View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by ethesis@nitr

Code Obfuscation
using
Code Splitting with Self-modifying Code

Shakya Sundar Das

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India
May 2014

Code Obfuscation
using
Code Splitting with Self-modifying Code

Thesis submitted in partial fulfillment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

(Specialization: Software Engineering)

by

Shakya Sundar Das

(Roll No.- 212CS3370)

under the supervision of

Prof. S. K. Jena

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India

May 2014

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India.

Certificate

This is to certify that the work in the thesis entitled ***Code Obfuscation using Code Splitting with Self-modifying Code***, by ***Shakya Sundar Das (212cs3370)***, is a record of an original research work carried out by him under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology with the specialization of Software Engineering in the department of Computer Science and Engineering, National Institute of Technology Rourkela. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Place: NIT Rourkela
Date: May 30, 2014

(Prof. Sanjay Kumar Jena)
Professor, CSE Department
NIT Rourkela, Odisha

Author's Declaration

I hereby declare that all work contained in this report is my own work unless otherwise acknowledged. Also, all of my work has not been submitted for any academic degree. All sources of quoted information has been acknowledged by means of appropriate reference.

Place: NIT Rourkela
Date: May 30, 2014

(Shakya Sundar Das)
M.Tech, 212cs3370, CSE Department
NIT Rourkela, Odisha

Acknowledgment

Writing of thesis is a journey through gravel road, but you can make it with the help of the people and resources you get in touch through out your journey. I am thankful to all of them for their hands to frame this thesis.

First of all, I would like to express my unfeigned thanks to **Prof. S. K. Jena** for his observations and guidance during my thesis work constantly encouraged me to build the direction to the research and to move forward with in depth analysis. Being a source of knowledge, his words make my road become straight and easier to go. I am also grateful to **all the professors** of our department for their advice and support.

I would also like to special thank to **Asish Dalai** Sir, PhD Scholar at NIT Rourkela, and my maternal uncle **Atanu Das**, PSA at NIC-Kolkata, for their help and support to clear my hurdles and understanding. And also thankful to **Vivek Balachandran** of NTU Singapore, the author of the paper "Potent and Stealthy Control Flow Obfuscation by Stack Based Self-Modifying Code", for his help to clear my doubts on mails.

I would like to thank all my friends and lab-mates for their cooperation and pepping up, which can never be penned with words.

One of the most important acknowledgment is for the academic resources that I have got from NIT Rourkela from the administrative and technical staff members who have been kind enough to advise and help in their respective roles.

Despite being mentioned after everyone else, **I would like to lovingly dedicate this thesis to my Mom & Dad** for their love, patience, understanding and to being my source of inspiration.

Shakya Sundar Das

Abstract

Code Obfuscation is a protection technique that transforms the software into a semantically equivalent one which is strenuous to reverse engineer. As a part of software protection and security, code obfuscation got commercial interest from both vendors' side to keep their proprietary as secret and customers' side to have a trusted software that don't leek or destroy their personal information. Today most of the software distributions contain complete source code in the form of machine code, which are easy to decompile and increase the risk of malicious reverse engineering.

The basic idea of the obfuscating technique that has been described in this research work is to hide the proprietary code section through preventive design obfuscation and insertion of self-modifying code at binary level. In this proposed technique the combination, while complementing each other, provides protection against all kind of reverse engineering.

Keywords: Software Protection, Reverse Engineering, Vendor, Decompile, Control obfuscation, Code Splitting, Self-modifying code.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reverse engineering attacks

Protection techniques for software intellectual properties

Objective

Thesis Organization

Chapter 1

Introduction

From the development of operating system to till now, software industry creates its own space in research, science and business. Softwares become the most desirable part of the digital world. Not only computers almost all the electronics devices have softwares embedded within them or installed on them. To hold this digital market software vendors started to put their proprietary code or algorithms as a secret part of the softwares and distribute in the form of executables without the source code. Alongside this evolution of the softwares make them complex to other people to understand the executables. Even it is difficult for the developer to maintain, to update and to patch new add-ons on client and customer side. This creates the need of software analyzing tools like disassembler, de-compilation [1] tools, reverse engineering [2] tools and many others. This analyzing tools again create a new threat for software industries - the stealing of the intellectual properties like code sections or algorithms.

Though these analyzing tools are the most essential part of software development life cycle phases [3], specially for testing, maintenance and up-gradations, these tools are also getting used for reverse engineering aiming at malicious intention of steeling or exploring the intellectual properties or vulnerabilities. These kind of software analyzing tools are largely available [4–7] on websites with documentation.

1.1 Reverse engineering attacks

The Reverse engineering [8] is the process of analyzing a target system to uniquely identify the system's modules and relationships between them and create a representations of the target system in another form or at a higher level of abstraction. In software world the reverse engineering method has been widely used in code reuse, software testing [9], to identify vulnerabilities [10], to analyze malicious codes [11] and protocols [12]. But the hackers and crackers are using this reverse engineering for their wicked purposes like code theft, code tampering, crack and piracy as shown in Figure 1.1, which is a open challenge to software proprietary informations protection. These kind of illegal activities cause the loss of money, business, reputation and the trust factor of the vendors.

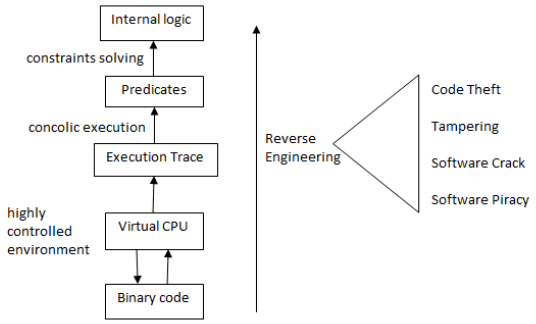

Figure 1.1: Attacks by Reverse Engineering

In the battle of software market every organizations have their proprietary codes or algorithms built into the executables which are sold to the customer. Which will be executed on untrusted computing environment where attacker have complete access to the software and hardware. Therefore, in such kind of environment software protection [13] is a definite need but harder to handle. Several cases of software law suits are filed, involving intellectual property theft using reverse engineering, like Atari Games Corp. v. Nintendo of America Inc. [14] in 1992, Sony Computer Entertainment Inc. v. Connectix Corp. [15] in 2000, in 2002 the case of Blizzard Entertainment on bnetd, which is claimed as a software package that was developed by reverse engineer the Blizzard Entertainment's Battle.net,a online multiplayer gaming service, poses almost equal workings [16].

1.2 Protection techniques for software intellectual properties

From these it is prominent that the protection of intellectual properties is a critical issue for the vendors while capturing customers with new technologies, with new software is the aim of the every software vendors, but to protect their new ideas they need copyright or patent. Even after if the ideas is copied somewhere else then again there will be court cases. It will be then investigated by technical experts. In order to perform such investigation several software tools, mathematical models and theoretical frameworks like SMAT [17] [18], MOSS [19] [18], AFC(Abstraction-Filtering-Comparison) [20] [18] and others are followed. In such theoretical framework (eg. AFC) the expertise first have to create an abstraction [18] depends upon purpose of the code, program structure and architecture, modules, source code & object code, then depending on these abstraction the expertise again have to compare [18] data structure, algorithms, system calls, formating, macros, bugs, execution paths, error, language to create a proper report. But still the evidence collected by the technical experts is open to legal challenges irrespective of the outcome which may produce delay in the process of litigation. All of these are time consuming, we have to also take into the monetary factors and above all the perfection of the thorough, authentic and convincing report of the technical expert in interest of proper justice. Both patenting and filing case on copy is not possible for small or medium size organizations or software vendors for all of their intellectual properties. This introduce the other technique of software protection like code obfuscation [21], software water marking [22], Tamper Proofing [22], white box cryptography [23], software fingerprinting [22], software diversification [24].

Some of the above named techniques are mainly to avoid reverse engineering by making its execution difficult or force it to crash on analyze & tampering and some of them creates special copyright key depending on execution path or provide security keys to identify theft.

Software watermarking [22] involves embedding a unique identifier or signature

within a piece of software. Watermarking does not prevent theft but instead discourages the stealing of codes by providing a way to uniquely identify the origin of the stolen software.

Watermark embedding : Program x Signature x Key → Watermarked program

Watermark extraction : Watermarked program x Key → Signature

Tamper proofing [22] are used to stop the execution of our software if it has been altered by adding tamper-proofing code to the software that will detect any change in code and cause the execution to a dead state or fail state.

Software Diversification [24] is a technique that generates different, but semantically equivalent, assembly instruction from a code sequence.

Software fingerprinting [22] is more or less same as above where the software vendors add a unique customer identification number for each distributed copy to track copyright violation.

