KTH Computer Science
and Communication

Control Flow Graph Based Attacks

In the Context of Flattened Programs

ZHEN LI

Master's Thesis at NADA

Supervisors: Professor Hanne Riis Nielson, Technical University of Denmark
Professor Mads Dam, Royal Institute of Technology
Examiners: Professor Mads Rosendahl, Roskilde University
Professor Johan Håstad, Royal Institute of Technology

September 2014

Abstract

This report addresses de-obfuscation on programs. The targeted obfuscation scheme is the control flow flattening, which is an obfuscation method focusing on hiding the control flow of a program. This scheme introduces a special block named *dispatcher* into the program. The control flow of the program is reconstructed to be directed back to the dispatcher whenever the execution of a basic block ends. By doing this, in the flattened program, each basic block could be recognized as a precursor or a successor of any other basic blocks. While the real control flow of the program is merely disclosed during the execution of the program.

This report aims to remove the dispatcher added in the flattened program and rebuild the control flow of its original program. To achieve the targets, this report presents a de-obfuscation model based on the *Control Flow Graph* of an obfuscated program. The de-flattening model makes use of both static analysis and dynamic analysis.

The de-flattening model primarily relies on execution paths which are obtained by executing a program dynamically. The idea is that in the execution paths, after eliminating the dispatcher block, the real control flow of the original program is disclosed. Then based on these real execution paths, the control flow of the program without obfuscation could be constructed.

In order to obtain the full program structure, we need to gather the execution paths that result in a full coverage of the program. Merely with dynamic analysis, this could hardly be achieved. Therefore, static analysis are introduced. In the de-flattening model, the execution paths within a program are computed with the assistance of *dynamic execution path analysis*, which is a study to statically compute the feasible paths in a program by solving logical formulas obtained during the exploration of the program code. With this static analysis method, the model is adequate to reverse the flattened program to its original structure.

The obfuscated programs are distributed in binaries, our research provides insights to de-obfuscation on binaries directly. Besides, the de-flattening result obtained in the report is valuable for improvements to existing code obfuscation techniques.

Acknowledgements

The work was carried out in Technicolor R&D France. I would like to thank my company advisors Antoine Monsifrot and Charles Salmon-Legagneur for their guidance in this project and with my professional development. Next I would like to thank my university supervisors Prof. Hanne Riis Nielson and Prof. Mads Dam. They gave me excellent feedbacks and supports throughout the course of this project. Without their efforts, the work would not have been as quickly completed and as enjoyable.

I would also like to thank the NordSecMob consortium who offered me the great chance to study in the NordSecMob master program. Joining this program was definitely one of my best decisions I had ever made. I appreciate the friendship with the NordSecMob students. Special thanks to Anusha Sivakumar, Kamran Manzoor and Nikita Martynov. I would like to thank their encouragement and assistance during the whole process.

Contents

Chapter 1

Introduction

In recent years, with more and more software distributing over the Internet, the importance of software security is emphasized.

On one hand, the protection against malicious software such as viruses, worms is important in this open environment. To protect against malicious software, virus scanners, firewalls are developed. In their detection of viruses, the classic detection techniques statically look for the presence of signatures. A signature is a sequence of code that serves to identify a special virus. However some newer viruses have evolved to be able to shield their signatures against virus scanners by using *code obfuscation*. Take the polymorphic viruses [SSCS07] as an example, this type of viruses adopts code obfuscation to keep mutating their code every time they run without changing the code function. Therefore, in order to capture obfuscated viruses, the virus scanners should be able to undo the obfuscation in the first place. The relationship between the virus writer and the virus scanner developer could be described as the competitors in an obfuscation-de-obfuscation race [CJ06], where the virus writer is the code obfuscator and the virus scanner developer is the code de-obfuscator.

On the other hand, malicious users have become a threat to software owners. When software is distributed to a user, the software owner loses the control of how the user would use the software. In this case, the software could be exposed to attacks by a malicious user. In an untrusted user environment, examples of at-

tacks by a malicious user consist of grabbing confidential data from the software, inspecting the intellectual property such as algorithms used in the software, and tempering the software for illegal use such as hacking and piracy, etc [Cap12]. These attacks are carried out based upon the knowledge gained about the software internal. To protect the software internals being examined and modified by a malicious user, the same techniques, code obfuscation schemes, could be used by the software owner. However in this case, the obfuscator is the software owner while the de-obfuscator is the malicious user.

In spite of the dissimilarities between the protection against malicious software and the protection against malicious users, similar techniques are used to achieve the same goals. For obfuscators, code obfuscation is used to protect software internals from being inspected. De-obfuscators, on the other hand, resort to de-obfuscation to remove the obfuscation inserted or change the software code into a format that is easy for analysis.

1.1 Code obfuscation

Code obfuscation transforms a program into a format that is "difficult" to understand by making the task of reverse engineering expensive in terms of time or resource consuming to do so [UDM05]. Since it is difficult to gain knowledge of the program internals, the content and the structure of the software program are shielded from reverse engineering and tampering.

The use of code obfuscation is not limited. It could be either used by a virus writer to subvert virus scanners, or applied by a software owner to protect software inner structure against reverse engineering attacks. However the unlimited use of code obfuscation raises two related questions: (1) What techniques are useful to understand code obfuscation programs? For example, a de-obfuscator e.g., a virus scanner, might be interested in the analysis techniques that are available to help them detect obfuscated programs e.g., obfuscated viruses. (2) Is it possible to remove the code obfuscation that is inserted in an obfuscated program? For instance, if a de-obfuscator, e.g., a malicious software user, could eliminate the code obfuscation schemes and reverse engineer the obfuscated program to its original, then the obfuscator, e.g. a software owner, should consider a new protection.

1.2 The aims of the research

This report focuses on answering the proposed questions. In this report, concentrating on the obfuscation schemes on program structure, we present *Control Flow Graph* (CFG) based de-obfuscation attacks that are effective in eliminating the obfuscation on program structure. Our research addresses de-obfuscation on binary files. It is effective in obtaining a CFG which reveals the original program structure. This CFG is suitable for advanced studies based on the structure of obfuscated programs. Besides, our research evaluates the strengths and weaknesses in current obfuscation designs. It suggests new views for improvements in future code obfuscation studies.

1.3 Outline

The rest of the report is organized as follows: In Chapter 2, we introduce several important theories and concepts for understanding the report, including the ideas and techniques related to obfuscation, de-obfuscation and reverse engineering. Then in Chapter 3, we begin our de-obfuscation by presenting the targeted obfuscation schemes and the base de-obfuscation model. In the design of this base model, we primarily employ CFGs constructed with dynamic analysis. After the base model, the corresponding evaluation of it against several program examples could be found in Chapter 4. Then in Chapter 5, we discuss enhancements that could be inserted to improve the de-obfuscation method presented in Chapter 3 by introducing static program analysis. It is followed by the evaluation of the improved de-obfuscation method in Chapter 6. Finally Chapter 7 concludes the report with a summary of achievements and several directions for further work.

Chapter 2

Background

To aid in understanding the following discussion, this chapter represents several important theories and concepts. In the discussion, we assume that the reader already has a good knowledge in compilers and assembly language. The reader is familiar with the definition of source code, machine code, basic blocks and is able to read simple assembly code in Intel syntax and/or AT&T syntax.

We begin our discussion with three representations of programs, namely source code, assembly code and machine code. They defines different levels of analysis and transformations that could apply on a program. Then we present the CFG which is a graphical representation of a program. This graphical representation enables a universal description for a program at all levels. After that, we focus on three important concepts, namely obfuscation, de-obfuscation and reverse engineering. The relationship of these three concepts could be described as follows: The obfuscation is the operation that transforms a program into a format that is hard to understand. While the de-obfuscation is the opposite transformation of obfuscation. In the course of de-obfuscation, to gather essential information, reverse engineering techniques are deployed. In the course of the discussion, we take a closer look into each topic in depth.

2.1 Program representations

Software programs could be represented at different levels, namely high level, intermediate level and low level. Each level corresponds to several types of code:

- High level: source code
- Intermediate level: bytecode
- Low level: assembly code and machine code

From top to bottom, the syntaxes complexity of program code decreases while the difficulty to comprehend by humans increases. Source code at high level is usually written in advanced program languages such as C/C++, C#, Java. The code at high level is usually straightforward to read and to understand by programmers. While at low level, machine code is coded in binary (0s and 1s). The code in a machine language could be comprehended and executed by a computer directly. At the same level, assembly code is still human-readable, however, it almost has a direct mapping to machine code. Between high level and low level, there is another intermediate level. The complexity of intermediate code falls between source code and machine code.

The code shown in Listing 2.1, Listing 2.2 and Listing 2.3 gives examples of source code, assembly code and machine code (in hexadecimal format), which are the representations to be used in this report. The source code is mainly given in C and the assembly code is written in AT&T syntax (x86 platform, Linux operating system). For the reader who is only familiar with assembly code in Intel syntax, the AT&T syntax differs from Intel syntax mainly in two aspects: (1) In each AT&T assembly instruction, the destination operand comes after all source operands; (2) With AT&T syntax, registers start with % and constant numbers start with $. A more detailed discussion of the differences between AT&T syntax and Intel syntax could be found in [Ass].

Listing 2.1: Function GCD in code.c

```
int GCD(int x, int y) {
        while (1) {
                int m = x % y;
                if(m == 0) return y;
                x = y;
                y = m;
        }
}
```

Listing 2.2: Function GCD in code.s

```
GCD:
L0:
        push   %rbp
        mov    %rsp,%rbp
        mov    %edi,-0x14(%rbp)
        mov    %esi,-0x18(%rbp)
L1:
        mov    -0x14(%rbp),%eax
        mov    %eax,%edx
        sar    $0x1f,%edx
        idivl  -0x18(%rbp)
        mov    %edx,-0x4(%rbp)
        cmpl   $0x0,-0x4(%rbp)
        jne    L2
        mov    -0x18(%rbp),%eax
        pop    %rbp
        retq
L2:
        mov    -0x18(%rbp),%eax
        mov    %eax,-0x14(%rbp)
        mov    -0x4(%rbp),%eax
        mov    %eax,-0x18(%rbp)
        jmp    L1
```

Listing 2.3: Function GCD in code.o

```
55 48 89 E5 89 7D EC 89  75 E8 8B 45 EC 89 C2 C1
FA 1F F7 7D E8 89 55 FC  83 7D FC 00 75 05 8B 45
E8 5D C3 8B 45 E8 89 45  EC 8B 45 FC 89 45 E8 EB
D9
```

2.1.1 Transformation among representations

The different representations of programs are not strictly independent. In fact, it is quite common to change the program representations from one to another. This subsection presents the techniques for transforming programs among three main representations, namely source code, assembly code and machine code.

Compilation: from source code to machine code

The transformation from high level code to lower level code usually could be achieved with no extra effort by using a compiler. As we all know, in order to execute a program on a local machine, the source code of the program has to be converted to machine code. This conversion from source code to machine code is done by using a compiler or an interpreter. In the process of compilation, instead

of converting source code directly into machine code, compilers first convert source code into certain intermediate code to ease optimization or to increase portability. Finally the machine code is generated when the optimization finishes or when the intermediate code is interpreted.

Disassembling: from machine code to assembly code

The transformation from machine code to assembly code could be done with a disassembler. A disassembler is a program that converts machine code into assembly code. As assembly language almost has a direct mapping to machine language, a disassembler is not hard to build. Example applications of disassemblers consist of *OBJDUMP*, *GDB*[gdb] for Linux and *IDA*[ida] for Windows. With Linux operating systems, we could obtain the assembly code from a binary file either with *OBJDUMP* using the simple command given bellow:

```
linux> objdump -d code.o
```

Or with *GDB* to obtain a certain piece of code for a special function using the commands given as follows:

```
linux> gdb code.o
gdb> disassemble GCD
```

With a disassembler, the machine code becomes human-readable as assembly code. In the rest of the report, for simplicity, we will represent binary programs in assembly directly. With the information given here, we assume that the reader could convert programs between these two program representations easily.

Decompilation: from assembly code to source code

The converter from assembly code to source code is hard to build. Though some efforts have been made to build such kind of tools [BMM06] [Boo], there has not been any satisfying converters that could get the original source code from its compiled assembly code till today. The difficulties in building such converter lie in finding general pattern matching between assembly code and its original source code. Therefore, when reverse engineering a binary program, the analysis is frequently performed at assembly code level rather than source code level.

2.2 CFG

This section introduces the CFG, which describes a program in terms of a directed graph. The uses of a CFG consists of illustrating the structure of a

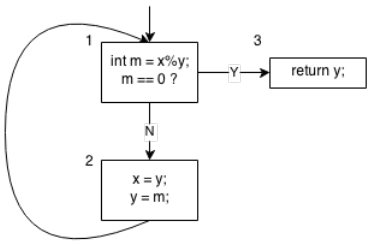
program and assisting analysis that was based on program control flow. For the former use, we take the construction of a CFG from assembly code to show how it could be used to describe the program structure. For the second use, we give graph isomorphism as an application example to demonstrate how to recognize programs by program control flow with program CFGs.

2.2.1 Definition

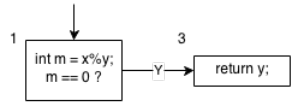
A CFG is given by a directed graph $G = (V; E)$, where V is the set of basic blocks and $E \subseteq V \times V$ is the set of directed edges defining control flow between basic blocks. As a reminder, a basic block is a portion of instructions or statements in a program with only one entry and one exit. A program CFG statically shows all the possible execution paths in the program. An execution path consists of a sequence of instructions or statements that would appear in one program process.

For the program shown in Listing 2.1, the CFG of it is constructed as the graph in Figure 2.1(a). From this figure, we could conclude that there are three basic blocks in this program. Following the directed edges, we could construct possible execution paths within this program. Figure 2.1(c) illustrates two possible execution paths. Based on the simple execution path p1 in the figure, a sub-CFG, which only covers part of the basic blocks in the program CFG, is constructed in Figure 2.1(b).

