
cvt	674	108	83.97%
fields	339	103	69.61%
init	415	124	70.01%
limits	162	43	73.45%
sort	506	132	73.91%
spill	433	108	75.05%
struct	505	79	84.35%
switch	893	205	77.04%
wf1	597	118	80.23%
yacc	2411	637	73.57%
Mean			78.32%

5.4.2 Control Flow Error

The error in identifying the control flow of the program by a disassembly tool is referred to control flow error. The number of control flow instructions in the assembly program before obfuscating is represented as CFG_{before} . The control flow instructions that are not displaced by the algorithm will be appearing in the program even after obfuscation. CFG_{after} represents the number of jump instructions in the assembly program after obfuscation. CF_{cfg} is the confusion factor in the control flow of the program given by the equation,

$$CF_{cfg} = \frac{|CFG_{before} - CFG_{after}|}{CFG_{before}} \quad (\text{Eq. 5.2})$$

Table 5.2, shows the control flow disassembly error caused by our obfuscation on the

Table 5.2: Control flow error

Programs	CFG_{before}	CFG_{after}	CF_{cfg}
8q	13	4	69.23%
array	24	9	62.5%
cf	11	5	54.54%
cq	1363	421	69.11%
cvt	7	2	71.42%
fields	4	1	75%
init	15	3	80%
sort	11	2	81.81%
spill	8	4	50%
stdarg	15	6	60%
struct	8	2	75%
switch	92	21	77.17%
wf1	29	12	58.62%
yacc	139	38	72.66%
Mean			69.11%

test programs. An average control flow error rate of 69.11% has been obtained by the proposed obfuscation method.

5.4.3 Function Level Obscurity

Every function is split into code fragments and the code fragments are stored in the other functions. We used cyclomatic complexity[McC76] to measure the function level complexity. Cyclomatic complexity is a quantitative measure, to find the complexity of a program when it is represented in graph form. The complexity is defined as follows

$$M = E - N + 2 \quad (\text{Eq. 5.3})$$

E = the number of the edges in the graph

Table 5.3: Function Level Obscurity

Programs	Number of functions(N)	Edges(E)	E-N+2
8q	3	6	5
array	3	6	5
cf	2	4	4
cq	106	325	221
cvt	2	4	4
fields	6	18	14
incr	5	16	13
init	9	28	21
limits	2	3	2
sort	7	17	12
spill	6	13	12
stdarg	3	10	9
struct	8	26	20
switch	33	108	77
wf1	7	20	15
yacc	7	13	8
Mean	209	617	412

$N =$ the number of the nodes in the graph

In our measure we assume each function as a node. In a program, which is not obfuscated, each function is defined one after another. Since each function follows other we assume there is a single edge between two functions. The total number of edges for a program that is not obfuscated is $N-1$. Once the program is obfuscated, when a code segment from one function is moved to another function, it represents an edge. The number of edges increases as the shuffling increases.

Table 5.3 shows the cyclomatic complexity measure of the test programs. For an average of 209 nodes the cyclomatic complexity after obfuscation is 410. The average

Table 5.4: Space overhead

Programs	$Space_{before}$	$Space_{after}$	$Space_{ovh}$
8q	4189	4263	1.01
array	4907	5192	1.05
cf	2777	3111	1.12
cq	235092	240139	1.02
cvt	11756	12007	1.02
fields	5172	5451	1.05
incr	4116	4226	1.02
init	5668	5982	1.05
limits	2697	2891	1.07
sort	7546	7944	1.05
spill	6283	6335	1.01
stdarg	6046	6254	1.03
struct	7936	8096	1.02
switch	11987	12451	1.03
wf1	8882	9097	1.02
yacc	32140	33524	1.04
Mean			1.03

cyclomatic number is 1.96 times the average number of nodes in a program.

5.4.4 Space Overhead

Obfuscation will have effect on the size of the program. $Space_{ovh}$ defines the increase in the size of the program.

$$Space_{ovh} = \frac{Space_{after}}{Space_{before}} \quad (\text{Eq. 5.4})$$

The average increase of binary programs after obfuscation is 1.03 times of the original size as shown in Table 5.4.

5.4.5 Time Overhead

One of the costs of obfuscation is the execution time. The execution time of the program increases due to the obfuscation mechanisms. The two jump instructions that get executed additionally for the obfuscation of each sequence of selected instructions adds to the execution time of the program. $Time_{before}$ refers to the time taken by the program to execute without obfuscation and $Time_{after}$ is the time taken by the obfuscated program to execute. We evaluate the effect of obfuscation on execution speed with $Time_{ovh}$ defined as,

$$Time_{ovh} = \frac{Time_{after}}{Time_{before}} \quad (\text{Eq. 5.5})$$

Table 5.5 shows the time overhead caused by our obfuscation on various binary programs. An average time overhead of 1.21 is the overhead caused by the obfuscation. We have calculated the time overhead by running each of the original program and obfuscated program in the test programs for thousand runs with the input program given with the test suite of lcc [Frab].

5.4.6 Time Overhead with Varying Input

We ran bubble sort and linear search program for different inputs. We used arrays of size 10000 to 100000 for linear search program. For bubble sort we used input array size varying from 1000 to 10000. We ran both the original and obfuscated programs and got the results as shown in Fig. 5.7 and Fig. 5.8. Similar to our previous result time overhead of the obfuscated program is sandwiched between $T(n)$ and $2T(n)$.

Table 5.5: Time overhead

Programs	Time_{before} (in milliseconds)	Time_{after} (in milliseconds)	Time_{ovh}
8q	1300	1523	1.17
array	1011	1347	1.33
cf	2777	3111	1.33
cq	1205	1439	1.19
cvt	1077	1342	1.24
fields	1089	1322	1.22
incr	1271	1465	1.14
init	1198	1459	1.21
limits	1324	1654	1.25
sort	1037	1301	1.25
spill	1174	1432	1.21
stdarg	1329	1743	1.31
struct	1342	1753	1.31
switch	1178	1245	1.06
wf1	1245	1460	1.17
yacc	1062	1320	1.24
Mean			1.21

Figure 5.7: Time complexity of linear search program

Figure 5.8: Time complexity of bubble sort program

5.4.7 Nested Obfuscation Overhead

While nested obfuscation increases the complexity of the obfuscation, there by confusing the reverse engineer, it has its costs. The execution time and the space requirement of the programs increase as we nest the obfuscation. We performed nesting up to ten levels and noticed that the execution time of the program increases to 6.35 times the original execution time. The program size increased to 1.32 times the original program size. Fig. 5.9 and Fig. 5.10 shows the change in space and time overhead as the number of obfuscation changes from one to ten.

5.5 Comparison with Other Algorithms

We compared the results of our obfuscation algorithm with three other algorithms. Signal based obfuscation [11], stack based algorithm [14] and self modifying code based algorithm [18], are the algorithms with which we are comparing our obfuscation algorithm. Fig. 5.11 shows the performance of the proposed method with the other algorithms.

Figure 5.9: Space overhead with nested obfuscation

Figure 5.10: Time overhead with nested obfuscation

The proposed method has better control flow error, space overhead and time overhead compared to other algorithms. The instruction disassembly error of our algorithm is as good as the other algorithms. We have an instruction disassembly error of 78.32% compared with the 79.78% of stack based obfuscation algorithm.