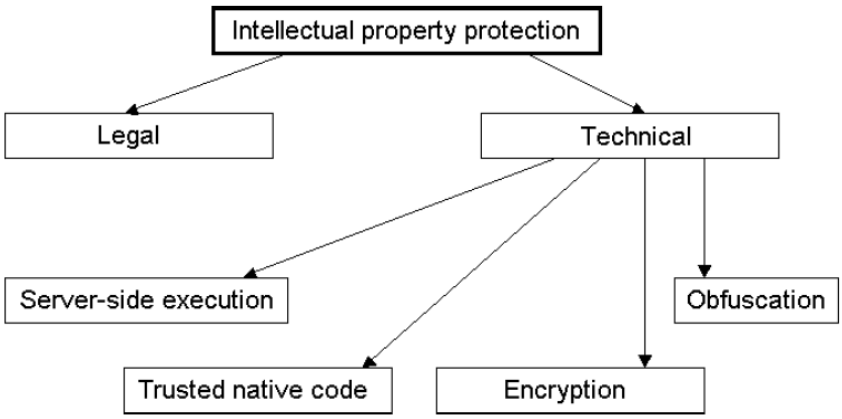

Figure 1.2: Protection of Intellectual Properties

All these methods are to protect the software piracy after reverse engineering. None of these methods can stop reverse engineering. Our research topic code obfuscation [22] is the only technique that make the target software difficult for reverse engineering and reduce human understandability of the code. As the executable is open to all we can not stop any body from using de-compiling tools or de-assembler for reverse engineering. But through code obfuscation [21] we are trying to make it difficult for the tools or humans to understand and to generate higher level abstraction of the target software without hampering the output the

result, for which the software is developed.

Other possible way is server side execution [25], trusted native code(eg. code authentication) [25], legal step on stealing.

1.3 Objective

The objective of this research work is to develop a technique that will prevent the attackers from dynamic reverse engineering [26] forces them to static reverse engineering [26] which they can't execute to trace its execution path and to get the actual high level code.

1.4 Thesis Organization

The rest of the thesis is organized as follows:

The basic informations to understand code obfuscation is given in **Chapter-2**, as Theoretical Background. It contain the definition and classification of code obfuscation and reverse engineering in software domain, the assumptions and the environments those are assumed for code obfuscation and the evaluation process of the obfuscation technique.

Chapter-3 will provide a overview of the related work and research done till now on code obfuscation. Which will lighten you up with a little deep knowledge about code obfuscation and help you to understand the needs, the drawbacks and various way to protect a software by code obfuscation.

Chapter-4 will take you through our proposed technique of code obfuscation with some basic theoretical knowledge about the technology used and the simulation and results of our proposed work.

In **Chapter-5** the overall work with strength and drawbacks is expressed in the section Conclusion and the possible future work is described in the Future Scope section.

Chapter 2

Theoretical Background of Code Obfuscation

Chapter 2

Theoretical Background of Code Obfuscation

2.1 Introduction

The main target of code obfuscation is to protect the sensitive information of the software from getting disclosed to the outer world. As the softwares are distributed in executable form in today's world, to get the sensitive informations or information about proprietary or intellectual properties from the executable reverse engineering is the only technique available in digital market. And code obfuscation is also the only technique that can prevent reverse engineering to some extent to analyze the target software.

2.2 Threat Model

After developing the software it is assumed that the executable will run on an untrusted host machine where the attacker have the complete access over the host machine, like the attacker have administrative access to the operating system, can add or remove hardware parts from the host machine, have access to the executable code of the software and to all kind of reverse engineering tool (e.g. disassembler, debugger) to analyze the code. After analyzing they can reuse any module for their program or can extract any proprietary algorithm or data structure and reveal it publicly for the loss of the vendor or insert extra code to get customers' information by violating the trust factor between vendor and customer.

2.2.1 Software Distribution Model

Here software distribution model means the various categories of user, who can use the software. A simple software distribution model consist of four different participants as follows:

A **Software Vendor**

They develop the softwares with new ideas and technologies to capture the market and want to maximize their profits by selling their software products, in present and in future.

B **Legitimate User**

These people are ready to pay to use the software which is not pirated and not malicious for their work. These people are conscious about their private information from getting damaged or disclosed. They need a trust factor from the products they are using.

C **Illegitimate User**

These people have no technical knowledge but want to use the software with the privileges of a legitimate user without proper compensation. They don't need the trust factor.

D **Pirates or Attacker**

These kinds of people with technical knowledge want to break all the security measure taken to protect a software code and use that code for their own software or to make pirated copy of that software for the illegitimate or legitimate user with minimizing the risk of being caught.

2.2.2 Attacks against Software Intellectual Properties

In the paper "Watermarking, tamper-proofing, and obfuscation-tools for software protection" [22], the authors defines various attacks against software intellectual properties using reverse engineering as defined bellow.

- **Software Piracy Attack**

  Attacker reverse engineers an application, which he has legally purchased from Vendor, and make pirated or cracked copy removing the vendor's signature and sells them to unsuspecting customer in low price. Which will create loss for the vendor from both profit from selling and the trust factor with the customers.

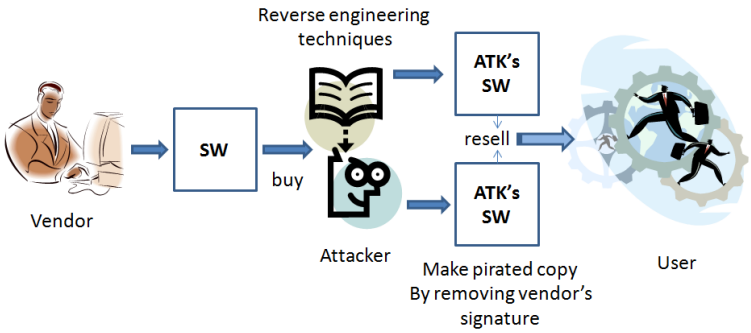

Figure 2.1: Software Piracy Attack

- **Module Reuse Attack**

  Attacker reverse engineers an application, which he has legally purchased from Vendor, in order to reuse one of the modules in his/her own program. As described in Figure-2.2, the attacker, after reverse engineer, reuse the code of Comp2 within his own software without developing by his own. This will create problems for the vendor in business to capture the market.

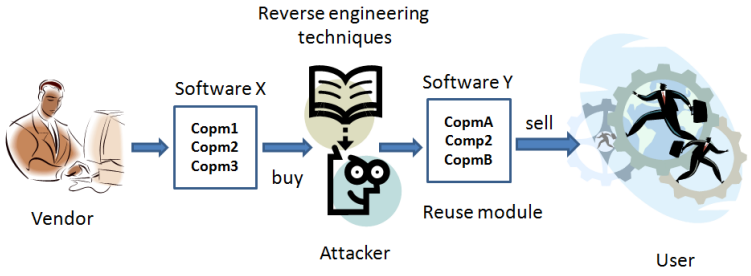

Figure 2.2: Module Reuse Attack

- **Code Alteration Attack**

  Attacker reverse engineers an application, after legally purchased from Vendor, makes some changes to the code for own profit irrespective the loss of

vendors or customers and resell the products to the market. As described in Figure-2.3, the attacker change the code as whenever a user execute the play() methods of the software as well as the vendor the attacker is also getting paid by the customer.

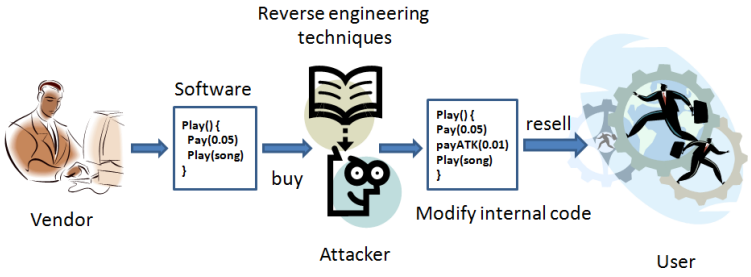

Figure 2.3: Code Alteration Attack

2.2.3 Reverse Engineering

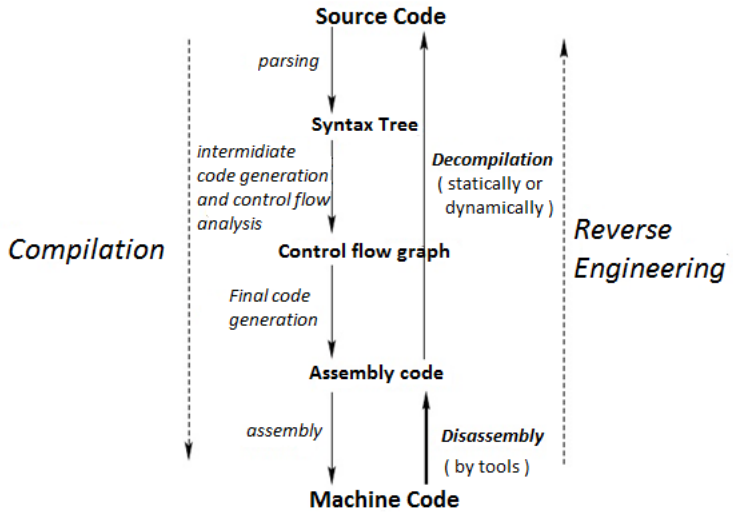

Figure 2.4: Process of Reverse Engineering

As described previously the Reverse engineering [8] is the process of analyzing a target system to uniquely identify the system's modules and relationships between them and create a representations of the target system in another form or at a higher level of abstraction. In the world of software reverse engineering allows attackers to understand the internal behavior of the executables and extract proprietary algorithms or code sections from it. In Figure-2.4 the reverse engineering

process is described.

The analysis of reverse engineering can be classified [26] as static reverse engineering and dynamic reverse engineering.

- **Static reverse engineering** is a technique where the structure of software executable is analyzed without actually executing it.