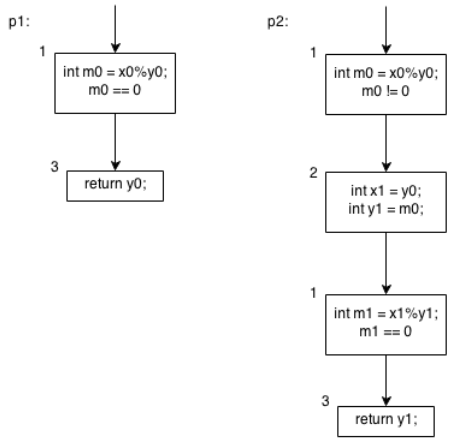
A CFG describes a structural abstraction of a program. It leaves out language specific details but maintains structural control flow within a program. Program control flow describes how a program is organized and how data could be passed from one part to another. Thus, a CFG is entitled to represent the semantic structure of a program. Besides, CFGs are extremely suitable for analysis that is primarily based on semantic aspect (structure) of programs, resulting from that program control flow varies little among different program language implementations. Considering the C program in Listing 2.1, we write the same program in Java. The CFG build from another different language code, in this case, maintains the exact same structure as the CFG shown in Figure 2.1(a). As another example, we compile the code in assembly language. The CFG constructed from the assembly code (Figure 2.2) also maintains a very similar structure as the CFG constructed from the source code.



(a) The program CFG



(b) The sub-CFG



(c) Two execution paths

Figure 2.1: CFGs and execution paths constructed from the program in Listing 2.1

2.2.2 Construction of CFGs

We've seen CFGs constructed from source code and assembly code. In this subsection, we present the idea for constructing such CFGs from programs. When the CFGs are constructed, the inner structure is hereby exposed.

The basic algorithm

In this report, we mainly consider CFGs constructed from assembly code. Given a piece of assembly code, the construction of a program CFG consists of two topics: dividing nodes and adding directed edges.

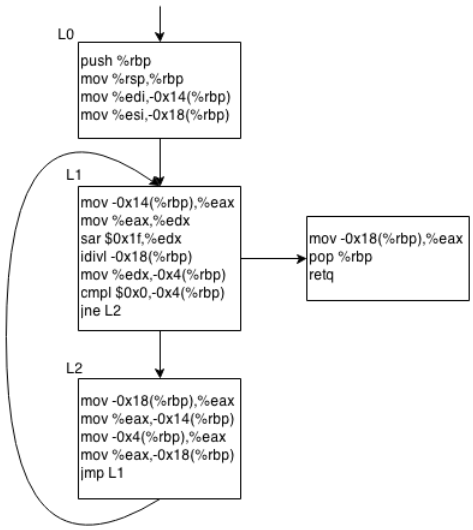


Figure 2.2: The CFG constructed from the assembly code in Listing 2.2

A node is a basic block in the code. In assembly code, a basic block starts at

1. label instructions,
2. instructions after jump instructions;

and ends at

1. jump instructions,
2. instructions before label instructions.

Then directed edges are constructed from jump instructions. Conditional jumps specify two directed edges

1. one from current node to its direct successor,
2. another from current node to the node where the program control flow want to jump to;

while unconditional jumps define one direct edge

1. from current node to its direct successor.

The algorithm for the CFG construction from assembly code is given by scanning the code once and reasoning on *jump* instructions and labels. As it is simple, we will not list the algorithm details here. More information could be found in [XSS09] [BKH05].

Improvements

The basic construction algorithm above is complete. However, the CFG constructed could be improved. The problem with the CFG constructed is that its structure is not as clear as that of the CFG constructed from source code. This problem results from the compilation.

For example, in the compilation procedure, a big *switch* statement will be compiled into several small successive *jump* instructions. In the CFG constructed from source code, the *switch* statement is put in one block, while in the CFG constructed from assembly code, it could be split into a main *switch* block followed by a sequence of small basic blocks. Each of the small basic blocks is made of a *cmp* instruction and a *jump* instruction. Therefore, in a case where we are only concerned about the structure of programs, we could consider removing all the basic blocks that are made of *cmp-jump* instructions. However, while removing them, original edges should be kept.

This improved algorithm simplifies the CFG without losing any program structural information. However this improved algorithm will not guarantee that the structure of CFGs from assembly code and source code would be the same in any case. It only simplifies the structure for successive branches. A CFG built with this improved algorithm is called a simplified CFG.

2.2.3 Graph isomorphism

A CFG is a directed graph, therefore all the theories of directed graphs are applicable to it. In this subsection, we introduce graph isomorphism, which is a study about an equivalence relation on graphs. The graph isomorphic theory plays a central role in recognizing programs by program control flow.

The isomorphism of two directed graphs G and H is given by a bijection relation $f : V(G) \to V(H)$, where $V(G)$ and $V(H)$ are the vertex sets of G and H, such

that for any vertex pair u, v of G, there is $u \to v$ in G if and only if there is $f(u) \to f(v)$ in H.

This theory indicates two isomorphic graphs are identical on structure despite differences in their drawings. For small graphs, it might be easier to decide whether two graphs are isomorphic by looking at them directly. However for complex graphs, it is definitely better to rely on a computer to do this for us. For most types of graphs, the algorithm to determine graph isomorphism is in the low hierarchy of class NP [Joh05]. Currently the best theoretical algorithm is based on Luks's earlier work [Luk82]. The implementations of graph isomorphism algorithm based on Luks' work could be found in several graph library such as JGraphT [JGr] for Java language.

The isomorphism of CFGs can be used to verify whether two pieces of code come from the same source. If two programs come from the same source (e.g., one comes from the source code, another comes from the assembly code after the source code is compiled), the program control flow varies little, thereby for most cases, the CFGs built from them should be isomorphic. Therefore if we already know one program, then we could use this technique to recognize programs from the same source.

2.3 Code obfuscation

As mentioned previously, code obfuscation is a set of transformations that are made on programs to make the code or the execution of the program difficult to analyze. It could be employed to protect the intellectual property of program owners and to prevent vulnerabilities in a program from being detected by untrusted users. However, it is also important to note that it might be misused in virus to avoid detection by virus scanners. In this section we focus on the obfuscation added at source code level - the level which provides most flexibility for obfuscators to make changes on a program.

2.3.1 Surface obfuscation and deep obfuscation

Code obfuscation at source code level could be classified into two broad classes: surface obfuscation and deep obfuscation, as suggested by Udupa et al.[UDM05].

Surface obfuscation focuses on modifying the representation of the code to confuse a programmer while reading it. However it applies no change on the pro-

gram structure. Types of surface obfuscation include renaming of variables and functions, and inserting or deleting whitespace.

Deep obfuscation, on the other hand, concentrates on changing the control flow of a program. Deep obfuscation methods conceal the original control flow by breaking the original program structure into several parts and then reassembling them into a new structure which is quite different from the original one. A representative deep obfuscation technique is the *control flow flattening*.

2.3.2 Control flow flattening

Control flow flattening is a basic but effective deep obfuscation technique. It is usually added on each function in a program respectively. It obscures the control flow of a function in a program by constructing all the basic blocks to have the same set of predecessors and successors. The real control flow will not be disclosed until the execution.[UDM05] In the rest of this subsection, we represent the basic control flow flatting technique with two variants. They are originally disclosed in Wang's dissertation [Wan00] [WHKD01] [WHKD00] and summarized by Udupa et al. [UDM05]. These obfuscation schemes are also considered in several studies [UDM05] [Cap12] [CGJZ01]. In the discussion, we will adopt CFGs to assist highlighting the changes on program control flow. Besides, for simplicity, we assume that each program is only made of one function.

Basic control flow flattening

The basic flattening scheme inserts a new switch block named *dispatcher*. The idea is that every time when the execution of a basic block finishes, the execution of the program is directed back to the dispatcher to inquire the next basic block to executed. This basic block is then selected based on the value of a dispatcher variable v. Its value varies after the execution of basic blocks. In a simple case, the new value of v could be disclosed at the last statement in each basic block.

As an example, consider the program in Figure 2.1(a). We present the program after flattening in Figure 2.3(a) where D is the dispatcher block and v is the dispatcher variable. On the top of the dispatcher block D, we add a block *init* for dispatcher variable initialization. This block is essential as it routes the program execution back to the original program entry. At the bottom, we insert an empty basic block. It is added to simplify the drawing of this figure. It also indicates that the execution is directly routed back to the dispatcher after the execution of its previous basic blocks.

In a traditional program, most basic blocks select next basic block from 1 or

2 successors. However, in a flattened program, after the dispatcher block, any basic block could be selected as the next basic block where the program continues. Suppose there are n basic blocks out of the dispatcher, then in theory the possible execution paths would be $O(n^k)$ where k donates the number of executed blocks. [Cap12] By doing this, the flattening stops an attacker from predicting the control flow of a flattened program.

Flattening with artificial blocks

The basic flattening method could be enhanced by adding artificial blocks. These artificial blocks might never be executed in any real executions. However it might be hard to detect by a de-obfuscator who analyzes the program statically, as a result, the computation complexity to find the next basic block to execute is expanded dramatically.

Figure 2.3(b) shows an example result of applying flattening with artificial blocks to the program in Figure 2.1(a). There is no limitation on how many artificial basic blocks that could be added. However the more fake code added, the bigger the size of the program will be.

Consider the following application scenario. A de-obfuscator might want to statically analyze all the possible values of the dispatcher variable v in a flattened program. After introducing artificial blocks, the static analysis becomes harder, as not only more basic blocks but also more fake assignments of v should be analyzed.

Flattening with interprocedural information passing

The value of the dispatcher variable v could be hard-coded as it appears in Figure 2.3(a). In this case, the value of v at the exit of each basic block is still predictable by examining the basic block locally. To avoid the value of v being disclosed, the basic flattening method could be improved by using interprocedural information passing.

Figure 2.3(c) displays an example of flattening with interprocedural information passing on the program from Figure 2.3(a). In this scheme, the value of v is passed using a global array. The global array is initiated at each call of the program with a variable w. The variable w which is a random number, marks the start of the meaningful content in the global array. The content of the global array varies every call since w changes at different call of the program. As this method introduces a global array and randomness into analysis, it increases the difficulty to reverse engineer.

Cappaert [CP10, Cap12] summaries some other enhancements to conceal the

values of v. For example, one-direction hash function could be used to make the computation unable to reverse.

2.4 De-obfuscation

De-obfuscation is the opposite operation of obfuscation. It aims to eliminate the obfuscation added in programs. In this section, we present a short discussion of the de-obfuscation on surface and deep obfuscation.

2.4.1 Elimination of surface and deep obfuscation

Eliminating surface obfuscation is easy to achieve since essentially the obfuscation transformations are carried out on the syntax aspects of programs and thereby most of the changes could be undone by designing a parser. For example, "Dotfuscator" [Dot], which is a tool designed for obfuscating .NET code , obfuscates a program by renaming the variables in different scopes to the same identifier. For this type of obfuscation, the identifiers of these variables could be easily re-assigned by using a parser following the scope rules of the language.

Deep obfuscation is harder to work around than surface obfuscation as it requires reasoning on the semantic structure of the program. To work around deep obfuscation, e.g. restoring the original program control flow, information about the relationship among different parts in a program should be gathered, which could be done with the assistance of reverse engineering.

2.5 Reverse engineering

Generally, reverse engineering (a.k.a reverse engineering analysis) could refer to all the processes of collecting necessary information of one closed system in order to reconstruct it by analyzing its components and workings [EJPJ]. It fetches program information that could be further employed to assist performing de-obfuscation transformations, as well as other modifications such as bug fixing, malicious code insertion. Based on whether the analysis requires the execution of a program, reverse engineering could be generally classified into two types of analysis: dynamic analysis and static analysis. In the rest of this subsection, we look into each of them in detail and discuss several applications that are essential to our research.

2.5.1 Dynamic analysis

Dynamic analysis (a.k.a dynamic program analysis), collects software information during the execution of a program on a real or virtual processor. The information records certain aspects of the program run-time state. By examining the information collected, dynamic analysis derives the properties that hold for one or more executions.

The advantage of this analysis is the precision of information collected by it. In dynamic analysis, the information is recored during the execution of a program. The information describes precisely the program run-time state which is essential to address a certain problem. For example, in order to detect the data structure used by a program, we could examine how the space is assigned for a new object in the program storage.

However dynamic analysis suffers from the code coverage problem [AO08]. This problem happens because execution paths rely highly on the inputs to the program. An execution path records the sequential statements that are explored in an execution of the program. Given different inputs, the observed execution path (statement sequence) might vary, resulting in dissimilar program behavior being observed. In order to ensure that ample interesting program behavior is observed, disjoint inputs might be expected. The study on selecting representative inputs to steer execution into new execution paths is called *dynamic symbolic execution* (a.k.a. smart white-box fuzzing), which is actually a static analysis technique. With the help of dynamic symbolic execution tools, the code coverage problem could be partially overcome. More basic concepts of dynamic analysis could be found in [Bel99] [SS02].

Dynamical exploration of a program with *GDB*

After the introduction to the basic idea of dynamic analysis, we present an application about how to explore a program dynamically with *GDB* on Linux. Given an executable program file (.o file) and inputs to the program, we could use *GDB* to "execute" the program and obtain an execution path which records all the assembly instructions explored. In previous sections, we have already introduced *GDB*, which could be employed to translate machine language to assembly language. However it provides far more functions for debugging a program dynamically besides disassembling.

Taking the program shown in Listing 2.1 as an example again, we demonstrate how to obtain a program execution path with *GDB*. While using it for debugging a program dynamically, we should first specify a program, an interested function and program arguments. This is done by using the commands given bellow.

```
linux> gdb code.o
gdb> break *GCD
gdb> set args 2 4
```

This piece of commands shows that the input file is code.o, the interested function is GCD and the inputs to the program are 2 and 4. Then the following commands ask *GDB* to list each executed assembly instruction in terminal during the run time.

```
gdb> display/1i $pc
```

Then we start the program and keep on running the program to next instruction using the following commands:

```
gdb> r
gdb> ni (many times)
```

Finally, all the assembly instructions on the explored execution path is shown in Listing 2.4. This call explores the execution path p2 shown in Figure 2.1(c).