Figure 5.11: Comparison with other algorithms

5.6 Obfuscation of Factorial Program

Fig. 5.12 shows the original and the obfuscated assembly representation of factorial program. Only one code snippet from the *factorial* function is moved into the subsection of *main* function. The first function *factorial* is given the subsection number - 0 and the second function *main* is given the subsection number - 1. The code block that is selected for obfuscation is marked as is assigned the subsection number -1 with the assembler directive *text*.

5.7 Conclusion

In this chapter we proposed an obfuscation algorithm to obscure the control flow of the program within the function and across functions. In our method code fragments from each function is stripped from the original function and is stored in another function. Thus, each function will be having code fragments from different functions thus having a function level shuffled version of the original program. This makes it harder for a reverse engineer to disassemble the program successfully. Unlike other control flow obfuscation algorithms, our obfuscation method can be nested and thereby making it more difficult to reverse engineer. Experimental results indicate that the algorithm performs well against

automated attacks from reverse engineering tools like IDAPro. In comparison with other control flow obfuscation algorithms, our method has better time and space overheads and control flow error.

<pre> .file "factorial.c" .text .globl factorial .type factorial, @function factorial: .LFB0: .cfi_startproc pushq %rbp .cfi_def_cfa_offset 16 .cfi_offset 6, -16 movq %rsp, %rbp .cfi_def_cfa_register 6 subq \$16, %rsp movl %edi, -4(%rbp) cmpl \$0, -4(%rbp) jne .L2 movl %eax, %eax movl %eax, %eax movl \$1, %eax jmp .L3 .L2: movl -4(%rbp), %eax subl \$1, %eax movl %eax, %edi call factorial imull -4(%rbp), %eax .L3: leave .cfi_def_cfa 7, 8 ret .cfi_endproc .LFE0: .size factorial, -factorial .globl main .type main, @function main: .LFB1: .cfi_startproc pushq %rbp .cfi_def_cfa_offset 16 .cfi_offset 6, -16 movq %rsp, %rbp .cfi_def_cfa_register 6 movl \$5, %edi call factorial popq %rbp .cfi_def_cfa 7, 8 ret .cfi_endproc .LFE1: .size main, -main .ident "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3" .section .note.GNU-stack,"",@progbits </pre>	<pre> .file "factorial.c" .text .globl factorial .type factorial, @function .text 0 factorial: .LFB0: .cfi_startproc pushq %rbp .cfi_def_cfa_offset 16 .cfi_offset 6, -16 movq %rsp, %rbp .cfi_def_cfa_register 6 subq \$16, %rsp movl %edi, -4(%rbp) cmpl \$0, -4(%rbp) jne .L2 jz labelJPBMRR jnz labelJPBMRR .byte 10 labelJPBMRR: jmp labelJJXU8 .text 1 labelJJXU8: /* New block to text 1 */ movl %eax, %eax movl %eax, %eax movl \$1, %eax jmp .L3 jmp label22CMNI /* Block ends */ .text 0 label22CMNI: .L2: movl -4(%rbp), %eax subl \$1, %eax movl %eax, %edi call factorial imull -4(%rbp), %eax .L3: leave .cfi_def_cfa 7, 8 ret .cfi_endproc .LFE0: .size factorial, -factorial .globl main .type main, @function .text 1 main: .LFB1: .cfi_startproc pushq %rbp .cfi_def_cfa_offset 16 .cfi_offset 6, -16 movq %rsp, %rbp .cfi_def_cfa_register 6 movl \$5, %edi call factorial popq %rbp .cfi_def_cfa 7, 8 ret .cfi_endproc .LFE1: .size main, -main .ident "GCC: (Ubuntu/Linaro 4.7.3-1ubuntu1) 4.7.3" .section .note.GNU-stack,"",@progbits </pre>
Original Program	Obfuscated Program

Figure 5.12: Assembly representation of factorial program

Chapter 6

Return Oriented Obfuscation

In this chapter we discuss a new function level obfuscation techniques based on return instructions. In this method, each function is split into various function blocks, each ending with a return instruction. The function blocks are independent blocks and can be moved within the program, letting the obfuscator shuffle the function blocks, similar to our function level obfuscation technique. The control flow of the original program is maintained by manipulating the return address in the stack.

The chapter is organized as follows. Section 6.1 discusses the motivation behind designing this algorithm. Section 6.2, discusses the proposed method followed by the algorithm implementation in section 6.3. In section 6.4 we analyse the overhead caused by our obfuscation which is substantiated by experimental analysis in section 6.5. Section 6.6 concludes the chapter.

6.1 Motivation

Control flow obfuscation at the binary/assembly level program mostly tries to manipulate with control flow instruction based on jump instructions, both conditional and unconditional. The main reason for this trend is that, most control flow obfuscation tries to obscure the control flow within a function and the best way to attain is to manipulate the jump instructions. In our function level obfuscation algorithm discussed in chapter 5,

we try to attain inter functional obfuscation by shuffling code fragments across functions. The original control flow was restored by adding jump instructions at the end of the code fragments. These jump instructions helps an attacker to find the original order of the program. To address this issue we wanted to add a more natural ending to the code fragment, which is a return instruction. Our algorithm performs a function level obfuscation by splitting every function into different code segments which works like an independent function, ending with a return instruction. Additional instructions are added in the block to manipulate the return address in the stack. These instructions make sure that the control flows to the correct address location when the return instruction is executed.

6.2 Proposed Method

In this section, our new obfuscation method against software reverse engineering is discussed. Our obfuscation algorithm takes an assembly program as input and split the functions in the program and shuffles them, maintaining the semantics of the program. The assembly representation generated by any assembler maintains the functional structure of the program. The functions in the program are spatially arranged one after another. Each function starts with the standard set of instructions to set the stack and ends with return instructions. When a reverse engineering tool disassembles a binary program to an assembly representation, it is thus capable of identifying functions and could segregate them into different functional units. This can help the reverse engineer, better analyze the program or creating a function call graph [HCS09].

The basic idea of our technique is to disturb this normal representation of the program. In our technique, a function is split into various code fragments by inserting return instructions. Each of these code fragments are then shuffled between functions, giving a inter-functional mix as shown in Fig. 6.1. One of the advantages of this method is that the linear arrangement of functions (one after another) is obscured. One of the challenges

in such an implementation is to maintain the semantics of the program. In a normal program, a return instruction is used to return the control flow from a callee function to a caller function. Adding new return instructions could thus affect the behavior of the function. In this section, we explain in detail about inserting the return instructions into functions while maintaining the semantics of the program.

Figure 6.1: Overview of the algorithm

6.2.1 Splitting the Function

The first step of our algorithm is splitting the function into different segments. The input assembly program is scanned for finding all the functions in the program. Once the functions are identified, each function is split into different code fragments. The obfuscator has the option to specify the number of splits in the function. In the default mode the obfuscator splits each function into four code fragments. While splitting the function into code fragments, our implementation put a constraint that the code fragment should contain at least five instructions.

For each function, the line numbers at which the function has to be split are identified

and a unique label is inserted. The unique label is the reference to the code fragment. The label is randomly generated by the obfuscation algorithm.

Fig. 6.2, shows the insertion of the labels to split the function into different segments. In the example shown, two labels are inserted at the beginning of the code fragments.

Figure 6.2: Inserting labels at the splits

6.2.2 Pushing the Return Address to the Stack

In our algorithm each code fragment will end with a return instruction, which is explained in section 3. To maintain the semantics of the program, proper address location should be stored in the stack. The base pointer register (*ebp*) stores the base address of the frame of the function in the stack. The address location $ebp + 4$ is the location where the return address is stored.