  Here a attacker can use a disassembler to translate the executable machine code into a visible, understandable assembly language code including all the instructions those affects the control flow. From this by manually checking each instruction attacker can generate the control flow graph from which the high level abstraction of the software/code-section can be reconstructed without executing it. It is a hard and tedious job but a possible way and need expertise of this domain. Static reverse engineering can be carried out by two ways. One is *Linear sweep* where attacker is just following each instruction as encountered one after another but this is very tedious and error prone technique. Other is *Recursive Traversal* where the attacker start with the program entry point and stops at every control flow instruction and determines possible predecessor and successor and then again continues on every possible paths. But determining the possible successor of every control instruction without executing is a very tough and error prone.

- **Dynamic reverse engineering** is a technique where the attacker execute the software executable within a debugger [27] to inspect its internal structures as well as the various execution paths.

  For encrypted code it is too hard to identify the key and tracking the code's transformation by static analysis. Then with the help of debuggers [27] [6] it is possible to execute the code step by step to identify the key and to track the code transformation.

2.3 Code Obfuscation

Code obfuscation technique is to obscure the control, data, layout, design of the software original implementation and give a semantically same but new implementation, as described in Figure-2.5, of the target software that make reverse engineering much harder.

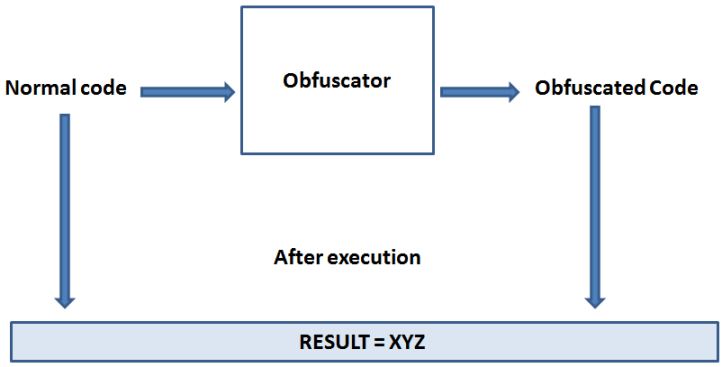

Figure 2.5: Code Obfuscation

2.3.1 Definition of Code Obfuscation

There is no common formal definition for code obfuscation. It is a transformation method to convert one program into another, which prohibits the same characteristics of the old program. It can also be treated as an encrypted code that is decrypted prior to execute at runtime. Obfuscated code can be an executables that contain encrypted sections, and a simple code section to decrypt the encrypted code section.

According to the authors of the paper "A taxonomy of obfuscating transformations" [21], the definition of code obfuscation is as follows

Definition : Let T(P) be a transformed program of program P. Then T is the Obfuscating Technique if T(P) poses the same observable characteristics as P and T(P) must follows the following conditions:

- If program P does not terminate or has an erroneous termination, then T(P) may or may not terminate.

- Else as P terminates successfully, T(P) must terminate with the same outcome as P.

According to the authors of the paper "A security architecture for survivability mechanisms" [28], if T is obfuscating technique that transform the program P into the obfuscated binary B, then the reverse transformation from B to P will take much greater effort and time(almost impossible),as T is a one way translator.

2.3.2 Classification of Code Obfuscation

Obfuscation comes in four flavors [21] based on obfuscation target - Layout obfuscation, Data obfuscation, Control obfuscation and Preventive transformation.

1 **Layout Obfuscation :** It refers to obscuring of the software layout by deleting comments for instance, changing format of the source code, variables renaming, and the removal debugging information through obscuring the lexical structure of the program.

2 **Data Obfuscation :** This prevents the extraction of information from data. Data obfuscation techniques are array splitting, variable splitting, changing the scope and lifetime of data etc.

3 **Control Obfuscation :** This refer to the obscuring of the control flow of the program. This kind of obfuscation technique mainly of dynamic obfuscation type based on self modifying code.

4 **Preventive Transformation :** Depending on debuggers' or disassemblers' weaknesses, modify the program such that code itself will force the debugger or disassembler to fail.

But this classification does not include all types of obfuscation techniques. Another possible classification is **Design Obfuscation** [29] which deals with obscuring the design related informations of the software. Like merging and splitting of code sections or classes, type hiding, will help in obscuring the design intend of the programs.

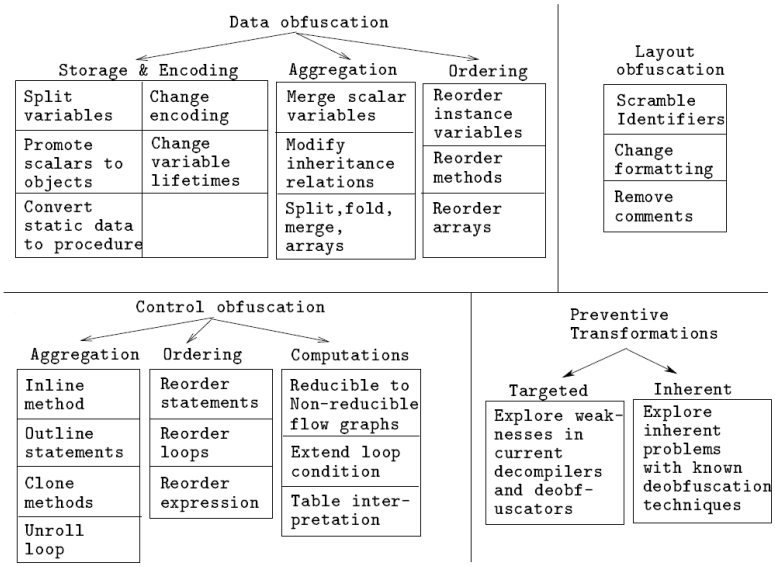

Figure 2.6: Classification of obfuscating transformation

2.3.3 Evaluation of Obfuscation Technique

There are two ways to check the quality [25] of a obfuscation technique as software engineering not only includes the technological and computational measures, it also include the human factor for every development. The two quality evaluation methods are defined bellow.

Analytical Method

Analytical method checks the quality of the obfuscating technique T() depending upon the parameters of both original/source program P and the obfuscated program T(P). According to authors of the paper "A taxonomy of obfuscating transformation" [21], they are evaluating the quality depending upon three parameters - potency, resilience and cost.

- **Potency :** It can be described as - how much obscurity T() adds to P. Let Pot(P) is the potency measurement of P and Pot(T(P)) is the potency measurement of T(P) then

$$TransformationPotency, TPot = Pot(T(P))/Pot(P) - 1 \qquad (2.1)$$

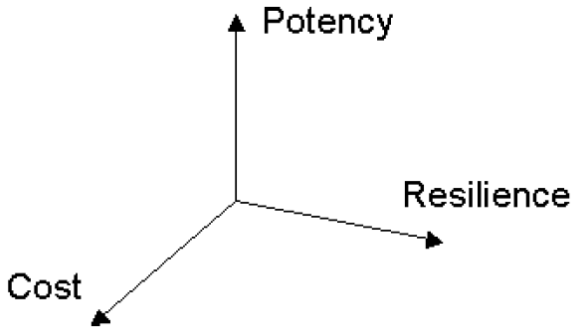

Figure 2.7: Metrics for quality measurement of the obfuscation technique

Potency for both P and T(P) can be measured by various software complexity metrics defined in Table-2.1.

Table 2.1: Software complexity metrics to measure potency

Metric Name	Description	Citation
Program Length	Pot(P) increase with the number of operators and operands in P	Halstead [30]
Cyclomatic Complexity	Pot(P) increase with the cyclomatic complexity of P	McCabe [31]
Nested Complexity	Pot(P) increase with the nesting levels of the conditionals in P	Harision [32]

- **Cost :** It is measure by how much computational overhead T() adds to T(P). It is the execution time penalty and space penalty that the obfuscation technique incurs on T(P).

  If executing T(P) requires exponentially more resources than P then

  $$Transformation Cost, TCost = Dear \tag{2.2}$$

  If executing T(P) requires $O(n^p)$, p>1, more resources than P then

  $$Transformation Cost, TCost = Costly \tag{2.3}$$

  If executing T(P) requires $O(n)$, more resources than P then

  $$Transformation Cost, TCost = Cheap \tag{2.4}$$

If executing T(P) requires $O(1)$, more resources than P then

$$TransformationCost, TCost = Free \tag{2.5}$$

- **Resilience :** It is measured by how difficult is T(P) to break for a deobfuscator means how well a T() holds up under attack from a automatic deobfuscator. Resilience can be measured by summing the total of programmer's effort and deobfuscator's effort [21].

  Programmer Effort (PEff) - The amount of time require by the programmer to build the automatic deobfuscator to regenerate P from T(P).

  Deobfuscator Effort (DeoEff) - The amount of execution time and space required for the automated deobfuscator to deobfuscate the transformed program.

