

國立交通大學

資訊科學與工程研究所

碩士論文

透過例外機制實作程式模糊化



Obfuscation Using Exception Handling

研究生：黃致超

指導教授：楊 武 教授

中華民國九十九年九月

透過例外機制實作程式模糊化

Obfuscation Using Exception Handling

研究生：黃致超 Student：Chih-Chao Huang

指導教授：楊 武 博士 Advisor：Dr. Wu Yang



Submitted to Institute of Computer Science and Engineering
College of Computer Science and Information Engineering

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Institution of Computer Science and Engineering

September 2010

Hsinchu, Taiwan, Republic of China

中華民國九十九年九月

透過例外機制實作程式模糊化

學生：黃致超

指導教授：楊 武 博士

國立交通大學資訊科學與工程研究所

摘要

為了保護軟體不被反組譯工具順利的轉為原始碼進而竊取其中重要的演算法，程式模糊化(obfuscation)技術是所有方法中最簡單卻又能達到很好效果的方法之一。一個好的程式模糊化技術不但能使被模糊後的程式不容易看懂，還能確保其程式正確性。

舊有的研究絕大多數都對於程式碼的轉換(program transformation)來達到模糊化的效果，雖然此種方法可順利的讓程式難以辨別，但卻會使得在編譯優化(compiler optimization)時無法套用所有的優化程式，進而使得產生的程式效能跟原本的程式差非常多

在本論文中，我們利用動態例外(runtime exception)不輕易在編譯時期(compile time)被發現的特性來把真正的程式碼隱藏起來，並加入一些假的程式碼(bogus code)來增加攻擊者判斷的難度。程式經過我們的方法後，從實驗數據可得知我們的方法除了確保程式正確性外，對於效能和程式大小的增加也不大，爾且經過我們方法的程式也無法再被反組譯回原始碼。

Obfuscation Using Exception Handling

Student: Chih-Chao Huang

Advisor: Dr. Wu Yang

Institute of Computer Science and Engineering

National Chiao Tung University

Abstract

There exist several reverse engineering tools that can easily recover source code from a lower level immediate representation. To protect intellectual property, obfuscation is one of the easiest and efficient way to achieve this goal. A good obfuscation tool can not only makes the obfuscated code much harder to understand but also ensures the correctness.

Previous obfuscating approaches mostly use program transformation that base on opaque predicate to obfuscate control flow transfer. However, although these methods can provide a good resilient, they usually decrease performance a lot if applied on the whole program.

In this paper, we use runtime exception to hide the real code. During obfuscation, the original program is obfuscated by changing each loop into a specific runtime exception and inserting bogus code after the runtime exception. The obfuscated code's correctness is maintained but the code is now unable to be decompiled. Experiment results show that our obfuscation technique increase less overhead and code size on SPECJVM2008.

誌謝

能完成此論文首先要感謝楊武教授在我碩士年兩中不斷的給予指導，提出許多我沒想到的盲點，讓我透過不斷的修正，最終能完成此研究。同時也要感謝口試委員游坤明教授，徐慰中教授以及雍忠教授，各位的意見令我的論文更具有說服力。

另外也要感謝安森貓，死敗，morain 及 SOD WU 學長們，由於你們適時的在我研究遇到困難時伸出援手，才能有此論文的產生。當然也要感謝 PLASLAB 的夥伴們，實驗室的歡樂氣氛使我在研究的路途上獲得了許多幫助，我們這一屆更與學長們共創了 PLASLAB 史上最輝煌，人最多的一段。

最後，當然要感謝我的父母及女友佳宜的支持與照顧，沒有你們在後方相挺，我一定無法完成此研究。僅將此篇論文獻給你們。

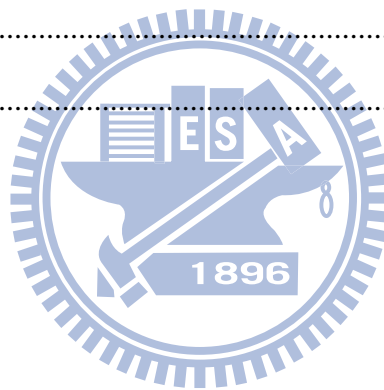


The work reported in this paper is partially supported by National Science Council, Taiwan, Republic of China, under grants NSC 96-2628-E-009-014-MY3, NSC 98-2220-E-009-050, NSC 98-2220-E-009-051, and 99-2219-E009-013 and a grant from Sun Microsystems OpenSparc Project.