In our obfuscation algorithm, at the end of each code fragment two assembly instructions are added. One instruction backs up the return address in $ebp + 4$ to a register that has not been used. It is followed by another instruction which stores the address of the next code fragment to the stack location $ebp + 4$.

In the example shown in Fig. 6.3, register `edx` is used to store the current return address in the stack. The address location `split_label_2` is then stored in the stack location `ebp + 4`. These two instructions can be stored anywhere between `split_label_1` and `split_label_2` and not necessarily at the end of the code fragment.

Figure 6.3: Inserting return address in the stack

6.2.3 Inserting Return Instruction

Once inserting the instructions for placing the desired address locations as return points in the stack, the obfuscation algorithm inserts `ret` instruction at the end of each code fragment. Like a standard return instruction the stack pointer and base pointer are reset using the two instructions, `move sp, ebp` and `pop ebp` which is then followed by the `ret` instruction.

We add an extra instruction to back up the stack pointer value to a free register. In the example shown in Fig. 6.4, the instruction is `mov ecx, esp`, is used to store the value of stack pointer to the register `ecx`. The reason for this instruction is that we cannot reset the stack pointer value as the function is not completely returning to its caller function.

So, the stack pointer value should be retained for the following code fragments of the same function. With the address location *split_label_2* in stack and return address, the control flows from the first code fragment to *split_label_2*, the beginning of the second code fragment when the *ret* instruction gets executed.

Figure 6.4: Inersting return instruction

6.2.4 Restoring the Stack

During the execution of the program, after *ret* instruction from one code fragment is executed, the program returns to the next code fragment. Since the stack pointer gets reset during the return instruction, the stack has to be restored.

The original return address of the function has to be restored, which is stored in the register *edx*. By pushing the register *edx*, we can restore the original return address in the stack. Instructions *push ebp*, and *mov ebp, esp* restores the *ebp* register. The original stack pointer value is stored in *ecx* register as shown in the example in Fig. 6.4. Instruction *mov esp, ecx*, restores the original stack pointer value.

6.2.5 Shuffling the Code Fragments

The obfuscation algorithm treats each code fragment of all the function as a separate item and then shuffles them randomly. The linear order of the function representation is disturbed and code fragments from different functions will be interleaved together. This helps in inter-functional control flow obfuscation.

In the example shown in Fig. 6.5, the code fragments in the functions A and B are shuffled together. To a reverse engineer it looks like small blocks of functions because of the return at the end of the code fragment. The succeeding code fragment has no relation to the previous code fragment because of the shuffling and it will confuse the reverse engineer.

Fig. 6.6 shows the function call graph of *nqueens* program generated by IDAPro, before and after obfuscation. The obfuscation has clearly confused the IDAPro that it is unable to generate the function calls from main after obfuscation. Fig. 6.7 shows the disassembled *main* function of the program before and after obfuscation. It is clear from the figure that the control flow of the function is completely obscured by the obfuscation.

6.3 Implementation

The proposed obfuscation method is implemented in python programming language. Our implementation expects assembly level representation of the program to be obfuscated. We have implemented the obfuscation algorithm for Microsoft Windows XP and Ubuntu

Figure 6.5: Shuffling of code fragments

Figure 6.6: Function call graph of nqueens program

Linux 13.04 operating systems. Our implementation accepts Microsoft Visual Studio 10.0 generated assembly programs and assembly program generated by gcc 4.7.3 as in-

Figure 6.7: Main function of nqueens program

put for obfuscation. Our algorithm generates the obfuscated assembly program which is assembled using the corresponding assembly program to generate obfuscated binary program.

The python code copies the input assembly program to a buffer. It analyzes the buffer to find the functions and the start and end of the functions. The number of code fragments for each function is calculated according to the size of the function. The instructions to modify the stack for return address and restoring the stack pointer are added to the beginning and end of each code fragments. The line numbers of beginning and end of each code fragments are changed due to the insertion of instructions. The code

fragments are given numbers in sequential order and are represented in a data structure with the number, starting line in the buffer and ending line in the buffer. A simple shuffling algorithm is used to shuffle the code fragments as shown in Fig. 6.8. The code fragments are then stored into a new file in the shuffled order to generate the obfuscated assembly file.

Shuffle_code_fragments (Code_Fragment_list [])

```
L = Length (Code_Fragment_list)
while (L > 1)
    R = Random (1, L-1)
    Temp = Code_Fragment_List [R]
    Code_Fragment_List [R] = Code_Fragment_List [L]
    Code_Fragment_List [L] = Temp
    L = L -1
Return Code_Fragment_List
```

Figure 6.8: Shuffling algorithm

6.4 Overhead Analysis

The insertions of the new instructions have significant effect on the size of the program and the time complexity of the program. In this section we will discuss, how the space and time complexity of the program is affected by applying the proposed obfuscation.

6.4.1 Space Overhead

If a function is split into 2 code fragments, then 10 new instructions are added into the program and 20 instructions are added if the function is split into 3 code fragments and so

on. Let the number of instructions in the program be N_{before} and the original program is split into $n+1$ code fragments. The total number of instructions in the obfuscated program will be,

$$N_{after} = N_{before} + 10n \quad (\text{Eq. 6.1})$$

In the worst case, the entire program is split into code fragments with 5 instructions. Let the program is split into $n+1$ code fragments, with each code fragment having five instructions, then the total number of instructions in the program before obfuscation is ,

$$N_{before} = 5(n + 1) \quad (\text{Eq. 6.2})$$

After the obfuscation, 10 instructions are added per code fragment and the total number of instructions in the program after obfuscation is,

$$N_{after} = 5(n + 1) + 10n \quad (\text{Eq. 6.3})$$

$$N_{after} = 3N_{before} - 10 \quad (\text{Eq. 6.4})$$

So, in the worst case there are three times more instructions in the obfuscated program than the original program. We will see in the experimental evaluation that this upper bound is held.

6.4.2 Time Overhead

We know that the size of the program increases by 3 times in the worst case. The time complexity of a program $T(n)$ refers to the order in which the instructions in the program are run.

The new instructions added to the program during the obfuscations are *mov*, *push*, *pop* and *ret*. The *ret* instruction acts like a jump to the next instruction. The execution of these instructions does not create more time overhead than the execution of that single instruction unlike some instructions like a recursive call instruction.. Hence the time complexity of the program remains the same. However, the number of instructions that get executed is higher. Hence, in the worst case the number of instructions that get executed is 3 times the original number of instructions.

$T(N)$ = Time complexity of the program

a = Number of instructions in the program

x = Number of instructions in the obfuscated program

Number of instruction execution in the original program = $aT(N)$

Number of instruction execution in the obfuscated program = $xT(N)$

In worst case, $x = 3a$

Then,

Number of instruction execution in the obfuscated program = $3aT(N)$

The time complexity of the obfuscated program is thus between $T(N)$ and $3T(N)$.

6.5 Performance Evaluation

In this section we perform experimental evaluation of our algorithm against reverse engineering. We use IDAPro [\[Hex\]](#) to measure the potency of our obfuscation. We measure instruction disassembly error to evaluate the potency of our obfuscation. The number of instructions that the reverse engineering tool is unable to disassemble properly is measured by the instruction disassembly error. The number of instructions that is successfully disassembled decreases as the obfuscation improves. In this section, we also analyze the space and time overhead caused by the obfuscation. The increase in space

and time at different levels of obfuscation is analyzed. We used the test programs from the lcc 4.2 [Frab] compiler source as input test programs for our obfuscation algorithm.