Listing 2.4: An execution path

```
0x400561 <GCD>:          push   %rbp
0x400562 <GCD+1>:        mov    %rsp,%rbp
0x400565 <GCD+4>:        mov    %edi,-0x14(%rbp)
0x400568 <GCD+7>:        mov    %esi,-0x18(%rbp)
0x40056b <GCD+10>:       mov    -0x14(%rbp),%eax
0x40056e <GCD+13>:       mov    %eax,%edx
0x400570 <GCD+15>:       sar    $0x1f,%edx
0x400573 <GCD+18>:       idivl  -0x18(%rbp)
0x400576 <GCD+21>:       mov    %edx,-0x4(%rbp)
0x400579 <GCD+24>:       cmpl   $0x0,-0x4(%rbp)
0x40057d <GCD+28>:       jne    0x400584 <GCD+35>
0x400584 <GCD+35>:       mov    -0x18(%rbp),%eax
0x400587 <GCD+38>:       mov    %eax,-0x14(%rbp)
0x40058a <GCD+41>:       mov    -0x4(%rbp),%eax
0x40058d <GCD+44>:       mov    %eax,-0x18(%rbp)
0x400590 <GCD+47>:       jmp    0x40056b <GCD+10>
0x40056b <GCD+10>:       mov    -0x14(%rbp),%eax
0x40056e <GCD+13>:       mov    %eax,%edx
0x400570 <GCD+15>:       sar    $0x1f,%edx
0x400573 <GCD+18>:       idivl  -0x18(%rbp)
0x400576 <GCD+21>:       mov    %edx,-0x4(%rbp)
0x400579 <GCD+24>:       cmpl   $0x0,-0x4(%rbp)
0x40057d <GCD+28>:       jne    0x400584 <GCD+35>
0x40057f <GCD+30>:       mov    -0x18(%rbp),%eax
0x400582 <GCD+33>:       pop    %rbp
0x400583 <GCD+34>:       retq
```

2.5.2 Static analysis

Static analysis (a.k.a static program analysis), on the other hand, performs analysis without any execution of a program. The analysis is totally based on the reasoning on the semantic aspect of program code. Static analysis explores all the possible behavior that might arise during the execution. It is widely used in the circumstance where the simulation of the real environment in a test bed is difficult or expensive. Examples include the software used in aircraft and nuclear plant.

The usefulness of static analysis mainly lies in two related aspects: simple error detection and formal methods.

Discovering simple defects in programs is one of the main use of static analysis. A very simple but representative application in this aspect is the compiler. A compiler employs static analysis techniques to optimize code such as detecting dead code and analyzing live variables [Cap12]. More sophisticated defects could be found in a program with the assistance of professional tools such as *FxCop* [FxC], *Lint* [Lin], *FindBugs* [Fin].

The use of static analysis can be as complex as formal methods where mathematical techniques are employed to prove certain program properties. An example application of formal methods is to work as a complement to program testing to ensure certain correct system behavior. For example, in the design of a compiler, in order to ensure the compiler always behaves as expected, besides rigorous testing, it is a favorable practice to use formal methods such as formal logic to theoretically prove its correctness.

In order to achieve a precise and valid analysis result, the computation could be very time consuming. Most of the static analysis problems, e.g., finding all the errors in a program, are undecidable. Thus to save analysis time, the static program analysis is frequently applied on program slices excised from a large program. Further information on this static analysis could be found in [GMGM] [NNH99].

Dynamic symbolic execution

Dynamic symbolic execution which is introduced to overcome the code coverage problem in dynamic analysis in the previous section also belongs to static analysis, as formal methods play a central role in it. It collects explored execution paths in a program as mathematical formulas and computes the inputs to different execution paths by solving the formulas.

For example, consider a simple example at source code level. We construct a program execution path by unfolding the while loop in the program given in Listing 2.1 twice (the long execution path p2 in Figure 2.1(c)). The unfolded program is shown in Listing 2.5, where we could find that the loop is exited in the second iteration.

Listing 2.5: Unfoled GCD

```
int GCD(int x0, int y0) {
        int m0 = x0 % y0;
        assert(m0 != 0);
        int x1 = y0;
        int y1 = m0;
        int m1 = x1 % y1;
        assert(m1 == 0);
        return y0;
}
```

The corresponding path formula is therefore given by

$$(m0 = x0\%y0) \wedge \neg(m0 = 0) \wedge (x1 = y0) \wedge (y1 = m0) \wedge (m1 = x1\%y1) \wedge (m1 = 0)$$

The path formula is then passed to a *Satisfiability Modulo Theories* (SMT) solver for computation. A SMT solver checks the satisfiability of a given logical formula in the context of some background theory e.g., the arithmetic theory. A logical formula is satisfiable if there is an interpretation that makes the formula true. The path formula given above is satisfiable since the interpretation

$$x0 \mapsto 2, y0 \mapsto 4, x1 \mapsto 4, y1 \mapsto 2, m1 \mapsto 0$$

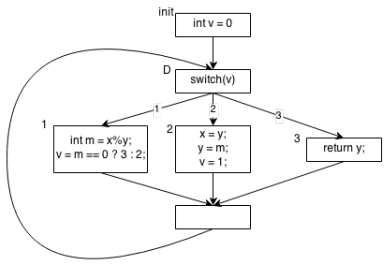
could make the formula true. Therefore this formula is satisfiable. This result indicates that if we call GCD(2, 4), the execution path where the loop is unfolded twice will be explored. This example is originally from a tutorial for *Z3* [dMB10] which is a SMT solver developed by Microsoft Research. However, the use of this static analysis is not limited to any program representations. It could also be applied on assembly code to compute the inputs for a certain execution path within a program, as we will see in Chapter 5.

Dynamic symbolic execution provides theoretical support to compute program inputs for different execution paths within a program. With the help of dynamic symbolic execution, the exploration of a program could become more efficient.

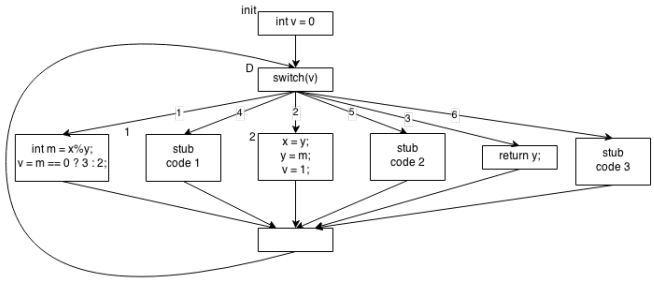
2.6 Summary

In this chapter, we have presented several important background information. First, we have discussed about program representations, namely source code, assembly code and machine code, and the transformations among them. We also introduce the graphical representation of programs - the CFG, as well as its construction from a program. In the rest of the discussion, programs might be presented freely in source code, assembly code, machine code, or even CFGs. The conversion from one format to another should already have been covered in this chapter.

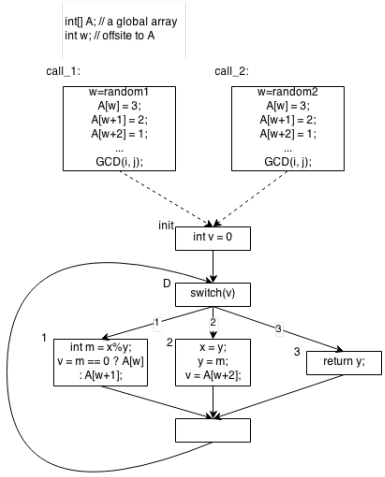
Second, we have focused on deep code obfuscation, especially the flattening approaches. We start from the fundamental basic control flow flattening and finish with improvements that could be implanted into the basic flattening. In the rest of the report, focusing on the control flow flattening, we will present a practical de-obfuscation method to remove flattening based on the program CFG. In the elimination of flattening, we will combine the static analysis and dynamic analysis to overcome the limitations and to expand the benefits of the two analysis methods.



(a) The CFG after basic contorl flow flattening



(b) The CFG of flattening with artificial blocks



(c) The CFG of flattening with interprocedural information passing

Figure 2.3: Flattened CFGs of the program in Listing 2.1

Chapter 3

The base model

As it is introduced in the previous chapter, code obfuscation could be classified into two types: surface obfuscation and deep obfuscation. This report is concerned primarily with deep code obfuscation - that is the code obfuscation on program control flow.

In this chapter, we provide the base CFG based de-obfuscation model. The aim of our research is to restore the program control flow before obfuscation. The targeted obfuscation techniques of our research are the basic control flow flattening and its variants. These obfuscation schemes are added at the source code level. However the programs to be analyzed are disseminated the binary format, as this is the only code format that is available for analysis in real cases.

Our discussion begins with the assumptions that are drawn from the target flattening schemes. The assumptions are the basis of our discussion. Then we prepare our discussion with the construction of CFGs from binary program files. After that, we make our road to take a closer look at the base model where the transformation from a flattened program to a de-flattened CFG is addressed. The base model relies highly on dynamic analysis.

3.1 Assumptions

This section presents the assumptions derived from the targeted flattening schemes. Based on these assumptions, the rules are drawn to detect the dispatcher block in a flattened program. In this section, all the discussion is at the source code level. In this context, a basic block refers to a basic block in source code or on the CFG generated from source code.

3.1.1 Assumptions

The attributes of the targeted flattening methods contribute to assumptions for our de-obfuscation model. These assumptions derived from the targeted obfuscation methods are listed below:

1. One and only one dispatcher block exists in each obfuscated program.
2. After executing a block, program execution is always directed back to the dispatcher to inquire the next block to executed.

3.1.2 Claims

Based on these assumptions, we could conclude one useful claim:

1. The dispatcher is the basic block that is visited most frequently in all execution paths.

The Claim 1 could be proved by using contradiction: Suppose the basic block B_k, which is visited most frequently in all execution paths, is a different basic block than the dispatcher block D. Then we define the visit count of block B_k as v_{B_k} and the visit count of the dispatcher D as v_D. Thus based on Assumption 2, we could deduce that $v_{B_k} \leq v_D$. An contradiction is detected and therefore the claim is proved.

Therefore we could draw a conclusion that the dispatcher in a flattened program is the basic block that is visited most frequently in all execution paths. Provided this analysis conclusion, we could perform de-obfuscation detection and transformations.

3.2 Binaries → CFG

The de-obfuscation model primarily works on CFGs of programs. The programs in our research are given in machine code. Therefore the construction of CFGs from binary files are the first step for further analysis and transformations. In this section, we deal with CFG construction to prepare our programs ready for analysis.

As the programs are disseminated in the binary format, the first step lies in reversing the machine code into assembly code. As we have discussed in Section 2.1.1, this could be done by utilizing a disassembler.

Then to acquire a CFG from its assembly code, the construction could be done either statically or dynamically. When converting statically, given all the assembly code of a program, the CFG is constructed based on the language syntax. If all the destinations of *jump* instructions are hard-coded, this construction procedure could be referred in Section 2.2.2. The CFG constructed in this way is complete as it originally covers each statement in the assembly code. However, if the *jump* destination is indirect, e.g. a *jump* destination given in a register, several edges might be missing on the CFG built statically.

The CFG could also be constructed dynamically. The idea is that an execution path of a program consists of all the assembly code executed in one run of a program. If we could execute the program for plenty times and gain various execution paths that result in a complete code coverage, we could then build the CFG of the program based on these execution paths. The execution paths, as suggested in Section 2.5.1, could be yielded with *GDB*. The construction algorithm is the same as the algorithm presented in Section 2.2.2, but one additional combination process is required to reorganize all basic blocks and control flows in different execution paths together. In the base model, we mainly utilize the dynamical construction approach so that we could avoid possible indirect *jump* instructions. At the same time, we could also make use of some hidden information in the execution paths.

3.3 Flattened CFG → de-flattened CFG

This section talks about the base model for reversing the structure of a flattened program. In this section, we first introduce the overall algorithm of de-flattening. Then we explain each step in the algorithm in depth.

3.3.1 The algorithm of de-flattening

First of all, the base model could be describes with the following algorithm:

Data structures:

1. An array P where all execution paths are stored.
2. The simplified CFG G constructed from P dynamically.

Algorithm:

1. Remove the dispatcher block which is the most-frequently-visited basic block in P and reconstruct the simplified CFG G from P dynamically.
2. Remove the cycles that are not introduced by loops on G.
3. Remove the redundant edges between successive basic blocks on G.

The basic idea of our method to build the de-flattened CFG is illustrated as follows: First, after the discussion of Section 3.1, we have already know that the dispatcher is the basic block that is visited most frequently. Thus, in the process of constructing a CFG dynamically, if we remove the basic block with such characteristics from all the execution paths, the real execution paths without the dispatcher are revealed. Therefore, based on the real execution paths, the CFG without the dispatcher could be constructed.

However in the process of dispatcher elimination, there are a few things that need to be further discussed. Our CFGs are constructed from assembly code, while the assumptions and claims are based on CFGs from source code. Even though we could use the improved CFG construction algorithm presented in section 2.2.2 to get simplified CFGs for analysis, CFGs from assembly code and source code could still differ. In the rest this section, focusing on the differences caused by compilation, we explain the proposed de-obfuscation algorithm in detail. The de-obfuscation is carried out in terms of three analysis, namely dispatcher analysis, redundant cycle analysis and redundant edge analysis.

3.3.2 Dispatcher analysis

The dispatcher analysis is not affected by compilation on the simplified CFG, since no mater in which format the program is presented, the piece of code in

the dispatcher is always the code executed most frequently.

The only potential problem with the dispatcher after compilation is that the dispatcher might be split into more than one block. As already discussed, in the source code, a dispatcher is usually presented in a big *switch* or a series of *if-else* statements [WHKD01]. When these statements are compiled into assembly code, they are split into one main block followed by several successive small *jump* blocks. However, with the simplified CFG (Section 2.2.2), these small successive blocks will be removed smoothly. Therefore, ideally the dispatcher should be unique in the whole simplified CFG.