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

$$TransformationResilience, TRes = OneWay \tag{2.6}$$

  Otherwise

$$TransformationResilience, TRes = Res(PEff + DeoEff) \tag{2.7}$$

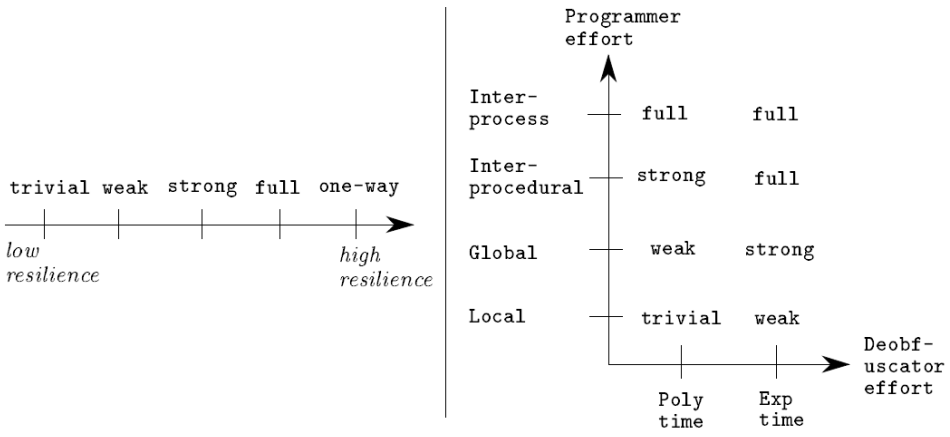

Figure 2.8: Measure of resilience

Empirical Method

The main target of code obfuscation is to protect the proprietary code sections or algorithms from unauthorized analysis and in reverse engineering the last step of analysis is totally depends on human effort [25] which can not be measured by any metrics. For this we need to perform empirical research on a group of people like programmers, hackers or crackers, students.

2.4 Summary

Throughout this chapter, what ever is discussed is just to create a basic understanding for this research work. For more details you can go through the papers and websites refereed throughout the paper.

For some more interesting informations you can have a look to the website of Prof. Christian Collberg [21] [22] [13] of University of Arizona, the website of "The International Obfuscated C Code Contest", the website of University of Florida on "Obfuscated C Code", the website of Princeton University on "Obfuscated code" and so many hacking websites those are available on Internet. These are some unnamed references of this paper.

Chapter 3

Literature Review

Review of related work

Motivation

Chapter 3

Literature Review

There are number of techniques and their implementation have been evolved by many researchers since past decade based on protection against static and dynamic reverse engineering. Each approach has different techniques and targets depends on static or dynamic reverse engineering like control flow flattering [28], obfuscation using signals [33] [34], dynamic code mutation [35] [36], binary level obfuscation [29], protective transformation [37] [38] and others.

3.1 Review of related work

Protection of software based survivability mechanisms - 2001

In this paper [28] authors used control flow obfuscation by the use of control flow flattering technique to confuse the disassembler about the execution sequence of the program. Here the researchers first divided their program into basic blocks depending upon the high level control structure. Then this control structure is replaced with "if-then-goto" statements. After this the construction is changed in such a way that the target address of goto will be determined dynamically after the execution of each block and will be stored in switch variable and a switch statement, depending on the value of the switch variable, determines which block to be executed next. In simple words each basic blocks will have the same predecessor and successor block, where basic block will calculate the switch variable,

then control will go to the successor block which will give the control back to the predecessor block and then depending on the switch variable predecessor block will determine which basic block will be executed next.

Though this is very good dynamic control flow obfuscation but with the increase of program and input size the requirements for memory and executing time will increase and above all modern debuggers can provide a rough diagram of the control flow by executing each instruction at assembly level one at time.

Software protection through dynamic code mutation - 2006

The researchers of this paper [36] implement a dynamic code obfuscation technique that will remove some set of code which will be restored at run time. To implement this idea they are using three extra code module - stub, edit script and edit engine. First thing they are doing is the identification of basic blocks, then they are removing a set of code from a basic block and put the restoring informations in a edit script. Afterwards they include a stub, which will have the address of the corresponding edit script, at the beginning of that block and desperately put some confusing erroneous code on place of removed set of code. At the time of execution stub will be executed first and transfer the control to edit engine with the address of corresponding edit script. Then according to edit script the edit engine will restore the original set of code at position of the erroneous set of code. This method is implemented in two ways by the researchers of this paper. One is *One-Pass Mutation* where each functions or basic blocks will have their own edit script. Other one is *Cluster-Based Mutation* where a group of similar functions will have a single edit script.

The major disadvantage of this technique is the stub section is always be in highlight, that will draw attention of the attacker. Other disadvantage is after restoring, the original code is fully exposed to the debugger or attacker.

Binary obfuscation using signals - 2007

Here [33] the researchers also give a new technique of control flow obfuscation by hiding the control flow information of a program using signal, which are used carry information between operating system and information. This research work is based on the replacement of every control instruction at binary level (eg. JMP, RET, CALL) with trap signals like SIGILL for illegal instruction, SIGSEGV for segmentation violation and SIGFPE for floating point exception. It first identify the the control instruction, then divides the code-before and code-after segment of the control instruction. After this the control instruction is replaced with a trap instruction and some bogus code is inserted between the trap instruction and the code-after segment. Then the user defined signal handlers are installed within the program with a special table that will contains the actual instruction for corresponding generated signals. At runtime when the trap signal will executed the control will go to the operating system's corresponding signal handlers, then the control will be transfered to user-defined signal handler for the corresponding signal. Then the user-define signal handler will execute the corresponding code and then transfer the control to the code-after segment.

The one disadvantage of this technique is the control instruction is available within the user-defined signal handler. If the attacker can identify the signal handler, he can identify the control instruction by analyzing the signal handler.

Mimimorphism: a new approach to binary code obfuscation - 2010

In this paper [39] the authors give a totally different kind of obfuscation technique based on mimic function that has three phases - a digesting phase for Huffman tree building, an encoding phase that use Huffman decoding technique and a decoding phase that use Huffman encoding technique. Here the mimimorphism technique use mimic function of higher order which differ in digesting phase from regular mimic function by building a collection of Huffman trees for better mimicry

and a mimimorphic engine,that include all the three phases, is added to the obfuscated program to restore the original code at run time. Here, in Digesting phase, from the executable with help of an assembler for each assembly instruction with all the parameters and the frequency of occurrence those parameters are stored and all the instruction is also get stored with a unique id and with the frequency of their occurrence, after this a Huffman tree for each instruction is created depending on their parameters frequency. At encoding phase this technique use the Huffman decoding operation based on the Huffman trees generated earlier in digesting phase and output a completely different assembly code that will convert into a binary with the help of an assembler. At execution time of the new binary code the mimimorphic engine apples its decoding function on the binary, that use the Huffman encoding operation, depending on the Huffman trees generated earlier to restore the original program for execution.

Here the binary code that will be distributed cant be reverse engineer statically but it includes the mimimorphic engine, the decoder with the Huffman trees with unobfuscated status. This may reveal the original code with dynamic analysis and also encoding and decoding the whole program is very time consuming when program size will increase.

Mobile agent protection with self-modifying code - 2011

This paper [35] introduces a light weight but self-modifying code based technique at binary level. The proposed obfuscation technique of the this paper camouflaged the control instructions with normal instructions or with other control instructions. This method defines each control instruction as a candidate block, the code section before the candidate block is named as preceding block and the code section afterwards is as succeeding block. At the time of obfuscation this technique replace the control instruction (for example JMP instruction) at the candidate block with normal instruction (for example MOV instruction) and add a modifying block to its preceding block and add a restoring block to its succeeding

block. The modifying block performs some AND-OR operations on the address of the candidate block to restore the original instruction at run time. After execution of the candidate block when control goes to the succeeding block, then restoring block again perform some AND-OR operations the address of the candidate block and restore the camouflaged instruction again in the candidate block at runtime.

The obfuscated code developed by this method will not be too much bigger than the original one, as no extra code section is add, instead 2-4 simple binary level code is added to the original binary one. This kind of obfuscation is very hard to be found by static reverse engineering and make the analysis error prone. But the original is exposed temporarily at the time of execution which can be detected by dynamic reverse engineering with the help of any debugger [27] [6] and also the modifying and restoring block can be identified by step-in execution(execute one instruction at a time) within the debugger.

Branch obfuscation using code mobility and signal - 2012

This research work [34] provide a obfuscation technique where resilience [21] is one-way means the original program can not be reconstructed from the obfuscated one. On the basis of the paper "Binary obfuscation using signals" [33] the researchers of this paper build their work. They are also using the trap instruction in place of the control instruction, that they want to be obfuscated. In the same way of the base paper [33] they removed the control instruction and put a trap instruction with bogus codes afterwards. When the trap instruction will execute depending on the generated signal control will transfer to operating system, then to the corresponding installed user-define signal handler. Here the signal handler will communicate to a remote trusted server/machine by passing the value of the actual condition variable to know the next code section that will going to be executed next. On receiving the value of the condition variable the server generate the corresponding result and pass it to the signal handler, which will then pass the control to the next executing block depending on value of the result. Here they

are not providing the complete executable code to the customer. They are removing some information from the provided binary one and add server-side execution of the removed information, code obfuscation technique is only used to hide the actual control instruction form the attacker.

This a hybrid method of code obfuscation and server-side execution. As some code is removed from the provided binary, the original code can never be reconstructed from the binary with the help of any kind of reverse engineering. But the performance of this code totally depends upon the connectivity of between the two machine. If the network bandwidth is too low or there is no connectivity between the two machine, this implementation is totally worthless.