Table of Contents

摘要.....	iii
Abstract	iv
誌謝.....	v
Table of Contents	vi
List of Figures	viii
Chapter 1 Introduction	1
Chapter 2 Related Work	4
2.1 Obfuscation Background	4
2.1.1 Layout Obfuscation	5
2.1.2 Data Obfuscation	5
2.1.3 Control Flow Obfuscation	6
2.1.3 Preventative Obfuscation	7
2.2 Control Flow Flattening.....	8
2.3 Binary Obfuscation Using Signals.....	8
Chapter 3 Motivation	11
Chapter 4 Implementation	12
4.1 Overview.....	13
4.2 How To Find Loops?.....	15
4.2.1 Basic Loop	15
4.2.2 Same Loop Header	16
4.2.3 Multi-Loop	16
4.3 Setup, Restore	17
4.4 Runtime Exception.....	18
4.4.1 Runtime Exception Collision Problem	18

4.4.2	Static Initializer	19
4.5	Bogus Code	20
4.5.1	Intersection Loop Obfuscation	21
4.6	Transfer Table	22
4.7	Local Variable Inconsistent	26
Chapter 5	Experiment	28
5.1	Performance of the Obfuscated Program	29
5.2	Code Size	31
5.3	Decompiler	32
5.4	Visualizing the Effects of Obfuscation	33
Chapter 6	Conclusion	34
Bibliography	36



List of Figures

2.1	Obfuscation concept.	5
2.2	Data obfuscation categories.	6
2.3	An example of a data encoding obfuscation.	6
2.4	Control flow obfuscation categories.	7
2.5	Internal concept of an obfuscator.	8
2.6	Source code and its control flow graph before flattening.	9
2.7	After control flow flattening.	9
2.8	Summary of Source Code Transformations [13].	10
4.1	Framework of our obfuscation.	13
4.2	Example for basic loop pattern.	15
4.3	Example for same loop header pattern.	16
4.4	Example for multi loop pattern.	17
4.5	Relationship between loop and original exception.	18
4.6	Before and after handling exception collision case two.	19
4.7	Concept of adding bogus code.	20
4.8	Gap between Java source code and Java bytecode.	21

4.9	Intersection loop obfuscation example. (1) is the original control flow and the rest is the transformation process for each step.	23
4.10	Loop after applying intersection loop obfuscation.	24
4.11	Concept of hashing source address.	25
4.12	Process of transfer target to PC.	25
4.13	Use switch to build transfer table.	26
4.14	How local variable inconsistent occur.	27
5.1	Performance on SpecJVM2008.	29
5.2	Relationship among Utils.java and other programs.	30
5.3	Performance on SpecJVM2008 without obfuscate Utils.java.	30
5.4	Code size increased after obfuscation.	31
5.5	Test result of decompilation obfuscated program.	32
5.6	Intuitive view on the effect of our obfuscation method.	33

Chapter 1

Introduction

To obtain good portability, there are many programming languages transform their source code into a platform independence intermediate representation (IR), such as Java and .NET, then compile the IR into a specific target's machine code. Imagine, if we can transform all programming languages into a common IR, compiler's optimization techniques can be developed without considering each language's features and portability can easily be achieved by using a VM translate the IR to a specific target's binary. But before this idea can be put into practice, security is the most important issue we need to think about.

Reverse engineering tools such as decompiler is used to recover source code and meaning from a IR. There are many free decompilers can be download from internet, such as Jad, CavaJ, JD-GUI and Mocha. Software hackers often use these tools to discover the valuable algorithms in software.