Even if the dispatcher is not unique on the simplified CFG, the claim for dispatcher analysis is still suitable to detect one dispatcher block. Suppose on the simplified CFG, the dispatcher is made up of two basic blocks B_e and B_t. Then definitely one of the two basic blocks is the entry to the original dispatcher block. Suppose this block is B_e. Then the visit count of the entry block B_e is not affected as every time the dispatcher is visited, B_e is visited for sure. While the visit count of another block B_t might be decreased since this block might be visited only if certain conditions are satisfied after B_e. Therefore, with claim 1, we could still detect the entry block B_e of the dispatcher. For the detection of B_t, it will be handled with the following two analysis.

3.3.3 Redundant cycle analysis

Due to compilation, the dispatcher block might become more than one basic block on a simplified CFG. In this case, dispatcher elimination might result in redundant cycles in the flattened CFG.

A redundant cycle is made up of two basic blocks. Given two basic blocks B_{cycle1}, B_{cycle2}, they composite a cycle if there are two directed edges that

1. one goes from B_{cycle1} to B_{cycle2};
2. another goes back from B_{cycle2} to B_{cycle1}. (Figure 3.1)

The relationship between the two basic blocks is similar to the relationship between the boolean test block and the loop body block in a *while* or for loop, where one edge goes from the boolean test block to the loop body block and one edge goes in the reverse direction. However the cycles we talked about here is different from the cycles introduced by *while* and for loops.

For the cycles generated from *while* or *for* loops, the visit count of the boolean test block $v_{B_{bool}}$ and the visit count of the loop body block $v_{B_{body}}$ should satisfy

$$v_{B_{bool}} = v_{B_{body}} + 1^1$$

or

$$v_{B_{bool}} = v_{B_{body}}.$$

The first equation describes the case where the loop is exited normally when the boolean expression is no longer satisfiable, while the second equation is caused by a *break* statement in the loop body.

However the cycles resulting from insufficient dispatcher removal usually do not satisfy the proposed equations. Therefore these redundant cycles could be recognized and removed. Combined with the dispatcher characteristics, the additional dispatcher blocks could be found and then removed.

3.3.4 Redundant edge analysis

Finally, there might be some successive basic blocks that could be merged into a bigger basic block without losing any accuracy on the program control flow.

Given two adjacent basic blocks B_{pre} and B_{next}, they could be merged into one basic block if

1. B_{pre} with $OUTdegree = 1$,
2. B_{next} with $INdegree = 1$. (Figure 3.2)

As a reminder, the $INdegree$ and $OUTdegree$ of a basic block are defined as the counts of directed edges that go in and come out of the basic block respectively.

In a program, it is incorrect to have such basic blocks as program execution does not change between them. These successive basic blocks are observed on the deflated CFG resulting from the elimination of dispatcher and cycle blocks. Therefore, after the elimination, these redundant edges on the CFG should also be removed.

[1] This is the equation for a standalone *while* or *for* loop. For nested *while* or *for* loops, the equation of a inner loop should be changed to $n(v_{B_{bool}}) = n(v_{B_{body}} + 1)$, where n is the loop count of its outer loop. It is not hard to obtain n, as it is the visit count of the basic block prior to the basic blocks of inner loops.

The redundant edge analysis might be carried out repeatedly. After removing the redundant control flow, the final de-obfuscated CFG is constructed.

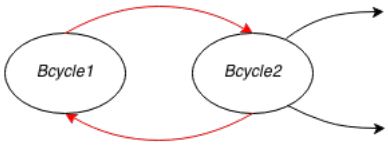


Figure 3.1: A redundunt cycle

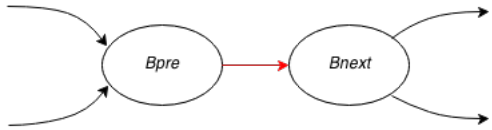


Figure 3.2: A redundant edge

3.4 Discussion

There are some interesting issues related to the basic de-flattening method that need to be expounded further.

3.4.1 Complete coverage and program parameters

When constructing de-flattened CFG, we make use of execution paths that are generated by dynamic analysis. As mentioned in Section 2.5.1, the dynamic program analysis suffers from the code coverage problem. In order to achieve a complete coverage (or n% coverage in case dead blocks exist) of the program statements, the current model repeats the execution of the program with *different parameters* until the dynamic execution paths that result in a complete coverage (or n% coverage) of the program are obtained.

The coverage of a program could be evaluated in terms of basic blocks or edges.

When evaluating with basic blocks, we compute the percentage of basic blocks covered dynamically. The percentage of the basic blocks that have been visited dynamically $p^b_{visited}$ are computable by using

$$p^b_{visited} = \frac{b_{visited}}{b_{total}},$$

where $b_{visited}$ is the count of different basic blocks that have been visited in any execution path and b_{total} refers to the total amount of basic blocks in the flattened program. The total amount of basic blocks b_{total} could be counted by static analysis.

The evaluation of program coverage with edges relies on the percentage of the edges that have been visited in any execution path on a flattened CFG. Therefore, the computation equation is modified accordingly

$$p^{e}_{visited} = \frac{e_{visited}}{e_{total}},$$

where $e_{visited}$ is the count of edges that have been explored dynamically on a flattened CFG and b_{total} is the total amount of edges on the same CFG. By definition, the code coverage in terms of edges suggests a better evaluation for dynamically constructed CFGs, as it takes not only basic blocks but also edges into computation. Thus, $p^{e}_{visited} = 1$ indicates $p^{b}_{visited} = 1$; but not the other way around. However it might be impossible to deduce the total count of edges existing on a flattened CFG by statically analyzing its assembly code, if indirect *jump* destinations are observed in the assembly code.

In the discussion of this chapter, we assume that all representative execution paths could be obtained, i.e., $p^{e}_{visited} = 1$ provided no artificial basic blocks, by "guessing" their corresponding input sets. However our base model could be improved by utilizing static program analysis to compute the parameters rather than to "guess". This improvement made by utilizing static program analysis will be fully discoursed in Chapter 5.

3.4.2 Code after dispatcher elimination

Our de-flattening result is the de-flattened program control flow. The code after dispatcher removal is no longer executable. This is evidenced by the fact that the algorithm removes basic blocks simply without any modification added to the assembly code. However, the de-obfuscation program control flow is enough for many researches on program structure. An example could be a study to classify viruses (might be obfuscated viruses) to their families based on program CFGs.

3.5 Summary

In this chapter, we have introduced the method to reveal a de-obfuscated CFG from an obfuscated program by utilizing dynamic program analysis. This scenario assumes that the inputs to the program that leads to the complete coverage (or n% coverage) of the program is guessable by repeating the program several times. In next chapter, we will present the evaluation of our method against some examples. In Chapter 5, we will refine our method by introducing static program analysis to deduct the correct program inputs that should be selected.

Chapter 4

Evaluation of the base model

This chapter evaluates the de-obfuscation method presented in the previous chapter against some program examples. The examples are given in C. They cover most essential syntax in the C language. The C code is compiled into machine code with *GCC* without any compiler optimization. In a similar way, the code protected with flattening is compiled. The targeted obfuscation scheme is mainly the basic control flow flattening. By default, it is added on the *main* function in each example. The flattening implementation is provided by Technicolor R&D, France [Tec]. In their implementation, besides adding the flattening directly at the source code level, they also apply some modification at the intermediate level. As a result, (1) indirect *jump* instructions are used in the compiled code and (2) there is no difference between the CFG constructed with the base construction algorithm and the CFG constructed with the improved algorithm.

In this chapter, we first present the implementation of the base de-obfuscation model. It clarifies how the experiment is set up and carried out. Then we introduce the examples to be used in the experiment. Following the examples, we present the experiment result and the evaluation of code coverage.

4.1 Implementation

This section discusses the implementation of the base model.

The algorithm of de-obfuscation is implemented in Java. Given all the possible execution paths within a program, the Java application constructs the simplified CFG dynamically. On this CFG, after detecting and removing the dispatcher, the application returns the de-flattened CFG constructed. The returned CFG is saved in a *dot* [Gra] file.

The CFG of the original program is also constructed dynamically. While we could also get this CFG statically by importing the binary file directly into *IDA*. With *IDA* on Windows operating systems, besides a list of disassembled instructions, we could also obtain a graphical view of the disassembled code.

4.2 Examples

There are five examples in total (Table 4.1). They cover most of the essential syntax in the C language.

The example 1 is a simple "hello world" program. It contains no branch or loop. Then the next three examples are C code that is made up of basic statements, namely *if-else, switch-case*, and *while*. Finally the example 5 is a freestyle code which consists of a combination of basic statements.

Basically, when constructing the de-flattened CFG, as long as we could construct the CFG of the flattened program dynamically, there is no constraint on the syntax in examples. While in Chapter 6, we will look into the examples again and discuss more issues about the constraints on the language syntax. The source code of the examples could be found in Appendix A.

4.3 Evaluation of the de-flattening algorithm

This section presents the de-obfuscation result with our base model against the basic control flow flattening method.

Table 4.1 illustrates the experiment result. From the table, we could conclude that when there is only one basic block in an original program, the basic block

count in the flattened program does not increase after flattening; while when there are more than one basic blocks in an original program, at least one dispatcher block will be added into the flattened program. The table also indicates that after de-flattening with the base model, the basic blocks that are added by flattening could be smoothly removed. In addition, each de-flattened program preserves the same structure as its original program.

Consider example 5 as an example. First, we take a closer look at the de-flattening process. Figure 4.1(a) illustrates the CFG of example 5. It is the CFG constructed dynamically with the execution paths obtained in the flattened program. On this CFG, the dispatcher block is found and marked in red. Then after removing the dispatcher block, we get Figure 4.1(b). In this figure, a cycle is detected between BB_orgBB_5 and 261. Therefore, we mark the basic block that is visited more frequently, i.e., 261 in pink. On the same figure, we also find that the directed edges between block pairs 96 and BB_orgBB_11, 377 and BB_orgBB_13, 422 and BB_orgBB_14, and 446 and BB_orgBB_15 are redundant. Each block pair could be merged into one. In other words, one of the basic blocks in each pair could be deleted. The basic blocks to be deleted for this reason are marked in green. Finally, after removing all the colored basic blocks, the de-flattened CFG is shown in Figure 4.2(a).

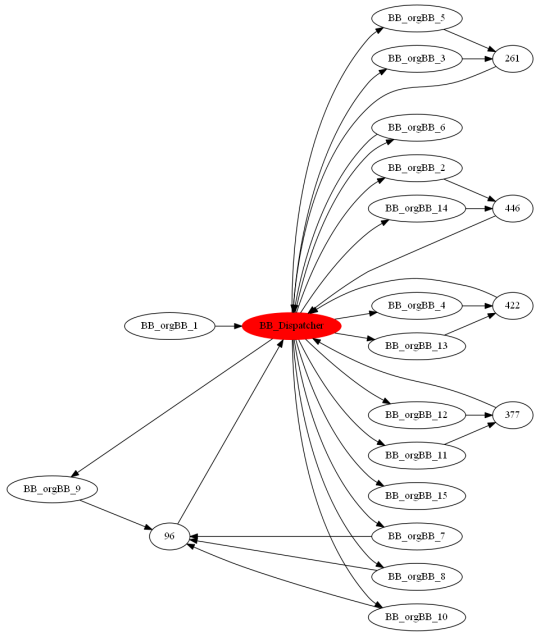
Second, we compare the de-flattened CFG against the original CFG. Figure 4.2(a) illustrates its de-flattened CFG and Figure 4.2(b) shows the CFG which is generated from the original program. The two CFGs are isomorphic by looking it directly. The isomorphism could also be verified by graph isomorphism algorithms given in Section 2.2.3. The same goes for all other examples.

Thus, based on the discussion above, we could conclude that the basic control flow flattening could be invalidated by de-flattening using the base de-flattening model.

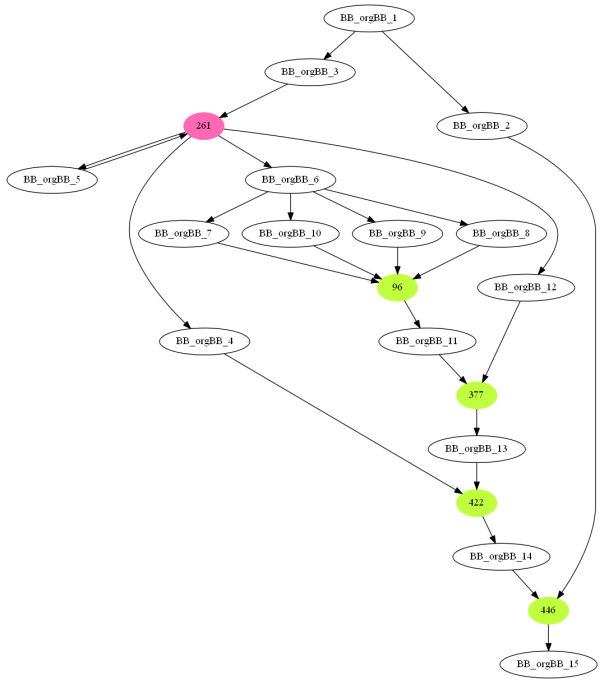
Table 4.1: Examples and evaluation result of the base de-flattening model

No.	Description	c_{CFGori}	c_{CFGfla}	c_{CFGdef}	*Same*
1	print "hello world"	1	1	1	Yes
2	*if-else*	4	6	4	Yes
3	*switch-case*	6	8	6	Yes
4	*while* loop	4	5	4	Yes
5	combination of basic statements	15	21	15	Yes

c_{CFGori}: the count of basic blocks in the original CFG.
c_{CFGfla}: the count of basic blocks in the flattened CFG.
c_{CFGdef}: the count of basic blocks in the de-flattened CFG.
Same: whether the original CFG and the de-flattened CFG are the same.