6.5.1 Instruction Disassembly Error

The potency of the obfuscation algorithm against reverse engineering tool, is measured by the error in the disassembly of assembly instructions. IDAPro [Hex] was used to disassemble the obfuscated test programs. We measured the total number of instructions in the original program and the instructions recognized by IDAPro [Hex] after reverse engineering the obfuscated program. Confusion factor is then calculated as the ratio of their differences, as defined by the following equation,

$$CF_{instr} = \frac{|T_{total} - T_{disasm}|}{T_{total}} \quad (\text{Eq. 6.5})$$

The total number of instruction addresses before obfuscation is represented by T_{total} . The number of instruction addresses recognized by IDAPro [Hex] after disassembling the obfuscated binary program is represented by T_{disasm} .

Table 6.1 shows the confusion factor while disassembling obfuscated test programs by IDAPro [Hex]. The table shows varies levels of splitting the program. The first column represented by zero splits is the original program and all the instructions are reverse engineered successfully. Column 2 represents the obfuscated program, where every function is split into two and the mean disassembly error is 55.9% when functions are split into two. The instruction disassembly error increases as the splitting of the program increases. The splitting of a program saturates after a while. For instance, the program *fields* has the same instruction disassembly error for 16 splits and 32 splits. This is because the program is split to the maximum possible split by 16 splits and the program cannot be further split down.

Mean instruction disassembly error of 85.16% is obtained at level 8, where each function is split into 8 code fragments.

Table 6.1: Instruction disassembly error

ProgSplits	0	2	4	8	16	32	64	128
8q	283	125	65	42	25	25	25	25
array	341	150	78	51	31	20	20	20
cf	184	81	42	28	17	11	11	11
cq	13786	6066	2068	2068	2068	2068	2068	2068
cvt	674	297	155	101	61	40	27	14
fields	339	149	78	51	20	20	20	20
incr	392	172	90	24	24	24	24	24
init	415	183	95	62	37	17	17	17
limits	162	71	37	24	12	12	12	12
sort	506	223	116	76	46	46	46	46
spill	433	191	100	65	39	26	26	26
struct	505	222	116	76	45	36	30	30
wfl	597	263	137	90	54	24	24	24
CF_{instr}		55.9%	82.9%	85.1%	86.6%	87.2%	87.3%	87.4%

6.5.2 Function Level Obscurity

We use the same cyclomatic complexity[McC76] used in our previous method to measure the inter functional obscurity. The functional level obscurity is measured with 4 levels of splitting.

The complexity M, is defined as follows

$$M = E - N + 2 \quad (\text{Eq. 6.6})$$

E = the number of the edges in the graph

N = the number of the nodes in the graph

Table 6.2 shows the cyclomatic complexity measure of the test programs. For an average of 166 nodes the cyclomatic complexity after obfuscation is 540, which is 3.25 times the number of nodes. The measure is better than our previous function level

Table 6.2: Function Level Obscurity

Programs	Number of functions(N)	Edges(E)	E-N+2
8q	3	14	13
array	3	12	11
cf	2	4	4
cq	106	438	334
cvt	2	4	4
fields	6	28	24
incr	5	20	17
init	9	36	29
limits	2	4	4
sort	7	48	43
spill	6	38	34
struct	8	32	26
wfl	7	26	21
Mean	166	704	540

obfuscation where 412 was the measure obtained from 209 nodes which is 1.96 times the number of nodes.

6.5.3 Space Overhead

Obfuscation will have effect on the size of the program. $Space_{ovh}$ defines the increase in the size of the program.

$$Space_{ovh} = \frac{Space_{after}}{Space_{before}} \quad (\text{Eq. 6.7})$$

In Table 6.3, we show how the program size increase as the program is obfuscated. The size of the program increases as the number of splits increase. In the worst case, the size increases to 2.2 times the original size (for program *array*). But on an average, the program size increases by 1.57 times the original size with 128 splits. So, even in

Table 6.3: Space Overhead

ProgSplits	0	2	4	8	16	32	64	128
8q	8743	8779	10256	13586	17843	17843	17843	17843
array	8678	8714	11324	14321	18545	18923	18923	18923
cf	8750	8786	9957	12985	13876	14178	14178	14178
cq	63599	75898	85699	85699	85699	85699	85699	85699
cvt	12983	13019	13091	13235	13523	14099	15251	21651
fields	8728	8764	12633	13589	13589	13589	13589	13589
incr	8594	8630	9987	12371	12371	12371	12371	12371
init	9134	9170	9242	9386	9674	9782	9782	9782
limits	8522	8558	8630	8774	12062	13458	13458	13458
sort	8846	8882	11954	12098	14350	14350	14350	14350
spill	8863	8899	10971	12115	16403	18799	18799	18799
struct	8843	8879	8951	9095	9383	14055	17127	17127
wf1	13179	13215	13287	17521	18754	21875	21875	21875
Mean	177462	190193	215982	234775	256072	269021	273245	279645
Space _{ovh}	1	1.07	1.21	1.32	1.44	1.51	1.53	1.57

the worst case the size is less than the theoretical upper bound, which is three times the original size.

6.5.4 Time Overhead

Obfuscation does have an effect on the execution time. The execution time of the program will increase, because of the execution of the additional instructions. The return instructions added in the program and the instructions for modifying and resetting the stack are the major reasons for the execution time overhead.

$Time_{before}$ refers to the time taken by the program to execute without obfuscation and $Time_{after}$ is the time taken by the obfuscated program to execute. We evaluate the

Table 6.4: Time Overhead

ProgSplits	0	2	4	8	16	32	64	128
8q	1300	1395	1534	1975	2589	2589	2589	2589
cf	2777	3276	3588	4215	5014	5581	5581	5581
cvt	1077	1121	1154	1178	1223	1498	1892	2548
fields	1089	1258	1685	1987	2406	2406	2406	2406
incr	1271	1301	1354	1537	1537	1537	1537	1537
spill	1174	1256	1325	1456	1658	2219	2219	2219
struct	1342	1398	1469	1566	1689	1726	1754	1754
wf1	1245	1322	1391	1499	1785	1997	2015	2015
array	1011	1250	1347	1678	2001	2022	2500	2694
cq	1025	1119	1243	1567	1874	2198	2198	2198
init	1198	1245	1376	1461	1653	1985	1985	1985
sort	1037	1134	1256	1370	1523	1523	1523	1523
Mean	15546	17075	18722	21489	24952	27281	28199	29049
Time _{ovh}	1	1.09	1.20	1.38	1.60	1.75	1.84	1.86

effect of obfuscation on execution speed with $Time_{ovh}$ defined as,

$$Time_{ovh} = \frac{Time_{after}}{Time_{before}} \quad (\text{Eq. 6.8})$$

We used *lcc* test programs for testing the time overhead. Input to the test programs are the input data provided by the *lcc* source [Frab].

Table 6.4, shows the time overhead caused by our obfuscation on various binary programs. In the worst case the time overhead is 2.36 times in the case of *cvt* with 128 levels of obfuscation. On an average the worst case time overhead is 1.86. According to our analysis in section 4.2 the worst case upper bound for time overhead is 3 times the original time overhead.