Potent and stealthy control flow obfuscation by stack based self-modifying code - 2013

Here [29] the researchers developed a stronger new obfuscation technique based on the paper "Mobile agent protection with self-modifying code" [35] described earlier. On the previous paper they are just trying to hide the control instruction but the address where the control will be transferred is still available after camouflaged. Here the researchers have shown a way to hide the address also as a local data to that function, which will be stored on the stack section of data area. In this research work the researchers take executable machine code and then generate its corresponding assembly code. Then they select the control instruction to be obfuscated. Lets take they are going to obfuscate a JMP instruction(an assembly instruction for unconditional jump with a address parameter). So to store the address in the stack they are just extending the size of stack that will always be allocated at the starting point of the function. After this before obfuscating the instruction they stored the jump address in the stack and then replaced the JMP instruction with a normal instruction and add an extra instruction in the modifying block after the de-obfuscation instructions to restore the address at run time and an extra instruction to restoring block to remove the address at run time,

before the re-obfuscation instructions.

This method provide a code obfuscation mechanism that is to hard to be analyzed by static reverse engineering as both address and the instruction is not visible until the function stores its stack onto the memory. This thing also make it hard for dynamic reverse engineering. But in modern debuggers [5] [4] [6] [27] if we execute the obfuscated binary with the step-in (execute one instruction at a time) execution it will shows all possible values of every registers and stack pointer, local and global variable values used at that moment.

An anti-reverse engineering guide - 2008 [ONLINE]

This is a article at "Code Project" website [37] [38] by Josh jackson. In this article he has provided various anti debugging technique based on facility we can get from the operating system and and the status of a program's environment variables. Though most of them are only for windows operating system, it is very helpful. Hare he has shown how to disable the interrupt signals to stop the step-in execution of the debuggers. Some technique have been shown to detect the presence of debugger by the code itself, depending on which code can crash itself or give erroneous result or correct result.

```
unsigned long NtGlobalFlags = 0;
__asm__ __volatile__(
        "movl %%fs:0x30, %%eax;"
        "movl 0x68(%%eax), %%eax;"
        :"=a" (NtGlobalFlags)     );
```

This is one of that code which can check the presence of debugger by checking program's environmental variable like

- FLG_HEAP_ENABLE_TAIL_CHECK,
- FLG_HEAP_ENABLE_FREE_CHECK,
- FLG_HEAP_VALIDATE_PARAMETERS.

If all of these are set means the program is executing within a debugger on windows platform. These are some tricky undocumented facility of windows operating systems are providing.

But the drawback is that when we execute on a cross platform environment(eg. executing a windows application on linux with the help of wine application) this kind of tricky code will not work, and debugger can easily debugged the target program.

3.2 Motivation

These are the some of the related those have been studied on code obfuscation. Till now the main trend of code obfuscation is to stop static reverse engineering. In the case of static reverse engineering, I found the binary level obfuscation is more tough to analyze, as assembly level code is hard to understand and also no compiler optimization will be there if some dead code or some extended code is there for obfuscation. But in modern debuggers, the step by step execution of code can reveal the obfuscated instruction if code is throughly analyzed with all registers and stack & heap values.

All of these methods either exposed to dynamic reverse engineering or static reverse engineering and some of them are exposed to cross platform debugging. In this research work, the motive is to implement a obfuscation technique that will extend the security measures to all the mentioned debugging techniques.

Chapter 4

Proposed Technique & Implementation

Introduction

Assumption

Proposed Technique

Implementation

Chapter 4

Proposed Technique & Implementation

4.1 Introduction

The proposed method of this research work motivated by the level of obfuscation against the dynamic reverse engineering. The idea of the proposed method developed here only to stop dynamic reverse engineering where to stop the static reverse engineer the method stated in the paper "Mobile agent protection with self-modifying code" [35] is implemented. The proposed method, depending on the general workings of debugger, puts some conditions which will prohibit the debugger [6] [27] from debugging dynamically and force him towards static reverse engineering. Where the method implemented in the referred paper [35] will make the static analyzing much tougher.

4.2 Assumptions

In the domain of software protection the assumption is simple and straight. Where attacker can have only the executable(in binary) of the target program but can have complete access to the host system's hardware and operating system and only using the reverse engineering technique with debuggers or disassembler. While the developers have complete access to the program's high level source code, to the machine level code and also have proper mappings of obfuscated code positions of the target program.

4.3 Proposed Technique

Code obfuscation using code splitting with self-modifying code :

To stop dynamic reverse engineering it is taking into account the working principle of debuggers [5] [6] [27]. Depending on this the proposed technique will split the original code in two main separate program which will communicate through shared memory and one more program for starting the client on server's call. Among this two main programs user can interact with only one program, the server program, where the proprietary code section will be in the other program, the client program. Our main technique prohibit the client program from getting executed under the debugger, so our binary level obfuscated proprietary code can not be debugged dynamically. The basic idea is: when a debugger try to execute a program it will start the target program as its child program. But before starting the target program it will start also many other program simultaneously to inspect the target program.

- So the debugger can't analyze every instruction on occurrence until the it starts the target program as its child or thread.
- And it is also possible to check the parent process id of any program from the program itself.

Depending on the design is done in such a way that the server program will create the client program as a zombie process (in linux environment whose parent process id is 1 means the INIT Process) to stop the execution of the client within the debugger.

4.3.1 Basics behind the proposed technique

This section will give you little bit of highlights on the working principle [40] of debuggers and shared memory for inter process communication [41] [42] and little bit of x86 architecture [43] [44] to understand the binary level obfuscation defined by the paper "Mobile agent protection with self-modifying code" [35].

Debugger's working principle

The target of a debugger is the complete analyze of the target program. For complete analyze, debugger have to check all the content machine level instruction, all variables and functions, all registers' and stack values [40] [7].

A **CPU view** consist of a disassembler pane for complete assembly level view of machine code(binary code), a register pane for register values, memory dump pane to display the content of any memory location, flag pane to display all the set flags.

B **Memory view** used for checking the variables, functions, types and their corresponding virtual address with the memory layout of the complete program, like different code section, data section, stack section of the program in execution.

C **Stack view** shows the calling and return trace of all the functions and all local variables accessed with the stack virtual address.

D **Break Point view** show the all breakpoints introduced by the developers or the attackers.

After setting up all these environment debugger, with the help of the operating system, either spawn the target program or debuggee with full control on the debuggee or attach itself with the debuggee for getting all the status information when ever the debuggee stops its execution depending on breakpoints or on error or on successful execution. On the first method the step by step execution of every instruction is possible but not possible with the second method.

x86 Architecture

x86 is a backward compatible instruction set architecture family based on Intel 8086 CISC (Complex Instruction Set Computing) processors with 8 general purpose registers and 6 segment registers of 16 bit or 32 bit or 64 bit.

x86 General Purpose Register (as 32 bit register symbol)

1 Accumulator(EAX), used for arithmetic operation.

2 Base register(EBX), used as a pointer to a data.

3 Counter register(ECX), used in shift/rotate instruction and loop statements.

4 Data register(EDX), used in arithmetic and I/O operation.

5 Stack Pointer(ESP), used as the TOP of stack memory.

6 Stack Base Pointer(EBP), used as a pointer to the base of stack memory.

7 Source Index Register(ESI), a pointer to a source in stream operation.

8 : Destination Index register(EDI), a pointer to a destination in stream operation.

x86 Segment Register

1 Stack Segment (SS) : used as a pointer to stack

2 Code Segment (CS) : used as a pointer to the start of text/code section

3 Data Segment (DS) : used as a pointer to data section in memory

4 Extra Segment (ES) : used as a pointer to extra data

5 F Segment (ES) : used as a pointer to more extra data

6 G Segment (GS) : used as a pointer to still more extra data

Shared Memory for Inter Process Communication

In shared memory inter process communication model where one process creates a common memory location, which other processes can access by attaching themselves with the shared location as shown in Figure-4.1. Here we use share memory as we divide one program in two co-operating process and once shared memory is created the communication between two process is fast even for huge size of data.

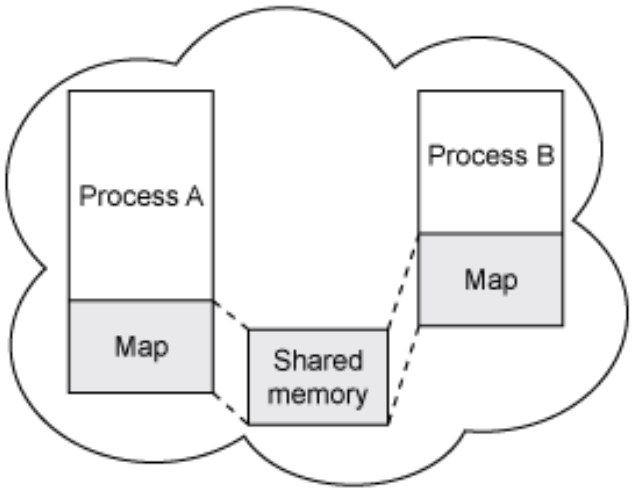

Figure 4.1: Shared Memory for Inter Process Communication

4.3.2 Design of the proposed technique

This proposed technique is based on design [29] and preventive [21] obfuscation techniques by dividing the target program in three part.

- **Server :** Holds the code for user interaction.
- **Client-Start-up :** Start the client on server call.
- **Client :** Holds the proprietary code section.