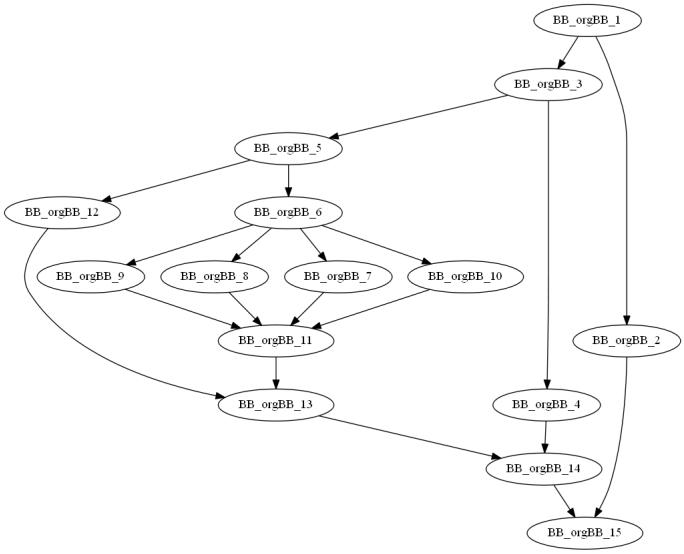


(a) The CFG constracted dynamically

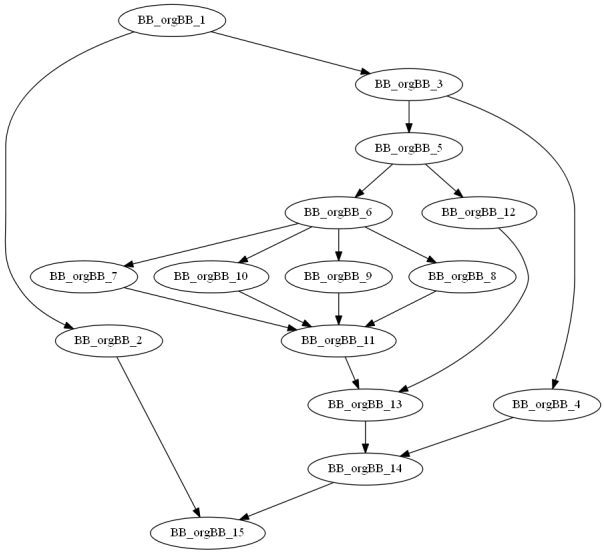


(b) The CFG after dispatcher removed

Figure 4.1: CFGs of example 5 (1). Red nodes: the dispatchers detected. Green nodes: the nodes resulting in redundant edges. Pink nodes: the nodes resulting in cycles.



(a) The de-flattened CFG



(b) The CFG without obfuscation

Figure 4.2: CFGs of example 5 (2)

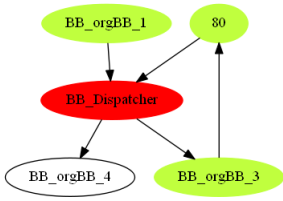
4.4 Code coverage evaluation

The results achieved by de-flattening using dynamic analysis are based on the assumption that all the basic blocks (except for artificial blocks) could be found in one of the execution paths that are obtained by providing different parameters to the program. However what would happen if we failed to cover some of the basic blocks and edges in the execution paths? Our experiment result indicates that it results in some nodes or edges missing in the final de-obfuscated CFG, but has little effect on the dispatcher detection.

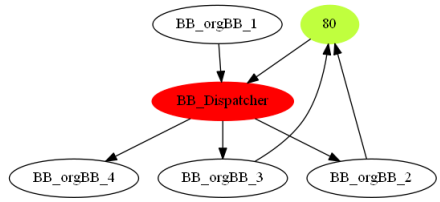
Take the example 2 as an example. The example 2 consists of one *if-else* structure in its main program body. Suppose when running the obfuscated program, the program arguments provided by guessing could only satisfy the boolean condition for the if branch. Then the *else* branch would be never reached in any execution paths. The flattened CFG constructed in this case is shown in Figure 4.3(a). The nodes that are marked in color in the figure are the nodes that will be moved in the de-flattened CFG. In other words, after de-flattening, there is only one node BB_orgBB_4 left in the de-flattened CFG (Figure 4.3(c)). The flattened CFG that covers all the basic blocks in the flattened program is shown in Figure 4.3(b), and its corresponding de-flattened CFG, as well as the original CFG is illustrated in Figure 4.3(d). Based on these four figures, we could draw a conclusion that the dispatcher block could be detected successfully in this example even if some basic blocks are missing in the execution paths for dynamic program analysis. However the missing of basic blocks in the execution paths might result in a lot of nodes uncovered in the de-flattened CFG.

4.5 Summary

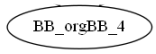
In this chapter, we have presented the evaluation of the basic de-flattening model using dynamic program analysis. The evaluation result indicates that the basic model invalidates the targeted basic control flow flattening smoothly. However we also detected the potential problem with this method that if some of the basic blocks are uncovered in any execution path, part of the CFG might be missing in the final de-obfuscated CFG. In next chapter, we will discuss this problem in depth.



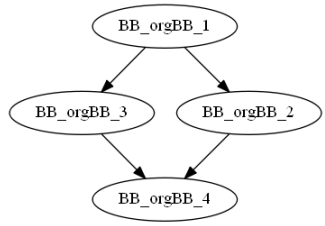
(a) The sub-CFG for *if* branch



(b) The complete CFG



(c) The de-flattened CFG of 4.3(a)



(d) The de-flattened CFG of 4.3(b)

Figure 4.3: CFGs of example 2. Red nodes: the dispatchers detected. Green nodes: the nodes result in redundant edges.

Chapter 5

The improved model

Based on the discussion in previous chapters, we have proved that the de-flattening using dynamic program analysis could be used to reveal the original program control flow. However it is also observed that the de-flattening method using dynamic program analysis suffers from the complete code coverage problem. In this chapter, we provide the solutions to this problem by introducing static program analysis into the basic model.

The idea is that in order to obtain the dynamic execution paths that cover all the basic blocks in the code, right inputs should be given to the program. These inputs could be computed with the assistance of dynamical symbolic execution which is a analysis technique that belongs to static program analysis. In this chapter, we focus on the use of dynamical symbolic execution to obtain representative inputs. Our discussion starts with the path feasibility analysis and ends with the algorithm to compute all the inputs for all possible execution paths.

5.1 Path feasibility analysis

Given an execution path, this section discusses the computation of inputs that contribute to a specified path. As we have seen in Section 2.5.2, dynamic symbolic execution computes the inputs to an execution path by solving the mathematical formulas that are derived during the exploration of the program. The mathematical formula computation is done with a SMT solver. In practice, we could use *Z3* as the SMT solver. Then the problem left is the derivation of the mathematical formulas. A SMT solver takes formulas as inputs, while our execution paths are written in assembly language. Therefore, the first step of the execution path computation is to construct a path formula from a piece of assembly code. This conversion is carried out by finding the mapping between assembly instructions and logical formulas.

In the following discussion, we expound the construction of the mapping step by step. In the end, a concrete example could be found for the conversion from an execution path to a path formula.

5.1.1 The mapping from assembly language to logical formulas

The mapping f from the assembly instruction set $\mathcal{I}$ to the logical formula set $\mathcal{F}$

$$f : \mathcal{I} \to \mathcal{F}$$

is given in Table 5.1. In the table, the first column lists all the instructions in the instruction set $\mathcal{I}$. Obviously, this instruction set is a subset of the whole assembly instruction set. The second column shows all the logical formulas that correspond to the instructions on their left. These formulas come from the logical formula set $\mathcal{F}$. Finally the third column is the variable set $\mathcal{V}$, which is made up of the variables used in each instruction respectively.

In this section, we elaborate the symbols and the functions used in the table.

Label l

In Table 5.1, each instruction is given with a label l which describes the virtual address of the instruction. The virtual address of $(i+1)$th instruction is given by $l_{i+1} = l_i + |l_i|$, where $|l_i|$ is the size of ith instruction. This address is useful in *jump* instructions where it specifies the destination that the control flow wants to change to.

The set symbols with labels suggest specific elements that belong to the set. For example, $\mathcal{I}^l$ indicates a specific instruction in the whole assemble instruction set. The same for $\mathcal{F}^l$ (a specific logical formula) and $\mathcal{V}^l$ (a specific variable set).

Variable set $\mathcal{V}$

In a path formula, the variable set $\mathcal{V}$ keeps track of all the variables and their indices used in the formula. As already seen in the example given in Section 2.5.2, the variables used in the path formula are indexed. The reason we use indexed variables is that we want to differentiate a variable in the different sides of an assignment so that we can express assignment statement in logical formulas. Consider the following example. It is quite common to have an assignment in C

$$i = i + 1;$$

However in mathematics this equation is totally wrong as i could only equals to i. Thus to express this code in a logical formula, it should be rewritten as

$$i1 = i0 + 1.$$

Therefore, it is necessary to have indexed variables.

Function $newIndVar(a)$ and function $indVar(a)$

Finally, $newIndVar(a)$ and $indVar(a)$ are two functions and a is a function parameter. The value of a could be a register, a memory location or a constant number. If a is a register or a memory location, the former function $newIndVar(a)$ returns the variable name followed by a new index; whereas the latter one $indVar(a)$ returns the variable name followed by the recently used index. If a is a constant number, its value is returned directly for both of th two functions. Therefore, the C code

$$i = i + 1;$$

could be rewritten as

$$(newIndVar(i) = indVar(i) + 1).$$

5.1.2 An example

After we have introduced all the symbols in the table, this subsection explains how the conversion from assembly code to a path formula is carried out with a concrete example.

Given a piece of assembly code shown in List 5.1, we first reason the path formula of the code by hand. The meaning of code suggests that if EAX equals to $0x1045101$, then the program jump to the address $0x8048480 < main + 144 >$. The address after the JE instruction indicates that the $jump$ operation is executed. Hereby the logic formula derived should be

$$indVar(EAX) = 0x1045101.$$

Then we derive the logical formula with the mapping defined in Table 5.1. First, for the CMP instruction, $op1$ is a register EAX and $op2$ is a constant number $0x1045101$. Therefore, the variables used in this formula are $op1 = indVar(EAX)$ and $op2 = 0x1045101$.

Second, the labels of the CMP and Jcc instructions are $l_0 = 0x804845b < main + 107 >$ and $l_1 = 0x8048460 < main + 112 >$ respectively. The address that the $jump$ instruction intends to go to is $l_k = 0x8048480 < main + 144 >$. which happens to be the address that directly comes after the $jump$ instruction $l_2 = l_k = 0x8048480 < main + 144 >$.

Third, for a JE instruction, the corresponding operation symbols are $=$ for Jcc^+ and $! =$ for Jcc^-.

Therefore, the transfer function leads us to

$$[(indVar(EAX) = 0x1045101) \wedge (l_k = l_2)] \vee [(indVar(EAX) ! = 0x1045101) \wedge (l_k ! = l_2)].$$

Which could be simplified to

$$indVar(EAX) = 0x1045101$$

as $l_k = l_2$.

This result indicates how a path formula is generated with Table 5.1. In the end, by providing the path formula and the variable set to $Z3$, the inputs to the specified path could be computed. For this example, the input should satisfy $EAX = 0x1045101$.

Listing 5.1: A section of assembly code

```
0x804845b <main+107>:   CMP     EAX, 0x1045101
0x8048460 <main+112>:   JE      0x8048480 <main+144>
0x8048480 <main+144>:   ...
```

Table 5.1: The mapping $f : \mathcal{I} \rightarrow \mathcal{F}$

$\mathcal{I}^l$	$\mathcal{F}^l$	$\mathcal{V}^l$
$[MOV\ src, dst]^l$	$newIndVar(dst)$ $= indVar(src)$	$indVar(dst)$ $\bigcup indVar(src)$ $\bigcup newIndVar(dst)$
$[PUSH\ src]^l$	$\emptyset$	$\emptyset$
$[POP\ dst]^l$	$\emptyset$	$\emptyset$
$[ADD\ src, dst]^l$	$newIndVar(dst) =$ $indVar(dst) + indVar(src)$	$indVar(dst)$ $\bigcup indVar(src)$ $\bigcup newIndVar(dst)$
$[SUB\ src, dst]^l$	$newIndVar(dst) =$ $indVar(dst) - indVar(src)$	$indVar(dst)$ $\bigcup indVar(src)$ $\bigcup newIndVar(dst)$
$[IMUL\ src, dst]^l$	$newIndVar(dst) =$ $indVar(src) * indVar(dst)$	$indVar(src)$ $\bigcup newIndVar(dst)$
$[OR\ src, dst]^l$	$newIndVar(dst) =$ $indVar(dst) \mid indVar(src)$	$indVar(dst)$ $\bigcup indVar(src)$ $\bigcup newIndVar(dst)$
$[XOR\ src, dst]^l$	$newIndVar(dst) =$ $indVar(dst)\hat{}indVar(src)$	$indVar(dst)$ $\bigcup indVar(src)$ $\bigcup newIndVar(dst)$
$[NOP]^l$	$\emptyset$	$\emptyset$
$[CMP\ op1, op2]^{l_0}$ $[Jcc\ l_k]^{l_1}$ $[Inst]^{l_2}$	$[(indVar(op1)\ Jcc^+\ indVar(op2))$ $\wedge (l_k = l_2)] \vee$ $[(indVar(op1)\ Jcc^-\ indVar(op2))$ $\wedge (l_k != l_2)]$	$indVar(op1)$ $\bigcup indVar(op2)$
$[CMP\ op1, op2]^{l_0}$ $[Scc\ dst]^{l_1}$ $[Inst]^{l_2}$	$[(newIndVar(dst) = 1) \wedge$ $(indVar(op1)\ Scc^+\ indVar(op2))]$ $\vee [(newIndVar(dst) = 0) \wedge$ $(indVar(op1)\ Scc^-\ indVar(op2))]$	$indVar(op1)$ $\bigcup indVar(op2)$ $\bigcup newIndVar(dst)$
$[CALL\ proc]^l$	$\emptyset$	$\emptyset$
$[JMP\ dst]^{l_0}$ $[Inst]^{l_1}$	$dst = l_1$	$indVar(dst)$
$[RET]^l$	$\emptyset$	$\emptyset$

Table 5.2: Jcc, Jcc^+ and Jcc^-

Jcc	Jcc^+	Jcc^-
JE	$=$	$!=$
JNE	$!=$	$=$
JZ	$=$	$!=$
JNZ	$!=$	$=$
JG	$>$	$<=$

Table 5.3: $Scc, Scc^+$and Scc^-

Scc	Scc^+	Scc^-
$SETG$	$>$	$<=$
$SETL$	$<$	$>=$
$SETE$	$=$	$!=$
$SETGE$	$>=$	$<$

5.2 Program execution path computation

Till this point, we are able to symbolically execute a path and compute the arguments required for entering that path. Then the next step is to design an algorithm to obtain possible execution paths along with the input sets from the whole assembly code. In this section, we first present the overall algorithm to compute the execution paths. It is followed by a detailed explanation of each algorithm step. Finally we propose a simple discussion on the correctness of the algorithm. As every assembly instruction in a program counts in the computation, the CFG used in this section is the CFG built with the basic construction method presented in Section 2.2.2.