6.5.5 Time Overhead with Varying Input

We ran bubble sort and linear search program for different inputs. We used arrays of size 10000 to 100000 for linear search program. For bubble sort we used input array size varying from 1000 to 10000. We obfuscated the programs with 8 splits. We ran both the original and obfuscated programs and got the results as shown in Fig. 6.9 and Fig. 6.10. The result is similar to our previous results but we kept the upper bound as $3T(n)$. The time overhead of the obfuscated program is sandwiched between $T(n)$ and $3T(n)$.

Figure 6.9: Time complexity of linear search program

6.6 Obfuscation of Factorial Program

Fig. 6.11 shows the original and the obfuscated assembly representation of factorial program. For simplicity the function is split only at one point, dividing the function *factorial* into two parts.

Figure 6.10: Time complexity of bubble sort program

6.7 Conclusion

In this chapter we proposed an obfuscation algorithm to perform inter functional obfuscation. Our method slices each function in the program into separate code fragments. Each fragment ends with a return instruction and starts with stack allocation instructions, thereby appearing itself like a function. The return instruction transfers the control flow to the next code fragment instead of returning to a caller function. The code fragments are shuffled disturbing the linear order of the functions. Unlike other control flow obfuscation, our method adds more control flow instructions (return instruction) to increase the control flow obscurity instead of removing the control flow instructions. The experimental results show that obfuscating with 8 splits provides a good obfuscation without too much overhead on the space and time requirements of the program. Experimental analysis shows that that our method has an instruction disassembly error of 85.1 % with 8 levels of splitting. An average time overhead of 1.38 and space overhead of 1.32 is observed while obfuscating with 8 splits.

<pre> .file "factorial.c" .text .globl factorial .type factorial, @function factorial: .LFB0: pushq %rbp .LCFI0: movq %rsp, %rbp .LCFI1: subq \$16, %rsp movl %edi, -4(%rbp) cmpl \$0, -4(%rbp) jne .L2 movl \$1, %eax jmp .L3 .L2: movl -4(%rbp), %eax subl \$1, %eax movl %eax, %edi call factorial imull -4(%rbp), %eax .L3: leave .LCFI2: ret .LFE0: </pre>	<pre> .file "factorial.c" .text .globl factorial .type factorial, @function factorial: .LFB0: pushq %rbp .LCFI0: movq %rsp, %rbp .LCFI1: subq \$16, %rsp movl %edi, -4(%rbp) cmpl \$0, -4(%rbp) jne .L2 movl \$1, %eax jmp .L3 #Obfuscation Additions pushq %rdx movq 8(%rbp), %rdx movq \$label0R8OXE+0, 8(%rbp) pushq %rcx movq %rsp, %rcx movq %rbp, %rsp popq %rbp ret #Split the function here label0R8OXE: pushq %rdx pushq %rbp movq %rsp, %rbp movq %rcx, %rsp popq %rcx popq %rdx .L2: movl -4(%rbp), %eax subl \$1, %eax movl %eax, %edi call factorial imull -4(%rbp), %eax .L3: leave .LCFI2: ret .LFE0: </pre>
Original Program	Obfuscated Program

Figure 6.11: Assembly representation of factorial program

Chapter 7

Contributions and Future Work

Software protection techniques at various points of software lifetime have been discussed in chapter 2. Different types of software obfuscation and techniques and algorithms in each type are also discussed in chapter 2. Many obfuscation techniques focus on control flow obfuscation to increase the potency of obfuscation against reverse engineering. Another advantage of control flow obfuscation is that if the control flow of a program is obscured then it will be hard for performing data flow analysis. The basic methodology of the control flow obfuscation algorithms is to remove the control flow instructions from the program and attaining the control flow back through unconventional methods.

Signals [PDA07], light weight process [GCT05] and self modifying codes [MAM⁺06] [SE10] are some of them. Even though they provide good potency against static reverse engineering attacks, the control flow addresses are available in the additional code added to the program for obfuscation. One of our first concerns is to remove this control flow information from the code area. Moving the control flow information onto stack and making it available only during runtime when the function is getting executed is one of our major contributions.

7.1 Our Contributions

In this section we will discuss about the four obfuscation algorithms that we devised as a part of our research.

7.1.1 Software Obfuscation by Hiding Control Flow Information in Stack

In this method we proposed a new software obfuscation algorithm based on self modifying code and using stack to conceal the control flow information. The advantage of using stack is that it is dynamically allocated when a function is initiated. So statically it is very difficult for an attacker to extract the control flow information.

The basic method is to replace the control flow instructions like jump instructions from the program by normal instructions. The target address of the control flow instruction is stored in the stack as local variables. Reconstruction instructions are added in the program to reconstruct the control flow instructions during runtime with the help of target addresses from stack.

One of the advantages of this algorithm is that the target addresses of the control flow instructions are not available in the code area. It is in the volatile stack and acquiring the addresses through static analysis is difficult.

We have computed the efficiency of our algorithm against static and dynamic analysis. The performance of our algorithm against automated reverse engineering tools is better than the signals based obfuscation[PDA07] and self modifying code obfuscation [SE10]. An average instruction disassembly error of 79.78 %, control flow error of 61.35 % is attained with a cost in time of 1.29 and space of 2.22.

One of the shortcomings of this method is that, even though we managed to move the target addresses to data area from code area, it is represented in its raw form. In the next algorithm we improve this by encrypting the target addresses stored in the data area.

7.1.2 Software Protection by Obfuscation and Encryption

In this method we proposed a method where we used encryption and obfuscation hand in hand to improve the security of software.

Similar to the previous method we used a self modifying code approach. The control flow instructions like jump instructions are replaced by normal instructions. The control flow instructions are then reconstructed dynamically at runtime with the help of reconstruction instructions embedded in the program.

The major difference is in the storing of the target addresses of the control flow instructions. Instead of storing the control flow address in stack, in this method it is stored in the static data area in encrypted form. The decryption happens only during runtime. This makes it harder for the attacker to retrieve the target address from the data area. In the stack based method the target addresses of obfuscated control flow instructions of a function are readily available in its raw form in stack when the function starts executing. The window period is lesser in this new method as the target address is decrypted just before the reconstruction instructions are executed.

The performance of this algorithm is good with average instruction disassembly error of 78.2 % which is similar to stack based obfuscation and a better control flow error of 63.1 % is attained with a cost in time of 1.26 and space of 1.04. The time and space overhead of this method is lesser than the stack based method, even though they are comparable. The major advantage of this method is that the target addresses required to restore the control flow are stored in encrypted form.

7.1.3 Function Level Control Flow Obfuscation

In this method we developed a new approach which obfuscates the control flow across functions. In this method code fragments from each function is stripped from the original function and is stored in another function. Each function will be having code fragments

from different functions, thereby creating a function level shuffled version of the original program. Control flow is obscured between and within the function by this method.

The major advantage of this method is that it limits the visibility of the functions and the linear order in which they are stored. Code fragments from different functions will be shuffled and interleaved. Another advantage of this function is its ability to perform nested obfuscation.

The performance of this algorithm is good with average instruction disassembly error 78.3 % similar to stack based obfuscation and a better control flow error of 69.1 % is attained with a cost in time of 1.21 and space of 1.03.

7.1.4 Return Oriented Obfuscation

In this method, we discuss a new function level obfuscation techniques based on return instructions. In this technique, each function is split into various function blocks, each ending with a return instruction. The function blocks are independent blocks and can be moved within the program, letting the obfuscator shuffle the function blocks, similar to our function level obfuscation technique. The control flow of the original program is maintained by manipulating the return address in the stack.