This division will be done manually, it depends upon the developer. After this division it will make the server to create a share memory that client can access. After the creation of share memory server will start a client-start-up program, which will terminate after starting the client. This technique make the client to execute as a zombie process(whose parent is the INIT process). Then the client will check whether its running with parent process id as 1 nor not. If it is a zombie process then only the client will attach itself to the share memory location, after successful attaching it will collect data from share memory, calculate result, write result back to share memory and will terminate successfully. After collecting the result from share memory, server will execute the rest code section. A generalized diagram is described the Figure-4.2 for the overall understanding of the design.

This obfuscation technique useful to restrict the debugger from dynamic analysis of the obfuscated executable.

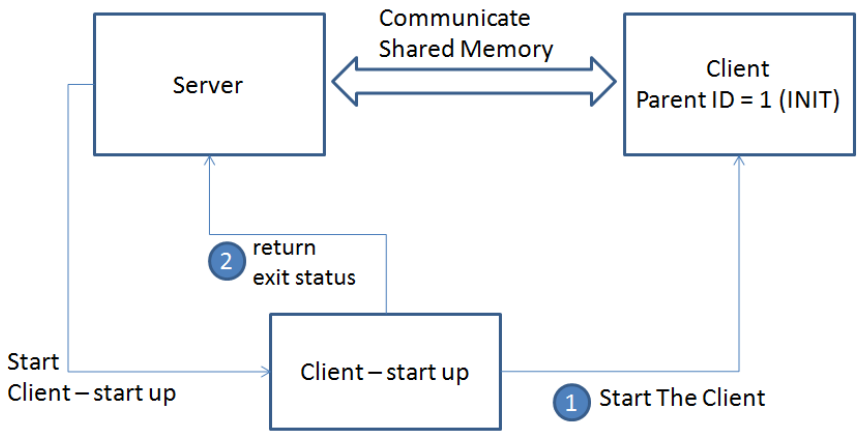

Figure 4.2: Design of proposed technique

4.4 Implementation

Implementation environment

In this research work the implementation of the proposed technique is based on Linux environment. The Linux environment is developed by setting up a virtual machine hosted by VMware Workstation on Windows platform.

- **Operating System:** Debian GNU / Linux Kali Linux 1.0
- **System Confuguration:**
  - Processor Model : Intel(R) Core(TM) i7 - 3632QM
  - CPU GHz : 2.20 GHz
  - Cache Size : 6144 KB
  - Core : 2 (on VMware)
  - Memory : 2 GB (on VMware)
  - HDD : 30 GB (on VMware)
- **Compiler:** gcc version 4.7.2

- **Debugger:** EDB Debugger(Evan's Debugger)
- **Programming Language:** ANSI C

A detailed design with a small example

For better understanding a very simple program is taken here, that will display the sum of first one million natural numbers (1 to 1000000) with help of a "sum˙million()" function which implements the logic using a simple for loop in C language and our target is to hide the implementation of "sum˙million()" function. To do so, first thing is the design level preventive obfuscation technique, that are proposed to stop dynamic reverse engineering and the next is machine level or binary level self modifying code [35] implementation to make it harder for static reverse engineering.

In design level obfuscation it will create total three different program - first one is the server program to interact with the user and to create the shared memory, second the middle element is the client-start-up program, which will be called by the server to start the client as zombie process (means the server will start the execution of this program and after starting the execution of the client it will terminate itself) and third one is the client program which will hold and execute the "sum˙million()" function. Figure-4.4 and Figure-4.3 describe the whole function visually with comparison between the obfuscated code normal code.

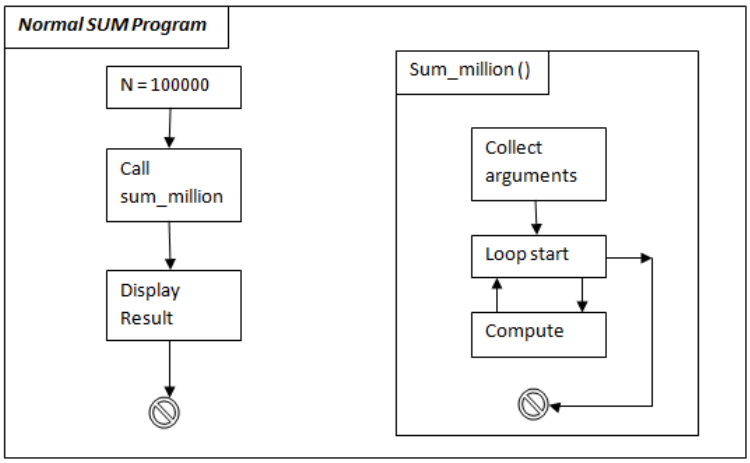

Figure 4.3: Design of Control flow between the Programs

According to McCabe's Cyclomatic Complexity formula [31], the complexity measure for this two program is being calculated . For the normal program the value is 4 where as for the obfuscated one the value is 7 implies that the obfuscated one is more tough to test, to analyze, to maintain and to understand.

So the measure of potency, according to Equation-2.1 defined in Chapter-2, for normal program is Pot(P)=3 and for obfuscated program is Pot(T(P))=7. So Transformation Potency(TPot) is 2.33 (greater than Zero) implies that the obfuscated one is harder to understand.

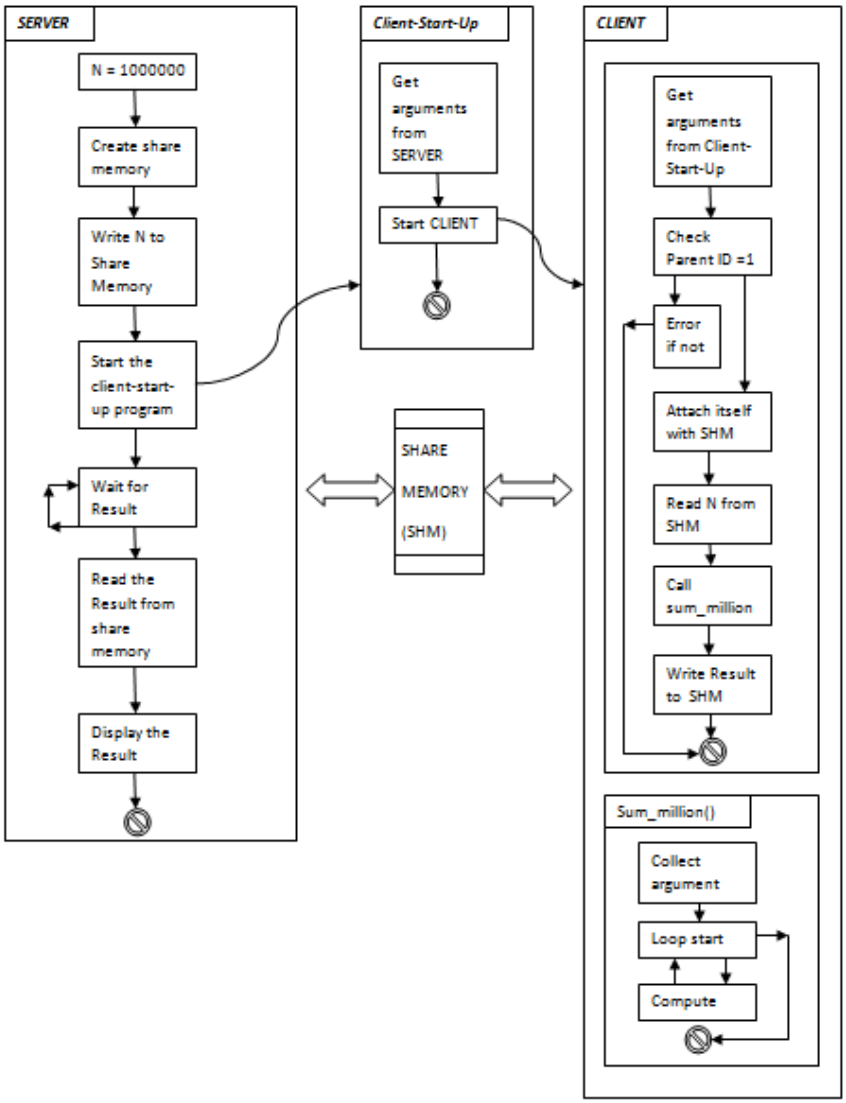

Figure 4.4: Design of Control flow between the Programs

And it also includes the inter process communication. This implies that the measure of resilience of the obfuscated program is "Full" according to the Figure-2.8 described in Chapter-2.

After the design level obfuscation the proposed technique will modify the binary code of the client program, in the way it is stated in the paper [35], to make it harder for static reverse engineering. Here in the implementation, only the preventive code section (checking of parent id) and the for loop to calculate the sum of the client program is obfuscated. On these sections only the jump instructions is modified into move instructions, which will be changed to the original jump code at runtime and after the execution of the jump instruction that will again changed to the move instruction on the go as described in the Figure-4.5 bellow.