5.2.1 The algorithm of execution path computation

First, the algorithm to compute all the inputs and their corresponding execution paths is given as follows:

Data structures:

1. A graph G where all basic blocks and the connection relationship between the basic blocks are stored.
2. A list l which records input sets to be verified.
3. An array P which keeps track of all execution paths obtained.

Algorithm:

1. Construct G statically. Initiate l by putting l_0 = the default function inputs (0s to all the arguments) to the list. Initiate P to be empty.
2. While the list l is not empty, then repeat:
   (a) Take one element l_i from l and execute the function with inputs l_i to get one execution path $P_i = bb_1, bb_2, ..., bb_n$.
   (b) If P_i is already in P, abandon this path.
   (c) Else, add P_i to P and connect the directed edges among the basic blocks on G.
   (d) Then for each basic block bb_j which ends with a *non-deterministic jump* instruction on the execution path P_i,

i. For each basic block bb_k that has no directed edge from bb_j to bb_k on G,
   A. Compute the satisfiable of the new execution path $bb_1, bb_2, ..., bb_j, bb_k$,
   B. If the new execution path is satisfiable, put the computed inputs in the list l.

5.2.2 Algorithm explanation

This subsection looks into the algorithm given above in depth.

First of all, we use a graph G to keep record of all the explored nodes and edges within the program. This CFG is first constructed statically. It is made up of all non-NOP constructions in the assembly code. Then in the course of the algorithm, the edges obtained in execution paths are added into the CFG. The graph obtained when the algorithm ends is a CFG constructed dynamically. This CFG is suitable for dispatcher detection and elimination.

Second, the algorithm uses a list l, which could either be a queue or a stack, to keep track of all the interested inputs. At the beginning of the algorithm (Step 1), the list l is initiated with the default input set, i.e., 0s to all the program arguments. Then in the *while* loop (Step 2), inputs computed from satisfiable execution paths are added into the list one by one.

Third, in order to avoid computing the same execution path again and again, the algorithm uses an array P to record all the execution paths that have already be explored. This array P is initiated as an empty array at Step 1. Then in the *while* loop in Step 2, every time we obtain a new execution path P_i with the input set l_i, we first verify whether we have already computed this path before. If it has been computed before, we abandon P_i and continue the *while* loop by trying another input set. If P_i has not been computed before, we then record the new path to P and go to Step 2d.

Forth, the main computation of path satisfiability and inputs is carried out in the *while* loop in Step 2. The stop condition of the loop is that the list l become empty. This stop condition indicates that all the possible inputs computed have been tried.

Fifth, Step 2a describes the generation of the execution path P_i dynamically with a set of inputs l_i. This procedure is the same computation procedure with *GDB* discussed in Section 2.5.1.

Sixth, Step 2b to Step 2c determine whether to process the execution path P_i

or not. As we have discussed, if P_i is a path that has already be recorded, the algorithm will go back to Step 2 and try another input set. While if it is a new execution path, then the algorithm registers it in P and processes to Step 2a.

Finally, in Step 2d, on the execution path P_i, we detect every basic block that includes a non-deterministic *jump* instruction, such as a conditional *jump* and an indirect *jump*. These basic blocks indicate branch points on the execution path P_i. In the current call, the execution takes one path at each branch point, but it is possible that it takes other paths, which results in different execution paths. Therefore at each branch point b_j, we replace the next basic block with bb_k which is one of the basic blocks that have never been visited from block bb_j in all execution paths. In other word, when selecting the next possible basic block to construct a new execution path, we try all the basic blocks that have never been reached from the current basic block on G. However, this execution path might be not feasible at all. The validity of this path will be verified with the path feasibility analysis presented in the previous section. Finally, if there is a set of inputs to make this path satisfiable, we put the inputs into the input list l.

5.2.3 Algorithm evaluation

In this subsection, we take a look at the correctness of the algorithm in terms of completion, efficiency and robustness.

Completion This algorithm ensures all the edges on the program CFG will be explored. The algorithm aims to compute possible execution paths that are then used in the construction of program CFG for the dispatcher elimination. For this purpose, we should ensure no node or edge is missing in the CFG. Therefore, when computing a new execution path in Step 2d, for each branch point, the algorithm explores all the nodes that have not been connected from the branch node.

Efficiency The efficiency of the algorithm is guaranteed in two aspects. First, this algorithm explores an edge once and only once. It makes sure that not only all the edges on the program CFG will be explored, but also the edges would be explored efficiently. As it is already said above, in the course of new execution path construction in Step 2d, the model only select a node from the nodes that have no connection with the branch node before, which speeds up the computation efficiently and ensures that the algorithm will not be trapped in a loop in the program permanently. Second, the algorithm will not repeat any paths that have already been computed. It utilizes an array P to keep track of the execution paths. By doing this, we make sure that the algorithm will not

lock itself in a path permanently. For example, while computing possible feasible paths on a path P_a, the path feasibility analysis indicates that another path P_b could be reached with inputs I_b. However after providing I_b to the program, the path P_a instead of P_b is executed again. Then in this case, without P, the algorithm will not terminate.

Robustness The algorithm is robust to artificial blocks. The edges to artificial blocks will not be included as the path formula to artificial blocks would probably not be satisfied in the first place. In addition, in the algorithm, all the execution paths are still obtained dynamically. The dead blocks could not appear in any real execution paths recorded in P.

5.3 Summary

This chapter focuses on two enhancements to the base model, i.e., the computation of input sets and that of execution paths. The two enhancements assist in constructing CFG dynamically. With the assistance of the techniques proposed in this chapter, the base model is improved to guarantee that flattened CFGs are constructed based on ample execution paths.

In this chapter, we have suggested an algorithm to compute the execution paths and their corresponding inputs. This algorithm has aimed at investigating not only all the basic blocks but also all the edges in a program. This algorithm relies highly on the path feasibility analysis. The analysis is performed by solving mathematical formulas derived from assembly instructions. An section is included specially for the construction of mathematical formulas from assembly instructions.

After this chapter, we have finished the discussion of constructing a de-obfuscation model to remove obfuscation on control flow from obfuscated programs. In next chapter, we will verify the improved model by evaluating the enhancements added to the base model.

Chapter 6

Evaluation of the enhancements to the base model

After the discussion of the enhancements to the base model, this chapter evaluates these enhancements against examples. Prior to the evaluation, we first present the implementation of the improvements to the base model and discuss several constraints applied on examples. Then we discuss the evaluation of the improvements. The evaluation is primarily carried out in two steps. Firstly, we evaluate the path satisfiability analysis method presented in Section 5.1. The correctness of this method is the base for exploring execution paths dynamically. Secondly, we concentrate on the evaluation of the execution path exploration algorithm given in Section 5.2. This algorithm enables the base model to function properly.

6.1 Implementation

This section gives details about the implementation of the algorithms presented in last chapter.

6.1.1 Implementation of path satisfiability analysis

In the implementation, we adopt $Z3$ as the SMT solver to help to compute the satisfiability of a path. To be specific, we use *Z3py* [Z3p] which is the Python implementation of $Z3$. In the computation, we first write the path formula computed from a piece of assemble code into a Python file, and then call the Python program to return the computed result.

Data type

While putting the path formula into *Z3py* format, we need first declare all the variables we will use in the formula and then write the formula down for computation. In the implementation, we declare all the variables in the variable set $\mathcal{V}$ as *BitVec*, which is a data type used by $Z3$ to represent a binary variable. The benefits of use this data type lie in two aspects.

First, the size of the variable is taken into consideration during computation. When declaring a variable as *BitVec*, we also need to specify the size of the variable. As we've known, in assembly language, the size of each operand is very important. Several special instructions only apply on operands with fixed sizes.

Second, binary operations such as AND, OR, and XOR are able to apply on the variables. Currently $Z3$ only supports the binary operations to apply on *BitVec*. There is no other way to perform this computation with other data types, provided that no type conversion between *BitVec* and any other data types is allowed in $Z3$. The common mathematical operations such as ADD and MUL are not affected with this data type. Both signed and unsigned mathematical operations are supported. Finally, the return format of the computed *BitVec* result could be chosen from a hex number, a signed integer number and an unsigned integer number.

Therefore, based on the two significant benefits given, we choose *BitVec* as the data type of each variable in the path formula. While the use of this data type enforces no floating number in the program. More information on the *Z3py* data type and supported computations could be found in the *Z3py* API [Z3p].

Mapping f

Following the variable declaration, we write the path formula obtained with the mapping f into the Python file. In the construction of the mapping f from assemble code to a path formula, we have implemented all the assembly instructions listed in Table 5.1. Variants of the instruction in the table are also

comprehensible. For example, for the $IMUL$ instruction, except *IMUL src, dst*, it is also acceptable to have *IMUL i, src, dst*, where i is an instant number. As this is a simple variant to the basic instruction, we do not put this in the table. Consider another example for the NOP instruction. The instruction *lea 0x0(%esi),%esi* is also acceptable as it is in fact a NOP instruction.

Other instructions that are not listed in Table 5.1, such as *divide, logic shifts*, are not considered instructions in our current implementation.

Inputs

After we input the path formula correctly, the SMT solver $Z3$ computes the satisfiability of the path. If it is satisfiable, one solution - a value for each declared variable will be returned. Among all the variables, we need to figure out the answers for the inputs. In our current implementation, it is the user who should specify the variable names for inputs.

The variable names for the inputs in fact is simple to deduce. As we have discussed, the default flattened function is the *main* function in each program. Therefore we need to figure out the inputs for this function. When carrying out a function call in assembly language, the inputs to the function, i.e., the function parameters, are organized in a stack. Figure 6.1 illustrates parts of the stack. From this figure, we could find where the function parameters are stored. As only the *main* function is considered, the inputs to the program could only be stored in $0x8(\%ebp)$ and $0xC(\%ebp)$ - one for $argc$, another for $argv$. Therefore the variables for these two inputs in set $\mathcal{V}$ should probably be the first indexed variables of the two memory position.

However when a function parameter is given in a pointer or an array, our implementation needs to know the memory address where the real contents are stored. For example, for the *main* function, the second parameter $argv$ is defined as *char***. Therefore in the stack, the position $0xC(\%ebp)$ stores the memory address where the array of each function argument $char*$ starts. The computation result of the memory address of $argv$ is not so appealing. The real results that we want are the arguments we give when running the program. Therefore our implementation requires the user to specify where each *char* in each argument is stored. When detecting the memory address of each element in an array, tools such as *IDA* and *GDB* could be utilized.

To simplify the program argument providing procedure, we could require that the examples only employ the first argument $argc$, i.e., the count of arguments given to the program, to decide next basic block to execute.

$0x10(\%ebp)$	- Third function parameter
$0xc(\%ebp)$	- Second function parameter
$0x8(\%ebp)$	- First function parameter
$0x4(\%ebp)$	- Old %eip (the function's "return address")
$0(\%ebp)$	- Old %eip (previous function's base pointer)
$-0x4(\%ebp)$	- First local variable
$-0x8(\%ebp)$	- Second local variable
$-0xc(\%ebp)$	- Third local variable

Figure 6.1: The stack structure for a function invocation

6.1.2 Implementation of the execution path computation algorithm

The algorithm to compute all program execution paths is implemented as it is given in Section 5.2. Given a flattened program and its disassembled code, the algorithm computes all the possible execution paths within the program. The list l for input sets is implemented as a queue.

6.2 Examples

Prior to the evaluation, we address the examples constructed for evaluation again. Basically, we choose the same example set used in Chapter 4. However, the C code, as well as the assembly code generated, is required to satisfy several requirements.

6.2.1 Constraints on the language syntax of the examples

As we have talked in last section, the requirements are added due to the path satisfiability analysis. The implementation of the analysis leads to the constraints on the language syntaxes of the examples. These constraints are summarized as follows:

1. No floating number or floating computation is allowed in the C code.
2. No pointer operation is supported in the C code.

3. Merely the assembly instructions listed in Table 5.1 are acceptable in the compiled assembly code.
4. Most of the examples only use the first program argument $argc$ to decide next basic block to execute.

Even though we place these constraints on the examples, the constraints could be removed easily by extending our current implementation. For example by adding the support for floating numbers, and by expending the mapping table with more instructions. For the Constraint 4, it is added just to simplify the argument giving process. Our examples have already designed to satisfy the requirements given above. Besides, the current implementation also supports the simple argument giving process using $argv[1][0]$ in example 5.

6.2.2 Constraints on the size of the examples

There is no specific constraint on the maximum size of the example programs. As long as a program satisfies the supported language syntaxes, our path satisfiability analysis and path exploration computation algorithm should be applicable. However, we have to admit that when the size of a program grows, the consumption time of the two algorithms might increase dramatically.

The computation time of the path feasibility analysis mainly relies on the implementation of the SMT solver $Z3$. However the computation time could be shortened if more knowledge of the program is acquired. For example, when we already know that an input should definitely be a number greater than 0 and less than 10, then the computation could be speed up if we write the knowledge as an supplement to the path formula generated directly by the path feasibility analysis.