The major advantage of this method over our previous inter functional obfuscation is that each code fragment will be seen as a function. Another advantage is that no new jump instructions are added to the program to maintain the semantics. The control flow is maintained by manipulating the return addresses in the stack.

Experimental analysis shows that that our method has an instruction disassembly error of 85.1 % with 8 levels of splitting. An average time overhead of 1.68 and space overhead of 1.32 is observed while obfuscating with 8 splits per function.

7.2 Future Work

On the basis of our current works and the literature reviewed we outline the future work directions.

7.2.1 Obfuscation for Distributed Programs

A research area of interest is to develop obfuscation algorithm for distributed programs. Many mobile phone applications running on smartphones with internet connections are distributed in nature. A client application will be running in the mobile phone constantly querying the server application for data. Devising obfuscation algorithms which take advantage of the features of distributed systems to generate potent obfuscations against static and dynamic attacks is the aim.

One of the works in this direction is the obfuscation of distributed programs using opaque predicates [MT06], where opaque predicates are used to obtain control flow obfuscation of the program. The truth value of the opaque predicate is calculated by different processes through communications. The advantage of this method is that it is a NP-hard problem to find the value of the opaque predicate statically. A disadvantage of the method is that this method is not potent against dynamic analysis. Some of the directions discussed below are possible methods to create obfuscated distributed programs.

7.2.1.1 Self Modifying Code

From the examples of obfuscations from non-distributed programs based on self modifying code, we know that they provide good performance against static and dynamic analysis [SE10] [MAM⁺06]. An added advantage of using self modifying code in distributed systems is that the features of distributed systems can also be used for attaining self modification.

One of the methods that can be used is the following. The control flow of a program can be obfuscated by camouflaging the control flow instructions with normal instructions as we discussed in chapter 3 and 4. These camouflaged instructions can be reconstructed during runtime using a different process in a different system.

The major challenge in this method is the time overhead incurred due to communications between processes. Since, a different process is responsible for the reconstruction of camouflaged instruction in a program, there should be communication between processes even for normal execution. This may slow down the program execution speed.

7.2.1.2 Encryption

Another modification that can be done is to incorporate encryption as discussed in chapter 4. The decryption algorithm for vital information can be stored in a different process and there by an attacker with only one program in hand cannot analyse the decryption algorithm.

One of the limitations of our obfuscation based on encryption technique is that it stores the decryption keys in the program itself. In a distributed architecture the keys can be stored in the server. The obfuscated client program can request for the decrypted data to the server. Another method is to store the entire encryption/decryption module at the server side. The advantage of such a method is that the attacker who is having access only to the client program cannot know what encryption algorithms are used in the obfuscation.

One of the challenges in this method is to camouflage the communication of the decryption key or decrypted data as normal communication. Another challenge would be the time overhead involved as there should be a client-server communication for an encryption/decryption.

7.2.2 Malware Detection

One of the characteristics of obfuscation is that it generates syntactically different but semantically equivalent variants of a program. This technology may be used by malware creators to create multiple versions of the same malware [HCS09].

The new variants of existing malware make the job of malware detection softwares difficult. One of the methods against this is discussed by Xin et al. in [HCS09]. The method used is indexing the known malwares using function call graphs. The graphs are based on the function calls made by the program. When a new program is suspected to be a malware variant, then it is also represented using function call graphs and compared with the indexed malwares.

One of the assumptions in this method is that obfuscations will not affect function calls in a program. But some of the obfuscation algorithms like signal based approach [PDA07] and self modifying based approach [SE10] also obfuscates function calls, along with other control flow instructions. Hence, it is an interesting research area to explore other features of software programs that is invariant to obfuscation.

List of Publications

V. Balachandran and S. Emmanuel, "Potent and Stealthy Control Flow Obfuscation by Stack Based Self-Modifying Code," *IEEE Transactions on Information Forensics and Security*, vol.8, no.4, pp.669,681, April 2013.

V. Balachandran, S. Emmanuel and N. W. Keong, "Obfuscation by Code Fragmentation to Evade Reverse Engineering," *IEEE International Conference on Systems, Man, and Cybernetics*, San Diego, USA, October 2014.

V. Balachandran, N. W. Keong and S. Emmanuel, "Function Level Control Flow Obfuscation for Software Security," *International Conference on Complex, Intelligent and Software Intensive Systems*, Birmingham, UK, July 2014.

V. Balachandran and S. Emmanuel, "Software Code Obfuscation by Hiding Control Flow Information in Stack," *IEEE International Workshop on Information Forensics and Security*, WIFS, Brazil, Dec. 2011.

V. Balachandran and S. Emmanuel, "Software Protection with Obfuscation and Encryption," *Information Security Practices and Experiences Conference*, ISPEC, Lanzhou, China, May 2013.

References

- [ABG12] M. Alkhalaf, T. Bultan, and J. L. Gallegos. Verifying client-side input validation functions using string analysis. In *34th International Conference on Software Engineering (ICSE)*, pages 947–957, June 2012.
- [AK08] A.B. Ashfaq and S.A Khayam. Performance comparison of four anomaly detectors in detecting self-propagating malware on endpoints. In *International Symposium on Biometrics and Security Technologies, 2008. ISBAST 2008*, pages 1–9, April 2008.
- [ALD] ALD. Assembly language debugger. <http://ald.sourceforge.net/>.
- [ALSU06] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2 edition, 2006.
- [Auc96] D. Aucsmith. Tamper-resistant Software: An Implementation. *Proceedings of Information Hiding: First International Workshop*, 1174:371–333, 1996.
- [Bar05] M. Bartoletti. Language-based Security: Access Control and Static Analysis. In *Ph.D. dissertation, University of Pisa*, 2005.
- [BE13] V. Balachandran and S. Emmanuel. Potent and stealthy control flow obfuscation by stack based self-modifying code. *IEEE Transactions on Information Forensics and Security*, 8(4):669–681, April 2013.
- [BGI⁺12] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *J. ACM*, 59(2):6:1–6:48, May 2012.

REFERENCES

- [Bis03] M. Bishop. *Computer Security: Art and Science*. MIT Press, 2003.
- [BS03] B. Brykczynski and R. A. Small. Reducing internet-based intrusions: Effective security patch management. *IEEE Software*, 20(1):50–57, Jan 2003.
- [CFS⁺09] Xiaojiang Chen, Dingyi Fang, JingBo Shen, Feng Chen, WenBo Wang, and Lu He. A dynamic graph watermark scheme of tamper resistance. In *Information Assurance and Security, 2009. IAS '09. Fifth International Conference on*, volume 1, pages 3–6, Aug 2009.
- [CG95] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs . *Software: Practice and Experience*, 25:811–829, 1995.
- [CJ03] M. Christodorescu and S. Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proc. of Conference on USENIX Security Symposium*, pages 12–12, 2003.
- [CKY09] H. Chang, H-J. Kwon, and I. You. The study on software tamper resistance for securing a game service. In *International Conference on Complex, Intelligent and Software Intensive Systems, 2009. CISIS '09*, pages 555–558, March 2009.
- [CLRS] T. H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 3 edition.
- [Coh93] F. Cohen. Operating System Protection Through Program Evolution. In *Computers and Security*, pages 565–584, 1993.
- [Cor] Microsoft Corporation. Dumpbin. <http://sourceware.org/binutils/docs/binutils/objdump.html>.
- [CP06] G. Canfora and M.D. Penta. New Frontiers of Reverse Engineering. In *Proc. of Future of Software Engineering*, pages 326–341. IEEE Computer Society, 2006.