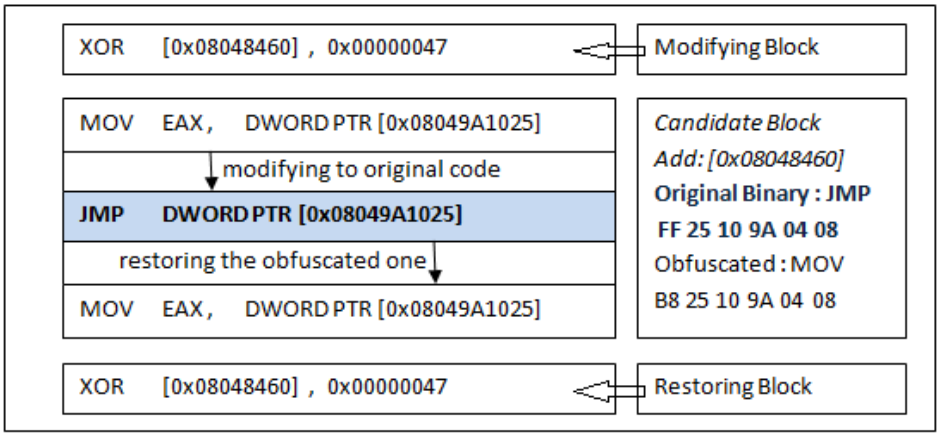

Figure 4.5: Design of binary level obfuscation

Table 4.1: Execution Time of the program : Sum of First One Million Natural Number

	Original Program	Obfuscated Program
real	0 m 0.011 s	0 m 0.021 s
user	0 m 0.008 s	0 m 0.012 s
sys	0 m 0.000 s	0 m 0.004 s

The result of Table-4.1 is obtained by running both the program with the "time" command in Linux. From this table we can calculate the time-cost factor

by considering only the "user" and "sys" time, as real time is depends on all other programs those are running at that time.

- **Original Program (P) :** user + sys = 0 m 0.008 s
- **Obfuscated Program (T(P)) :** user + sys = 0 m 0.016 s
- **Time-cost factor is { time(T(P))/time(P) } = 2**

From this factor it can be concluded that depending on the size of input this result is not so high. As both the program has same complexity level $O(n)$ for original program and $O(n+c)$ for the obfuscated one, which is equivalent to $O(n)$, concludes the Time-cost factor as "Cheap" according to the Equation-2.4 defined in Chapter-2 as stated in the paper [21].

The informations about memory requirements of the executables are also collected by running them with the "size" command in Linux, shown in Table-4.2 and Table-4.3. According to the Table-4.3 total space require for the obfuscated program

Table 4.2: Memory Requirements for the Original Program

	text	data	bss	total
size (in byte)	1281	292	04	1577

Table 4.3: Memory Requirements for the Obfuscated Program

	text	data	bss	total
size of Server (in byte)	1845	316	20	2181
size of Client-Start-Up (in byte)	1588	308	04	1900
size of Client (in byte)	1981	320	04	2305

is the summation of space requirement for the three programs which is equal to 6386 bytes. So the Space-cost factor { space(T(P))/space(P) } is equal to 4.049, which is much higher. According to space-cost our proposed method can be concluded as "Costly". But today's world space requirement will not create a big problem.

Testing of Preventive Transformation

Testing of the preventive transformation is done with the program "Sum of first One Million Natural Number" and EDB Debugger(Evan's Debugger) [27] on Linux platform. As client program can only run as zombie process, whose parent can be the INIT process only, so

- If debugger execute its as its child, client will crash.

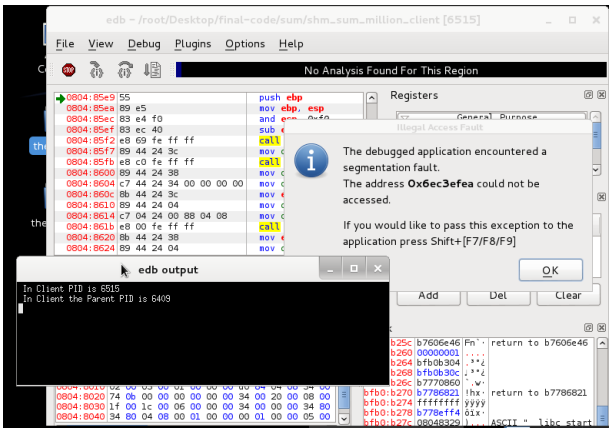

Figure 4.6: Screen shot : Crash testing

- Suppose the debugger attach itself with the client by calling the debugger loaded client version from the client-start-up program. But then after the

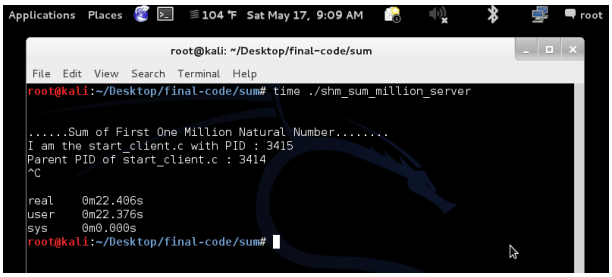

Figure 4.7: Screen shot : Failure testing

the termination of client-start-up, debugger will lost its control over the environment variables it needed from the operating system to start and to debug a program, as debugger itself is a zombie process now. Here the program is terminated forcefully which is cleared from time shown in Figure-4.7.

- If attacker change the client-start-up program such that it will not terminate after starting the client, then also client will crash as parent process is not the INIT process. As shown in the first Screen shot Figure-4.6.

- The debugger can not also debug the INIT process as debugger itself is the child of the INIT process.

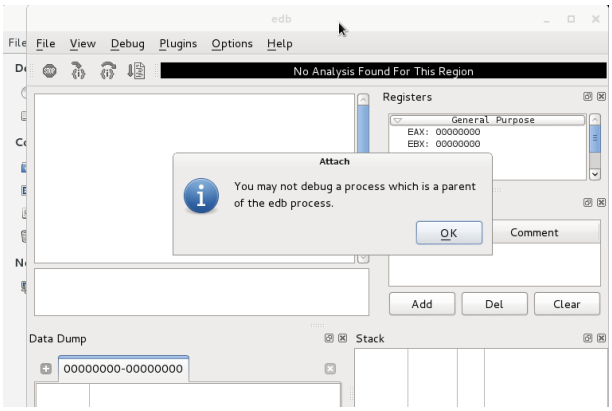

Figure 4.8: Screen shot : A try to debug INIT process

- The only way to execute the client is to remove the code for which the client get crashed if its parent is not the INIT process. For this debugger have statically analyze the code but the binary level obfuscation by the self-modifying code section make it much harder for the attacker.

Testing of efficiency of our technique on some sorting programs

Here some testings are also done with common sorting programs - Bubble sort, Insertion sort, Heap sort and Quick sort. All the programs have been tested with a array of 100000 unsorted integers generated with the “rand()” of C library, with the default seed value as ONE, for both the original and obfuscated program. Here the first thing for any program is to initialize the array with the “rand()”, then sorting of that array.

For obfuscation first implementation is the code splitting part for design level preventive obfuscation against dynamic analysis, then the modification of the binary code to make it harder for the static reverse engineering.

As this technique is based on inter process communication, according to the paper [21] the measure of resilience is "FULL". For detail description please go back to the Figure-2.8 described in Chapter-2. Potency of each program is measured with the McCabe's Cyclomatic Complexity, shown in Table-4.4.

Table 4.4: Measure of Potency

Program	Potency of Original Program, Pot(P)	Potency of Obfuscated Code, Pot(T(P))	Transformation Potency, TPot
Bubble sort	7	15	1.143
Insertion sort	7	15	1.143
Heap sort	13	22	0.692
Quick sort	9	16	0.778

The measure of Cost in terms of memory space requirements and execution time. Table-4.5 shows the space-cost factor and Table-4.6 shows the time-cost factor.

Table 4.5: Measure of Space-cost Factor (SIZE : byte)

Original	Size	Obfuscated	Size	Total Size	Space-Cost Factor
Bubble sort	1926	Bubble_Server	2594	6155	3.196
		Client-start-up	1844		
		Bubble_Client	2717		
Insertion sort	1896	Insertion_Server	2594	7137	3.764
		Client-start-up	1848		
		Insertion_Client	2695		
Heap sort	2356	Heap_Server	2858	7578	3.216
		Client-start-up	1844		
		Heap_Client	2876		
Quick sort	2224	Quick_Server	2594	7417	3.335
		Client-start-up	1844		
		Quick_Client	2979		

Table 4.6: Measure of Time-cost Factor (TIME : SECOND)

Program	Original program			Obfuscated Program			Time-cost factor
	real	user	sys	real	user	sys	
Bubble sort	105.894	101.240	0.032	113.533	104.008	0.056	1.028
Insertion sort	20.337	15.772	0.020	22.419	17.616	0.040	1.118
Heap sort	4.973	0.148	0.040	5.024	0.161	0.064	1.197
Quick sort	4.780	0.100	0.056	4.974	0.128	0.080	1.333

Here for Table-4.6, running each program for 8 times, with the Linux "time" command, the average time for each section has been taken excluding the highest value and the lowest value. For the measurement of time-cost factor, only "user"

and "sys" times are taken into account as "real" time is influenced with all other programs, those are running at that time. If Time(P) is the total of "user" and "sys" time of the original program and Time(T(P)) is the total of "user" and "sys" time of the obfuscated program, then Time-cost factor = {Time(T(P))/Time(P)}.

For checking the change of time with input data size, the size of the array has been varied from 10 to 100000 for the bubble sort program. As the sort time taken by bubble sort is much higher than all other program for both original code and obfuscated code. So the change in time will more prominent for bubble sort.