The computation time of the path exploration algorithm is affected by three factors: the count of basic blocks in a program c_b, the count of execution paths c_p, and the count of branch points in the ith execution path $c_{bp}^{p_i}$. The upper boundary of the computation time of the path exploration algorithm could then be given by

$$O(\sum_{i=1}^{c_p} c_{bp}^{p_i} * c_b).$$

To sum up, even though there is no strict size limitation on the examples, however when the program become bigger enough, the computation time might not be acceptable any more. In this report, we will not elaborate what is the

maximum size of a program that our algorithms could deal with. Instead, we focus on the proof of the basic syntaxes that our algorithms could support. Thus even though the examples are simple and small, however, as discussed in Chapter 4, they cover most of the important C syntax. Our experiment is suitable to simulate the situation where a key function in a large program is obfuscated.

6.3 Evaluation of path satisfiability analysis

This section evaluates the path satisfiability computation method.

The experiment result indicates that all the possible execution paths in the examples are satisfiable with our path feasibility computation algorithm. The inputs to them result in the exact paths from which the inputs are computed. Currently, with our examples, there is no false positive error or false negative error.

For example, taking the example 2 as an example. In the discussion of code coverage in Section 4.4, we have discussed the situation where the *else* branch is missing in the execution path. Given the code that leads the execution to the *else* branch (Listing 6.1), thereby, the full path formula is generated with the mapping from assembly instructions to a path formula (Listing 6.2). This code is then sent to $Z3$ for computation. Finally the computation result after processing is shown in Listing 6.3. By entering the inputs given in Listing 6.3, we finally obtain the same execution path that leads to the *else* branch.

Listing 6.1: The code to enter *else* branch of example 2

```
0x080483f0 <+0>:        push   %ebp
0x080483f1 <+1>:        mov    %esp,%ebp
0x080483f3 <+3>:        sub    $0x18,%esp
0x080483f6 <+6>:        movl   $0x804a020,(%esp)
0x080483fd <+13>:       call   0x8048300 <printf@plt>
0x08048402 <+18>:       movl   $0x0,-0x4(%ebp)
0x08048409 <+25>:       mov    0x8(%ebp),%eax
0x0804840c <+28>:       mov    %eax,-0x8(%ebp)
0x0804840f <+31>:       mov    0xc(%ebp),%eax
0x08048412 <+34>:       mov    %eax,-0xc(%ebp)
0x08048415 <+37>:       cmpl   $0x1,-0x8(%ebp)
0x08048419 <+41>:       setg   %al
0x0804841c <+44>:       movzbl %al,%eax
0x0804841f <+47>:       mov    $0x804845b,%ecx
0x08048424 <+52>:       imul   %eax,%ecx
0x08048427 <+55>:       mov    $0x1,%edx
0x0804842c <+60>:       sub    %eax,%edx
```

```
0x0804842e <+62>:       mov    $0x8048480,%eax
0x08048433 <+67>:       imul   %edx,%eax
0x08048436 <+70>:       add    %ecx,%eax
0x08048438 <+72>:       mov    %eax,-0x10(%ebp)
0x0804843b <+75>:       jmp    0x804844c <main+92>
0x0804844c <+92>:       movl   $0x804a0a0,(%esp)
0x08048453 <+99>:       call   0x8048300 <printf@plt>
0x08048458 <+104>:      jmp    *-0x10(%ebp)
0x0804845b <+107>:      movl   $0x804a040,(%esp)
0x08048462 <+114>:      call   0x8048300 <printf@plt>
0x08048467 <+119>:      mov    -0xc(%ebp),%eax
0x0804846a <+122>:      mov    0x4(%eax),%eax
0x0804846d <+125>:      mov    %eax,0x4(%esp)
0x08048471 <+129>:      movl   $0x8048580,(%esp)
0x08048478 <+136>:      jmp    0x8048440 <main+80>
```

Listing 6.2: The path formular to enter *else* branch of example 2

```
#variable declaration

EBP1 = BitVec('EBP1', 32)
vEBP1n81 = BitVec('vEBP1n81', 32)
vEBP1p81 = BitVec('vEBP1p81', 32)
EDX1 = BitVec('EDX1', 32)
EDX2 = BitVec('EDX2', 32)
vEBP1n41 = BitVec('vEBP1n41', 32)
EAX1 = BitVec('EAX1', 32)
EAX2 = BitVec('EAX2', 32)
EAX3 = BitVec('EAX3', 32)
EAX4 = BitVec('EAX4', 32)
EAX5 = BitVec('EAX5', 32)
EAX6 = BitVec('EAX6', 32)
EAX7 = BitVec('EAX7', 32)
EAX8 = BitVec('EAX8', 32)
ECX1 = BitVec('ECX1', 32)
ECX2 = BitVec('ECX2', 32)
vEBP1nc1 = BitVec('vEBP1nc1', 32)
vEBP1pc1 = BitVec('vEBP1pc1', 32)
vEAX7p41 = BitVec('vEAX7p41', 32)
vESP2p01 = BitVec('vESP2p01', 32)
vESP2p02 = BitVec('vESP2p02', 32)
vESP2p03 = BitVec('vESP2p03', 32)
vESP2p04 = BitVec('vESP2p04', 32)
vEBP1n101 = BitVec('vEBP1n101', 32)
vESP2p41 = BitVec('vESP2p41', 32)
AL1 = BitVec('AL1', 8)
ESP1 = BitVec('ESP1', 32)
ESP2 = BitVec('ESP2', 32)


#build the formula path

s = Solver()
s.add( EBP1 == ESP1,    #<+1>:  mov    %esp,%ebp
```

```
ESP2 == ESP1 - 24,       #<+3>:  sub    $0x18,%esp
vESP2p01 == 134520864,   #<+6>:  movl   $0x804a020,(%esp)
vEBP1n41 == 0,           #<+18>: movl   $0x0,-0x4(%ebp)
EAX1 == vEBP1p81,        #<+25>: mov    0x8(%ebp),%eax
vEBP1n81 == EAX1,        #<+28>: mov    %eax,-0x8(%ebp)
EAX2 == vEBP1pc1,        #<+31>: mov    0xc(%ebp),%eax
vEBP1nc1 == EAX2,        #<+34>: mov    %eax,-0xc(%ebp)
Or(And(vEBP1n81>1, AL1==1), And(vEBP1n81<=1, AL1==0)),
#<+37>: cmpl   $0x1,-0x8(%ebp)
#<+41>: setg   %al
EAX3 == ZeroExt(24, AL1),#<+44>:movzbl %al,%eax
ECX1 == 134513755,       #<+47>: mov    $0x804845b,%ecx
ECX2 == ECX1 * EAX3,     #<+52>: imul   %eax,%ecx
EDX1 == 1,               #<+55>: mov    $0x1,%edx
EDX2 == EDX1 - EAX3,     #<+60>: sub    %eax,%edx
EAX4 == 134513792,       #<+62>: mov    $0x8048480,%eax
EAX5 == EAX4 * EDX2,     #<+67>: imul   %edx,%eax
EAX6 == EAX5 + ECX2,     #<+70>: add    %ecx,%eax
vEBP1n101 == EAX6,       #<+72>: mov    %eax,-0x10(%ebp)
vESP2p02 == 134520992,   #<+92>: movl   $0x804a0a0,(%esp)
vEBP1n101 == 134513755, #<+104>:jmp    *-0x10(%ebp)
vESP2p03 == 134520896,   #<+107>:movl   $0x804a040,(%esp)
EAX7 == vEBP1nc1,        #<+119>:mov    -0xc(%ebp),%eax
EAX8 == vEAX7p41,        #<+122>:mov    0x4(%eax),%eax
vESP2p41 == EAX8,        #<+125>:mov    %eax,0x4(%esp)
vESP2p04 == 134514048)   #<+129>:movl   $0x8048580,(%esp)
print(s.check())
print(s.model())
```

Listing 6.3: The computation result

```
Sat
[vEBP1p81 = 8]
```

6.4 Evaluation of the execution path computation

After we have proved that the execution paths and their corresponding input sets are computable with the path feasibility analysis, this section focuses on the coverage percent of basic blocks and edges in flattened CFGs with the path exploration algorithm.

Table 6.1 illustrates our experiment result. From the table, we find that with the execution path computation algorithm, all the basic blocks in the program are fully explored. Besides, all the edges on the flattened CFGs are also visited.

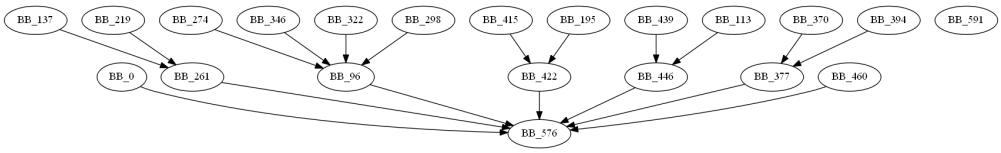
This is evidenced by the fact that all the execution paths that are needed to construct the complete flattened CFGs are successfully computed.

Table 6.1: Examples and the evaluation result of the execution path computation

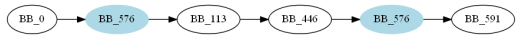
No.	c_{BB}	c_P	c'_{BB}	c'_P
1	1	1	1	1
2	6	2	6	2
3	8	4	8	4
4	5	1 or 2	5	2
5	21	7	21	7

c_{BB}: the count of basic blocks in the flattened CFG.
c_P: the minimum count of paths to construct the flattened CFG.
c'_{BB}: the count of visited basic blocks with the path exploration algorithm.
c'_P: the count of paths computed with the path exploration algorithm.

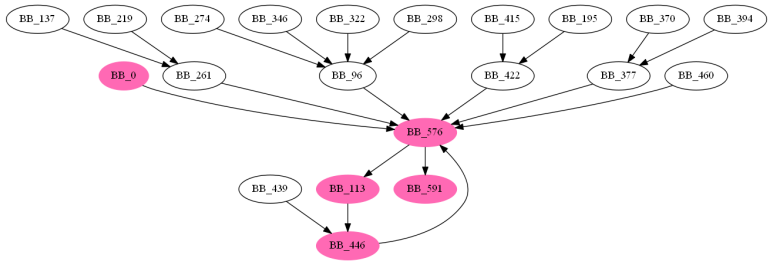
Take the example 5 as an example, Figure 6.2 presents the first few steps in our computation. In the initialization step, Figure 6.2(a) shows all the basic blocks and the edges. This figure is deduced by statically examining the whole assemble code. Then we start our algorithm by adding the default inputs to the input list l. Thus the first execution path with the default inputs is obtained and shown in Figure 6.2(b), where the branch points are marked in blue. After this execution path, the program CFG is updated in Figure 6.2(c). Then our algorithm predicts that the next execution path is a path passing basic blocks BB_0, BB_575, BB_137, where BB_137 is the basic block selected from all basic blocks that are not connected to the first branch point in Figure 6.2(c). Then the algorithm verifies the satisfiability of this path and a Sat is returned with a set of inputs. Therefore the inputs are added on l. Then the algorithm keeps on replacing the basic block after the first branch point with other possible basic blocks (all the basic blocks except for BB_576, BB_113 and BB_591 in Figure 6.2(c)) and compute the satisfiability of the newly constructed paths. The same goes for the second branch point. After exploring all the possible paths from the path shown in Figure 6.2(b), the algorithm takes the next inputs from l and repeats the same procedure for the default inputs. Finally the complete CFG is given in Figure 6.2(d). If we use the same annotation for the CFG in Figure 4.1(a), the same CFG in Figure 6.2(d) will be returned.



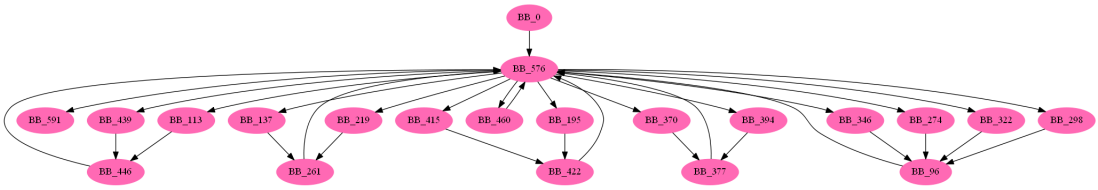
(a) Initialization - all the basic blocks



(b) The execution path generated with default inputs



(c) The CFG after the execution path in 6.2(b)



(d) The final CFG after the path exploration

Figure 6.2: Path exploration. Pink nodes: the basic blocks that have been visited. Blue nodes: the branch points.

6.5 Summary

In this chapter we have discussed the evaluation of algorithms for path satisfiability analysis and path exploration computation. Our experiment result has indicated that it is sufficient to use our algorithms to generate the execution paths on which the base model is based. Therefore, it is proved that our deflattening model is effective in removing the basic control flow flatting scheme.

Chapter 7

Conclusions and future research

This chapter concludes all our work and proposes several possible future work directions.

7.1 General conclusions

Obfuscation is a technique to protect software internals from being examined. This technique could not only be used to protect intellectual property of software owners, but also be utilized by viruses to avoid being detected by virus scanners. The unguided use of this technique leads to the research of de-obfuscation, which is the reverse transformation of obfuscation, aiming to eliminate the obfuscation added in software.

In this report, focusing on flattening which is a typical obfuscation approach on the program control flow, we have proposed a de-flattening model based on the CFGs of flattened programs. With flattening, a dispatcher block is introduced into the CFG of a program. After the insertion of the dispatcher, the program control flow is always directed back to the dispatcher whenever the execution of a basic block finishes. Based on these characteristics of the dispatcher, we have

deduced that a dispatcher is the basic block that is visited most frequently in a flattened program. Therefore it is not hard to find the dispatcher in a flattened program. Then the key problem lies in how to remove this dispatcher in the program and reconstruct its original control flow.

The de-flattening discussion starts with a base model where we primarily address the operations to remove the dispatcher and to reveal the original CFG. The base model makes use of dynamic execution paths. The idea is that by eliminating the dispatcher block, the real control flow of a program can be uncovered, thereby the de-flattened CFG can be constructed.