REFERENCES

- [CPN⁺09] M. Ceccato, M. Di Penta, J. Nagra, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella. The effectiveness of source code obfuscation: An experimental assessment. In *IEEE 17th International Conference on Program Comprehension, ICPC '09*, pages 178–187, may 2009.
- [CTL] C. Collberg, C. Thomborson, and D. Low. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. *Conference Record of the ACM Symposium on Principles of Programming Languages*, 5.
- [CTL97] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. *Technical Report 148, University of Auckland*, 1997.
- [CTT05] Chuan-Wen Chang, Dwen-Ren Tsai, and Jui-Mi Tsai. A cross-site patch management model and architecture design for large scale heterogeneous environment. In *Security Technology, 2005. CCST '05. 39th Annual 2005 International Carnahan Conference on*, pages 41–46, oct. 2005.
- [DL13] D. Dunaevand and L. Lengyel. Aspects of intermediate level obfuscation. In *Engineering of Computer Based Systems (ECBS-EERC), 2013 3rd Eastern European Regional Conference on the*, pages 138–142, Aug 2013.
- [DLM⁺78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, November 1978.
- [DLOa] Reverse Engineering Digital Law Online. Atari v. nintendo. <http://digital-law-online.info/lpdi1.0/treatise25.html>.
- [DLOb] Reverse Engineering Digital Law Online. Sega v. accolad. <http://digital-law-online.info/cases/24PQ2D1561.html>.
- [EAAT04] D. Ellis, J. Aiken, K. Attwood, and S. Tenaglia. A Behavioral Approach to Worm Detection. In *Proc. of the 2004 ACM Workshop on Rapid Malcode*, pages 43–53, 2004.

REFERENCES

- [EC05] E. Eilam and E. J. Chikofsky. *Reversing: Secrets of Reverse Engineering*. John Wiley and Sons, 2005.
- [ED] England and Wales High Court (Chancery Division) Decisions. Sas v. world programming. <http://www.bailii.org/ew/cases/EWHC/Ch/2013/69.html>.
- [FAA97] S. Forrest, A. Somayaji, and D.H. Ackley. Building Diverse Computer Systems. In *Proc. of Hot Topics in Operating Systems*, pages 67–72, 1997.
- [Fos05] J. C. Foster. *Buffer Overflow Attacks: Detect, Exploit, Prevent*. Syngress, 2005.
- [Fraa] C. W. Fraser. Copt. <https://sites.google.com/site/lccretargetablecompiler/downloads>.
- [Frab] C. W. Fraser. lcc, a retargetable compiler for ansi c. <https://sites.google.com/site/lccretargetablecompiler/downloads>.
- [FSR09] M. Feily, A. Shahrestani, and S. Ramadass. A survey of botnet and botnet detection. In *Third International Conference on Emerging Security Information, Systems and Technologies, 2009. SECURWARE '09.*, pages 268–273, June 2009.
- [GCT05] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. Control flow based obfuscation. In *Proceedings of the 5th ACM Workshop on Digital Rights Management, DRM '05*, pages 83–92, New York, NY, USA, 2005. ACM.
- [GHD10] S. Ghosh, J. Hiser, and J. Davidson. A secure and robust approach to software tamper resistance. In *Information Hiding*, volume 6387 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin / Heidelberg, 2010.
- [GNU] GNU. Objdump, gnu manuals onliner. <http://sourceware.org/binutils/docs/binutils/objdump.html>.

REFERENCES

- [GZ10] Xiang Guangli and Cai Zheng. The code obfuscation technology based on class combination. In *Distributed Computing and Applications to Business Engineering and Science (DCABES), 2010 Ninth International Symposium on*, pages 479–483, Aug 2010.
- [Han03] D. Hanchez. A Comparative Study of Software Protection Tools Suited for E-Commerce with Contributions to Software Watermarking and Smart Cards. In *Ph.D. dissertation, Universite Catholique de Louvain*, 2003.
- [HCS09] X. Hu, T. Chiueh, and K. G. Shin. Large-scale Malware Indexing using Function-call Graphs. In *Proc. of ACM conference on Computer and communications security*, pages 611–620, 2009.
- [HCT06] T. W. Hou, H. Y. Chen, and M. H. Tsai. Three control flow obfuscation methods for java software. *IEE Proceedings - Software*, 153(2):80 – 86, april 2006.
- [Hei11] D. Heimbigner. A tamper-resistant programming language system. *IEEE Transactions on Dependable and Secure Computing*, 8(2):194–206, March 2011.
- [Hex] Hexrays. Idapro. <http://www.hex-rays.com/>.
- [HNL⁺13] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–11, Feb 2013.
- [Hwa] K. Hwang. Blizzard versus bentd: A looming ice age for free software development? <http://www.lawschool.cornell.edu/research/cornell-law-review/upload/crn504hwang.pdf>.
- [IKA95] K. Ilgun, R. A. Kemmerer, and Porras P. A. State Transition Analysis: A Rule-based Intrusion Detection Approach. *IEEE Transactions on Software Engineering*, 1995.

REFERENCES

- [JS12] R. Johari and P. Sharma. A survey on web application vulnerabilities (sqlia, xss) exploitation and security engine for sql injection. In *2012 International Conference on Communication Systems and Network Technologies (CSNT)*, pages 453–458, May 2012.
- [KM14] A. Kulkarni and R. Metta. A new code obfuscation scheme for software protection. In *Service Oriented System Engineering (SOSE), 2014 IEEE 8th International Symposium on*, pages 409–414, April 2014.
- [KMNm03] Y. Kanzaki, A. Monden, M. Nakamura, and K. Matsumoto. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proc. of ACM conference on Computer and communications security*, pages 290–299, 2003.
- [KP11] D. A. Kindy and A-S. K. Pathan. A survey on sql injection: Vulnerabilities, attacks, and prevention techniques. In *2011 IEEE 15th International Symposium on Consumer Electronics (ISCE)*, pages 468–471, June 2011.
- [KP12] P. Kumar and R. K. Pateriya. A survey on sql injection attacks, detection and prevention techniques. In *2012 Third International Conference on Computing Communication Networking Technologies (ICCCNT)*, pages 1–5, July 2012.
- [Kus] D. Kushner. The real story of stuxnet. [http://http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet](http://spectrum.ieee.org/telecom/security/the-real-story-of-stuxnet).
- [LCM⁺00] D. Lie, C.Thekkath, M.Mitchell, P.Lincoln, D. Boneh, J.Mitchell, and M.Horowitz. Tamper-resistant Software: An Implementation. In *Proc. of 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–177, 2000.
- [LD03a] C. Linn and S. Debray. Exploiting Self-modification Mechanism for Program Protection. In *Proc. of Computer Software and Applications Conference*, pages 170–179, 2003.