Table 4.7: Measure of Time-cost Factor for Bubble sort

Array size	Original (in Sec)			Obfuscated (in Sec)			Time cost Factor
	real	user	sys	real	user	sys	
10	0.003	0.000	0.000	0.008	0.000	0.004	-
100	0.003	0.000	0.000	0.008	0.000	0.004	-
1000	0.018	0.008	0.000	0.024	0.012	0.004	2
10000	1.475	1.048	0.000	1.492	1.060	0.012	1.023
100000	105.894	101.240	0.032	113.533	104.008	0.056	1.028

For this table also only only "user" and "sys" time are considered as real has influence of other running program in the system on that time.

Chapter 5

Conclusion

Achievements

Limitations & Future Scope

Chapter 5

Conclusion

5.1 Achievements

Above all descriptions, results make you feel that code obfuscation is reducing the performance of your program or software. Yes it is. But if you need protection and security against piracy, code theft and other attacks you have to introduce some security measures, you have to except the trade off between the performance and level of security. This research technique uses a combination of code splitting and binary level obfuscation. Where code splitting is threatened by static reverse engineering and binary level obfuscation is threatened by dynamic reverse engineering. While the strength of them is just opposite, so the combination of them in the proposed technique provides a lot more security level by making all types of reverse engineering much harder for attacker and debugging tools while performance is reduced by 15% on an average for these small programs. For large programs(time consuming) it will not matter too much performance loss. As we can see for bubble sort performance reduction is just 2.8% while reduction of 33.3% for quick sort.

According to the proposal it can be concluded that the combination of code splitting and insertion of self-modifying code, while complementing each other against all kind of reverse engineering, provides a much stronger but lighter (in terms of performance reduction in execution) obfuscation techniques available today.

5.2 Limitation & Future Scope

After more than one decade of research on code obfuscation by all the researchers through out the world, none of the proposed technique is fully compatible with parallel processing where code or data are being shared between multiple threads. This proposed research technique is not also to much suitable for parallel processing. It can support parallel processing until only the client code section is written in parallel otherwise reduction in performance will be huge as for, every thread have to create their own shared memory section and it also need much more effort from the programmers for synchronizing each thread for accessing the shared memory data. One possible way, according to me, is that implementation of software transactional memory (STM) at binary level with the self-modifying code, which is much harder to implement, need in depth knowledge of assembly language and above all it will be hardware specific till now as lots of processor does not support STM.

——— END ———

Bibliography

[1] C. Cifuentes, *Reverse compilation techniques*. PhD thesis, Queensland University of Technology, 1994.

[2] H. J. Van Zuylen, *The REDO compendium: reverse engineering for software maintenance*. John Wiley & Sons, Inc., 1993.

[3] R. S. Pressman, *Software Engineering - A Practitionaer's Approach*. McGraw-Hill Higher Education, 2009.

[4] "Idapro debugger : Data rescue [Online]." `http://www.datarescue.com/`. Last Accessed: 03-08-2013.

[5] "Immunity debugger [Online]." `https://www.immunityinc.com/products-immdbg.shtml`. Last Accessed: 23-08-2013.

[6] "Olly debugger [Online]." `http://www.ollydbg.de/`. Last Accessed: 12-09-2013.

[7] "The legend of random - programming and reverse engineering [Online]." `http://thelegendofrandom.com/blog/tools`. Last Accessed: 15-09-2013.

[8] E. J. Chikofsky, J. H. Cross, *et al.*, "Reverse engineering and design recovery: A taxonomy," *Software, IEEE*, vol. 7, no. 1, pp. 13–17, 1990.

[9] G. Lee, J. Morris, K. Parker, G. A. Bundell, and P. Lam, "Using symbolic execution to guide test generation," *Software Testing, Verification and Reliability*, vol. 15, no. 1, pp. 41–61, 2005.

[10] D. Molnar, X. C. Li, and D. A. Wagner, "Dynamic test generation to find integer bugs in x86 binary linux programs," in *Proceedings of the 18th conference on USENIX security symposium*, pp. 67–82, USENIX Association, 2009.

[11] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pp. 231–245, IEEE, 2007.

[12] J. Newsome, D. Brumley, J. Franklin, and D. Song, "Replayer: Automatic protocol replay by binary analysis," in *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 311–321, ACM, 2006.

[13] P. Falcarin, C. Collberg, M. Atallah, and M. Jakubowski, "Guest editors' introduction: Software protection," *Software, IEEE*, vol. 28, no. 2, pp. 24–27, 2011.

[14] "Atari games corp. v. nintendo of america inc. - u.s. court of appeals, federal circuit." `http://digital-law-online.info/cases/24PQ2D1015.htm`. Last Accessed: 22-06-2013.

[15] "Sony computer entertainment inc. v. connectix corp. - u.s. court of appeals, ninth circuit." `http://digital-law-online.info/cases/53PQ2D1705.htm`. Last Accessed: 22-06-2013.

[16] "bnetd." `http://en.wikipedia.org/wiki/Bnetd`. Last Accessed: 03-07-2013.

[17] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue, "Measuring similarity of large software systems based on source code correspondence," in *Product Focused Software Process Improvement*, pp. 530–544, Springer, 2005.

[18] S. S. Baboo and P. V. Bhattathiripad, "Software piracy forensics: The need for further developing afc," in *Digital Forensics and Cyber Crime*, pp. 19–26, Springer, 2011.

[19] T. Lancaster and F. Culwin, "A comparison of source code plagiarism detection engines," *Computer Science Education*, vol. 14, no. 2, pp. 101–112, 2004.

[20] J. M. Walker Jr, "Protectable nuggets: Drawing the line between idea and expression in computer program copyright protection," *J. Copyright Soc'y USA*, vol. 44, p. 79, 1996.

[21] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," tech. rep., Department of Computer Science, The University of Auckland, New Zealand, 1997.

[22] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *Software Engineering, IEEE Transactions on*, vol. 28, no. 8, pp. 735–746, 2002.

[23] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot, "White-box cryptography and an aes implementation," in *Selected Areas in Cryptography*, pp. 250–270, Springer, 2003.

[24] B. Anckaert, B. De Sutter, and K. De Bosschere, "Software piracy prevention through diversity," in *Proceedings of the 4th ACM workshop on Digital rights management*, pp. 63–71, ACM, 2004.

[25] G. Wroblewski, *General Method of Program Code Obfuscation (draft)*. PhD thesis, Citeseer, 2002.

[26] S. Schrittwieser and S. Katzenbeisser, "Code obfuscation against static and dynamic reverse engineering," in *Information Hiding*, pp. 270–284, Springer, 2011.

[27] E. Teran, "Edb debugger : Codef00 [Online]." `http://www.codef00.com/projects#debugger`. Last Accessed: 27-03-2014.

[28] C. Wang, *A security architecture for survivability mechanisms*. PhD thesis, University of Virginia, 2001.

[29] V. Balachandran and S. Emmanuel, "Potent and stealthy control flow obfuscation by stack based self-modifying code," *Information Forensics and Security, IEEE Transactions on*, vol. 8, no. 4, pp. 669–681, 2013.

[30] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[31] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[32] W. A. Harrison and K. I. Magel, "A complexity measure based on nesting level," *ACM Sigplan Notices*, vol. 16, no. 3, pp. 63–74, 1981.

[33] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *USENIX Security Symposium*, pp. 275–290, 2007.

[34] Z. Wang, C. Jia, M. Liu, and X. Yu, "Branch obfuscation using code mobility and signal," in *Computer Software and Applications Conference Workshops (COMPSACW), 2012 IEEE 36th Annual*, pp. 553–558, IEEE, 2012.

[35] L. Shan and S. Emmanuel, "Mobile agent protection with self-modifying code," *Journal of Signal Processing Systems*, vol. 65, no. 1, pp. 105–116, 2011.

[36] M. Madou, B. Anckaert, P. Moseley, S. Debray, B. De Sutter, and K. De Bosschere, "Software protection through dynamic code mutation," in *Information Security Applications*, pp. 194–206, Springer, 2006.

[37] J. Jackson, "Introduction into windows anti-debugging [ONLINE], month = sep, year = 2008, url = http://www.codeproject.com/Articles/29469/Introduction-Into-Windows-Anti-Debugging."

[38] J. Jackson, "An anti-reverse engineering guide [ONLINE], month = nov, year = 2008, url = http://www.codeproject.com/Articles/30815/An-Anti-Reverse-Engineering-Guide."

[39] 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*, pp. 536–546, ACM, 2010.

[40] J. B. Rosenberg, *How Debuggers Work - Algorithms, Data Structure and Architecture*. John Wiley & Sons, INC., 1996.

[41] D. Marshall, "Ipc:shared memory [ONLINE], month = may, year = 1999, url = http:http://www.cs.cf.ac.uk/Dave/C/node27.html."

[42] G. Silberschatz, Galvin, *Operating System Concepts*. Wiley India Pvt Ltd, 2009.

[43] "Intel 64 and ia-32 architectures software developer manuals." `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html?iid=tech_vt_tech+64-32_manuals`. Last Accessed: 24-01-2014.

[44] "x86 [wikipedia]." `http://en.wikipedia.org/wiki/X86`. Last Accessed: 27-03-2014.