However, the base model suffers from code coverage problem due to dynamic analysis. To make the base model work, we have to ensure that all the execution paths that result in a full coverage of code are obtained. Therefore, the base de-flattening model is enhanced with dynamic symbolic execution analysis which is a static analysis method to compute all important execution paths automatically.

We have implemented the improved de-flattened model and evaluated it against several representative examples. The evaluation has been carried out in two steps. First we evaluate that when all important execution paths are provided, the base model is effective in re-constructing the original program CFG from flattened programs. Then we verify that the algorithm to compute all important execution paths results in the execution paths that cover all basic blocks and edges in the program. Combined with the two evaluation result, we note that the de-flattening model is successful in revealing flattened program structure.

The experiment is carried out on binaries. The compilation from source code to machine code adds difficulties to extract useful information from low level code. While our research overcomes this problem smoothly and provides new views for de-obfuscation on low level code. Besides, as the CFGs of programs are the base of many studies on program structure, the research result could be employed in many studies on de-obfuscated program control flow. Finally, our research evaluates the flattening designs and indicates possible improvements for future code obfuscation studies.

7.2 Discussion on de-flattening against basic flattening variants

Till this point, we only evaluate our model for basic control flow flattening. However, our method should also work well with variants of basic control flow

flattening methods, i.e. control flow flattening with artificial blocks and control flow flattening with inter procedure calls. This section discusses the applicability of our de-flattening model against the two variants.

In the construction of the de-flattening model, the ultimate bases of our work are the assumptions drawn from control flow flattening schemes (Section 3.1). The assumptions are constructed based on all the three control flow flattening schemes. Therefore our idea is applicable to other flattening schemes. However in order to work with the new variants, small modification might be required.

For the control flow flattening with artificial blocks, our de-flattening model is able to rule out artificial blocks during the execution path generation. In the execution path computation procedure, as the artificial blocks are dead blocks, they are not able to exist in any real execution paths. Therefore our model could also work against this flattening method. The only potential problem is the execution path computation time. As artificial basic blocks are added, the count of basic blocks increases. During the execution path exploration, more execution paths could be constructed. Thereby the computation time to verify the feasibility of paths could increase dramatically. While our de-flattening model sacrifices computation time for accuracy, the de-obfuscation on the assembly code merely with static approaches could be very imprecise.

The same goes for control flow flattening with inter procedure calls. Even though in this flattening scheme, the value of the dispatcher variable is passed by an array and randomness is introduced during the array initialization, as long as the same input set results in the same execution path, our model is applicable. In order to address this type of flattening methods, we should strengthen the path feasibility computation in two aspects: (1) The mapping function f should be able to handle the instructions in other invoked functions. (2) The variable set $\mathcal{V}$ should be improved to provide better supports to arrays and pointers.

Therefore, the de-flattening model presented in this report is suitable for extension. While we do not verify the result with examples, this might be left for the future work.

7.3 Related work

There is some work which also addresses de-obfuscation. A broad range of studies [SPY] [CLD11] [CJ06] have focused on de-obfuscation on binaries, especially the binaries of malware. However most of the studies in this realm have targeted at certain simple obfuscation methods such as block splitting and dead block

insertion. In these studies, the primary idea to remove the simple obfuscation is to use several significant techniques in static analysis such as UD chains and programing slicing [NNH99]. Among the studies, Coogan et al.'s work [CLD11] is valuable to mention, as they present a de-obfuscation method that is independent from the obfuscation added. Their study is ultimately based on the assumption that the instructions from the original program are the instructions that affect the behavior of the program. Then they provide an approach to keep record of the important instructions that contribute to the behavior of the program. By distinguishing the important instructions from the whole instructions, they recognize the real program instructions and obfuscator-added instructions. However, with their method, the analysis result could be inevitably imprecise.

Few studies have addressed de-obfuscation on the structure of a program as complex as control flow flattening. Udupa et al. [UDM05] have discussed the de-obfuscation techniques that are subjected to this transformation. In their work, they also combine dynamic analysis with static analysis together to work against the three control flow flattening schemes. The path feasibility analysis in their work is not as strong as the one presented in this report. As a result, they cannot obtain the original program control flow. On the other hand, they evaluate their work by computing how many edges added due to flattening are correctly or wrongly detected with their de-obfuscation model. However, false positive errors and false negative errors cannot be avoided in their model. The work presented in this report enhances their work by performing all the analysis on CFGs and adopting a SMT solver to compute feasible paths efficiently.

7.4 Future work

For the future work, we suggest four directions:

The first direction is to expand the supports for more complex, bigger examples. Currently the supported language syntax is limited, we cannot handle the whole C syntax. As an example, we cannot fully handle pointers or arrays. The current implementation also cannot support the mapping f from any assembly instructions in AT&T syntax to a logical formula. Therefore, one of the possible future work is to extend the supported language syntax. This relates to including pointers as program inputs and extending the mapping function f to support more instructions.

The second work direction, as suggested in previous sections, is to address the enhanced control flow flattening methods. To be successful in this direction, essential modification is required on the mapping function f. Besides, ample

tests against the enhanced flattening approaches should also be included.

The third research direction could be the construction of a full measuring framework in terms of graph similarity and computation time. First, in the current solution, we could only answer the question like whether a de-flattened program is exactly the same as the original program or not. In this case, the answer can only be a number from the set $\{1, 0\}$. However, our model could be extended by building a graph similarity measuring framework to give the answer as a number from the range [0, 1]. Second, in the current discussion we do not address the changes of the computation time when the size of a program grows. In the future work, we could also build a measuring framework to discuss the relationship between the computation time of algorithms and the size of programs.

Finally, the forth research direction lies in new obfuscation methods against dynamically analysis. The de-flattening model proposed in this report mainly uses execution paths obtained by running an obfuscated program dynamically. The current experiment and discussion result suggests that this de-flattening could invalidate most of the flattening methods. Therefore, for the forth work direction, researchers could try to work against our de-flattening model with new obfuscation schemes.

7.5 Summary

In this chapter, we have concluded all our work and proposed possible future work directions. Our research aims to eliminate de-obfuscation on low level code. This research evaluates the strengths and weaknesses in several obfuscation schemes. It provides new views for improvements in future obfuscation studies.

Bibliography

[AO08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[Ass] Computer Organisation and Assembly Language - Intel and AT&T Syntax, howpublished = `http://www.imada.sdu.dk/courses/dm18/litteratur/intelnatt.htm`, note = Accessed: 2014-06-15.

[Bel99] Thoms Bell. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, October 1999.

[BKH05] Nerina Bermudo, Andreas Krall, and Nigel Horspool. Control flow graph reconstruction for assembly language programs with delayed instructions. In *Source Code Analysis and Manipulation, 2005. Fifth IEEE International Workshop on*, pages 107–116, Sept 2005.

[BMM06] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *In Proceedings of International Symposium on Secure Software Engineering. IEEE*, 2006.

[Boo] Boomerang - A general, open source, retargetable decompiler of machine code programs, howpublished = `http://boomerang.sourceforge.net/`, note = Accessed: 2014-06-15.

[Cap12] Jan Cappaert. *Code Obfuscation Techniques for Software Protection*. PhD thesis, Katholieke Universiteit Leuven, Arenbergkasteel, B-3011 Heverlee (Belgium), 4 2012.

[CGJZ01] Stanley Chow, Yuan Gu, Harold Johnson, and VladimirA. Zakharov. An approach to the obfuscation of control-flow of sequential computer programs. In GeorgeI. Davida and Yair Frankel, editors, *Information Security*, volume 2200 of *Lecture Notes in Computer Science*, pages 144–155. Springer Berlin Heidelberg, 2001.

[CJ06] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. Technical report, DTIC Document, 2006.

[CLD11] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 275–284, New York, NY, USA, 2011. ACM.

[CP10] Jan Cappaert and Bart Preneel. A general model for hiding control flow. In *Proceedings of the Tenth Annual ACM Workshop on Digital Rights Management*, DRM '10, pages 35–42, New York, NY, USA, 2010. ACM.

[dMB10] Leonardo de Mooura and Nikolaj Bjørner. Z3 - a tutorial. Technical report, Microsoft Reaearch, 2010.

[Dot] PreEmptive Solutions. Dotfuscator, howpublished = `http://www.preemptive.com/products/dotfuscator/overview`, note = Accessed: 2014-06-15.

[EJPJ] D Evangelin, J Jelsteen, J Alice Pushparani, and J Nelson Samuel Jebastin. Importance of software re-engineering process and program based analysis in reverse engineering process.

[Fin] FindBugs™ - Find Bugs in Java Programs, howpublished = `http://findbugs.sourceforge.net/`, note = Accessed: 2014-06-15.

[FxC] FxCop, howpublished = `http://msdn.microsoft.com/en-us/library/bb429476(v=vs.80).aspx`, note = Accessed: 2014-06-15.

[gdb] GDB: The GNU Project Debugger, howpublished = `https://www.gnu.org/software/gdb/`, note = Accessed: 2014-06-15.

[GMGM] Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira. An overview on the static code analysis approach in software development. Technical report.

[Gra] Graphviz - Graph Visualization Software, howpublished = `http://graphviz.org/`, note = Accessed: 2014-06-15.

[ida] IDA Hex-Rays Home, howpublished = `https://www.hex-rays.com/index.shtml`, note = Accessed: 2014-03-19.

[JGr] Welcome to JGraphT - a free Java Graph Library, howpublished = `http://jgrapht.org/`, note = Accessed: 2014-03-19.

[Joh05] David S. Johnson. The np-completeness column. *ACM Trans. Algorithms*, 1(1):160–176, July 2005.

[Lin] Lint, howpublished = `http://developer.android.com/tools/help/lint.html`, note = Accessed: 2014-06-15.

[Luk82] Eugene M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *Journal of Computer and System Sciences*, 25(1):42–65, 1982.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[SPY] Hassen Saıdi, Phillip Porras, and Vinod Yegneswaran. Experiences in malware binary deobfuscation.

[SS02] Eleni Stroulia and Tarja Systä. Dynamic analysis for reverse engineering and program understanding. *SIGAPP Appl. Comput. Rev.*, 10(1):8–17, April 2002.

[SSCS07] Raghunathan Srinivasan, Raghunathan Srinivasan, Charles Colbourn, and Aviral Shrivastava. Protecting anti-virus software under viral attacks by, 2007.

[Tec] Technicolor - Technology-driven company for Media & Entertainment, howpublished = `http://www.technicolor.com/`, note = Accessed: 2014-03-19.

[UDM05] Sharath K. Udupa, Saumya K. Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *Proceedings of the 12th Working Conference on Reverse Engineering*, WCRE '05, pages 45–54, Washington, DC, USA, 2005. IEEE Computer Society.

[Wan00] Chenxi Wang. A security architecture for survivability mechanisms, 2000.

[WHKD00] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical report, Charlottesville, VA, USA, 2000.

[WHKD01] Chenxi Wang, Jonathan Hill, John C. Knight, and Jack W. Davidson. Protection of software-based survivability mechanisms. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (Formerly: FTCS)*, DSN '01, pages 193–202, Washington, DC, USA, 2001. IEEE Computer Society.

[XSS09] Liang Xu, Fangqi Sun, and Zhendong Su. Constructing precise control flow graphs from binaries. *University of California, Davis, Tech. Rep*, 2009.

[Z3p] Python: module z3 - Microsoft Research, howpublished = `http://research.microsoft.com/en-us/um/redmond/projects/z3/z3.html`, note = Accessed: 2014-06-15.

Appendix A

Examples used for evaluation

This appendix presents all the source code of the examples that are used for evaluation in Chapter 4 and Chapter 6.

A.1 Example 1

Listing A.1: Example 1

```
#include <stdio.h>

int main(int argc , char** argv) {
        printf("Hello World!\n");
        return 0;
}
```

A.2 Example 2

Listing A.2: Example 2

```
#include <stdio.h>
```

```
int main(int argc , char** argv) {
        if(argc > 1)
                printf("Hello %s!\n", argv[1]);
        else
                printf("Hello World!\n");
        return 0;
}
```

A.3 Example 3

Listing A.3: Example 3

```
#include <stdio.h>

int main(int argc, char** argv) {
        switch (argc) {
        case 1:
                printf("apple\n");
                break;
        case 5:
                printf("egg\n");
                break;
        case 2:
                printf("bananas\n");
                break;
        default:
                printf("%d\n",argc);
        }
}
```

A.4 Example 4

Listing A.4: Example 4

```
#include <stdio.h>
#include <math.h>

int main(int argc, char** argv) {
        int i = 5;
        int s = 0;
        while (i <= argc) {
                s += i;
                i++;
        }
        printf("s=%d, i=%d\n", s, i);

}
```

A.5 Example 5

Listing A.5: Example 5

```
#include <stdio.h>
#include <math.h>


#define HASH_A 0.6180339887498949
#define HASH_M 4294967295

unsigned int Hash(unsigned int x)
{
        return (unsigned int) floor(HASH_M * (x*HASH_A -floor (x*HASH_A)));
}


void debug_dyn(unsigned int a)
{
        printf ("DYN: %d\n",a);
}

void debug_sw(unsigned int a)
{
        printf ("SW: %x\n",a);
}
void debug_case(unsigned int a)
{
}



int f()
{
        printf ("function f\n");
        return 100;
}

void g(int a)
{
        printf ("DEBUG %x\n",a);

}

void gs(char* a)
{
        printf ("DEBUGSTR %s\n",a);

}

int main(int argc, char** argv)
{

        if (argc<2)
```

```
        {
                printf("hello world arg<2\n");
        }
        else
        {
                char* param = argv[1];
                char c = param[0];
                if (argc <3)
                {
                        printf("hello world arg == 2\n");
                }
                else
                {
                        if (argc >5)
                        {
                                printf("****hello world arg>5\n");
                                g(c);
                                switch (c)
                                {
                                        case 'a': printf("=>AA\n"); break;
                                        case 'f': printf("=>EE\n"); break;
                                        case 'z': printf("=>ZZ\n"); break;
                                        default : printf("=>unknown char\n"); break;
                                }
                        }
                        else
                        {
                                printf("hello world =3 ou 4 ou 5\n");

                        }
                }
        }
}
```