REFERENCES

- [LD03b] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.
- [LKK10] B. Lee, Y. Kim, and J. Kim. binOb+: a framework for potent and stealthy binary obfuscation. In *Proc. of ACM Symposium on Information, Computer and Communications Security*, pages 271–281, 2010.
- [LKR09] S. Liu, R. Kuhn, and H. Rossman. Surviving insecure it: Effective patch management. *IT Professional*, 11(2):49–51, March 2009.
- [LSNS03] Q. Liu, R. Safavi-Naini, and N. P. Sheppard. Digital Rights Management for Content Distribution. In *Proc. Australasian Information Security Workshop Conference on ACSW Frontiers*, pages 49–58, 2003.
- [LT08] H. Lui and H. B. K. Tan. Testing Input Validation in Web Applications through Automated Model Recovery. *The Journal of Systems and Software*, pages 222–233, 2008.
- [LWC⁺14] Kam-Yiu Lam, Jiantao Wang, Yuan-Hao Chang, Jen-Wei Hsieh, Po-Chun Huang, Chung Keung Poon, and Chun Jiang Zhu. Garbage collection for multi-version index on flash memory. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, pages 1–4, March 2014.
- [MAM⁺06] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere. Software Protection through Dynamic Code Mutation . *Information Security Applications*, pages 194–206, 2006.
- [McC76] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, Dec 1976.
- [MRB02] J. Maebe, M. Ronsse, and K. De Bosschere. Diota: Dynamic instrumentation, optimization and transformation of applications. In *In Proc. of 4th Workshop on Binary Translation*, 2002.

REFERENCES

- [MT06] A. Majumdar and C. Thomborson. Manufacturing Opaque Predicates in Distributed Systems for Code Obfuscation. In *Proc. of Australasian Computer Science Conference*, pages 187–196, 2006.
- [MuAMM06] J. M Memon, Shams ul Arfeen, A. Mughal, and F. Memon. Preventing reverse engineering threat in java using byte code obfuscation techniques. In *International Conference on Emerging Technologies, 2006. ICET '06*, pages 689 –694, nov. 2006.
- [New] PR Newswire. Sony v. connectix. <http://www.prnewswire.com/news-releases/siia-les-six-new-software-piracy-lawsuits-against-fraudulent-online-vendors-across-the-country-69854267.htm>.
- [NS] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. *ACM SIGPLAN Notices*, 42.
- [Oll] OllyDbg. Ollydbg. <http://www.ollydbg.de/>.
- [oQ] The University of Queensland. The dcc decompiler. <http://itee.uq.edu.au/cristina/dcc.html>.
- [OWAa] OWASP. Command injection. <https://www.owasp.org/index.php/CommandInjection>.
- [OWAb] OWASP. The open web application security project. <https://www.owasp.org/index.php/MainPage>.
- [PDA07] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary Obfuscation using Signals . In *Proc. of USENIX Security Symposium on USENIX Security Symposium*, pages 19–25, 2007.
- [PG05] M. Dalla Preda and R. Giacobazzi. Control code obfuscation by abstract interpretation. In *Third IEEE International Conference on Software Engineering and Formal Methods, 2005. SEFM 2005.*, pages 301 – 310, sept. 2005.

REFERENCES

- [PPSS14] A. Povyakalo P. Popov, V. Stankovic, and L. Strigini. Software diversity as a measure for reducing development risk. In *Dependable Computing Conference (EDCC), 2014 Tenth European*, pages 106–117, May 2014.
- [PYY14] S. Peng, S. Yu, and A. Yang. Smartphone malware and its propagation modeling: A survey. *IEEE Communications Surveys Tutorials*, 16(2):925–941, Second 2014.
- [RL07] J. Raber and E. Laspe. Deobfuscator: An automated approach to the identification and removal of code obfuscation. In *14th Working Conference on Reverse Engineering, 2007. WCRE 2007.*, pages 275 –276, oct. 2007.
- [RM13] K. A. Roundy and B. P. Miller. Binary-code obfuscations in prevalent packer tools. *ACM Computing Surveys*, 46(1):4:1–4:32, July 2013.
- [SB] W. Stallings and L. Brown. *Computer Security: Principle and Practice*. Prentice Hall, 2 edition.
- [SDAL01] B Schwarz, S Debray, G Andrews, and M Legendre. Plto: A link-time optimizer for the Intel IA-32 architecture. In *Proc. of Workshop on Binary Translation*, 2001.
- [SE10] L. Shan and S. Emmanuel. Mobile Agent Protection with Self-Modifying Code . *Journal of Signal Processing Systems*, 65:105–116, 2010.
- [SJ10] H. M. Sihvonen and M. Jaandntti. Improving release and patch management processes: An empirical case study on process challenges. In *2010 Fifth International Conference on Software Engineering Advances (ICSEA)*, pages 232–237, 2010.
- [SKF08] M. Shafiq, S. Khayam, and M. Farooq. Embedded Malware Detection using Markov N-grams. *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 88–107, 2008.
- [SNM03a] M. Sosonkin, G. Naumovich, and N. Memon. Obfuscation of design intent in object oriented applications. In *Proc. of Digital Rights Management*, 2003.

REFERENCES

- [SNM03b] M. Sosonkin, G. Naumovich, and N. Memon. Obfuscation of Design Intent in Object-Oriented Applications . In *Proc. of Digital Rights Management*, pages 142–153, 2003.
- [SS94] R. Sandhu and P. Samarat. Access Control: Principles and Practice. In *IEEE Communications Magazine*, 1994.
- [SSM05] Udupa S.K., Debray S.K., and Madou M. Deobfuscation: reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering*, page 10 pp., nov. 2005.
- [Sta07] K. B. Stan. Ode or code. programmers be used. *ACM Queue*, 2007.
- [Stu] BSA Global Software Piracy Study. The 2010 annual study on software piracy. <http://portal.bsa.org/globalpiracy2010/>.
- [SWL07] S. Stolfo, K. Wang, and W. J. Li. Towards Stealthy Malware Detection. *Malware Detection*, pages 231–249, 2007.
- [THW09] Hsin-Yi Tsai, Yu-Lun Huang, and D. Wagner. A graph approach to quantitative analysis of control-flow obfuscating transformations. *Information Forensics and Security, IEEE Transactions on*, 4(2):257–267, june 2009.
- [TK08] S. Theodoridis and K. Koutroumbas. *Pattern Recognition*, volume 11. Academic Press, 2008.
- [TYM04] Willard Thompson, Alec Yasinsac, and Todd Mcdonald. Semantic Encryption Transformation Scheme. In *Proc. of International Workshop on Security in Parallel and Distributed Systems*, 2004.
- [vO03] P.C. van Oorschot. Revisiting Software Protection. pages 1–13, 2003.
- [WCW⁺13] Wang, Chao, Wu, Zhizhong, Wang, Aili, Li, Xi, Yang Feng, Zhou, and Xuehai. Smartmal: A service-oriented behavioral malware detection framework for smartphones. In *High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 329–336, Nov 2013.

REFERENCES

- [WDHK01] C. Wang, J. Davidson, J. Hill, and J. Knight. Protection of Software-based Survivability Mechanisms . In *Proc. of Dependable Systems and Networks*, pages 193–202, 2001.
- [WGXW10] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang. Mimimorphism: A new approach to binary code obfuscation. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS '10*, pages 536–546, New York, NY, USA, 2010. ACM.
- [WS04] K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proc. of the 7th International Symposium on (RAID)*, pages 201–222, 2004.
- [YSY10] C. Yung, J. J. Syu, and S.Y. Yang. A graph-based algorithm of mostly incremental garbage collection for active object systems. In *Computer Symposium (ICS), 2010 International*, pages 988 –996, dec. 2010.
- [ZH10] A. Zhiyuan and L. Haiyan. Realization of buffer overflow. In *Information Technology and Applications (IFITA), 2010 International Forum on*, volume 1, pages 347 –349, july 2010.