Obfuscation is one of the efficient tools that can be used to protect the intellectual property. The goal of obfuscation is to make it difficult for an attacker from understanding the meaning and structure of a program. It works by transforming a program into a functionally-equivalent one but not readable for human reader. In this way, an attacker may takes more time to reverse engineering than rewrite a same program.

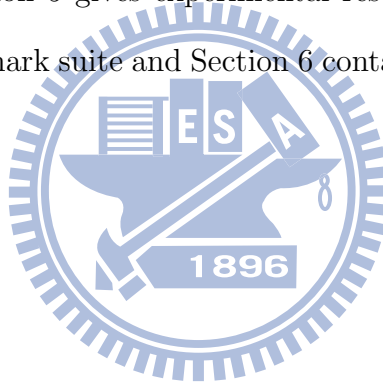
Previous researches on obfuscation mostly focus on program transforms that rely on opaque predicates to obfuscate the control flow. Although it can successfully blur the program, these methods usually come in with notable decrease in performance when applied on the whole program.

In this thesis, our obfuscation technique is to obfuscate intermediate representations that rely on runtime exception to against decompilers, though using runtime exception to implement obfuscation is not a new idea [10]. Prior works usually use runtime exception to hide the real control flow, but they did not notice that the exception table contains all the exception information. That is, an attacker can find out all the mapping of runtime exception by analyzing the exception table of the program.

The basic idea of our approach contains three parts: (1)find loops then move the loops away; (2)substitute the loops by runtime exceptions; and (3)build a transfer table that map the runtime exception to the substituted loop. To complicate the mapping of each runtime exception, we use a transfer table and hash function to further hide the real control flow. In this way, the problem of easy to break can be solved.

We have used BCEL [1] to implement our method and experimented on Java bytecode. The experiment shows that our approach can be done efficiently with moderate decrease in performance. We also test the obfuscated program on many decompilers, and none of them can be decompile successfully.

The rest of this paper is organized as follows: Section 2 provides background and related work of obfuscation. Section 3 describes the motivation of our work. Section 4 describe the techniques we used and explain how they are implemented. Section 5 gives experimental results for programs in the SPECJVM2008 benchmark suite and Section 6 contains concluding remarks.



Chapter 2

Related Work

This section reviews the background of obfuscation and gives a brief introduction to two related works that represent recently proposed mechanisms.

2.1 Obfuscation Background

It's easy to reverse engineer Java class files since Java bytecode contains a lot of the same information as its original source code. To overcome this problem, obfuscator can give us some help.

An obfuscator is a program used to transform original program. The output of the obfuscated code is more difficult to understand but is functionally-equivalent to the original. Figure 2.1 shows the process and concept of an obfuscator.

Obfuscation can be classified according to what kind of information they target and how they affect their target [8] [11]:

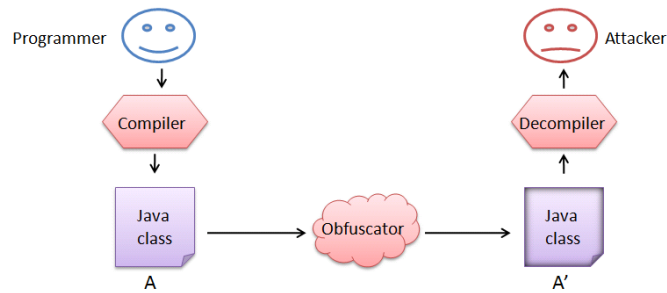


Figure 2.1: Obfuscation concept.

2.1.1 Layout Obfuscation

Layout obfuscation affect the information in the program code that is unnecessary to its execution. These obfuscations are typically trivial and reduce the amount of information available to a human reader. Examples include source code formatting, variable names and comments [5].

2.1.2 Data Obfuscation

Data obfuscations operate on the data structures used in the program. Can be classified according to what operation they perform on the data structures. Figure 2.2 shows the classification.

Data encoding obfuscations affect how the stored data is interpreted, for example replacing an integer variable i by the expression $8 * i + 2$. Source code would be transformed in the manner of Figure 2.3

Data aggregation obfuscation change how data is grouped. For example, transforming a two-dimensional array into a one-dimensional array and vice-versa.

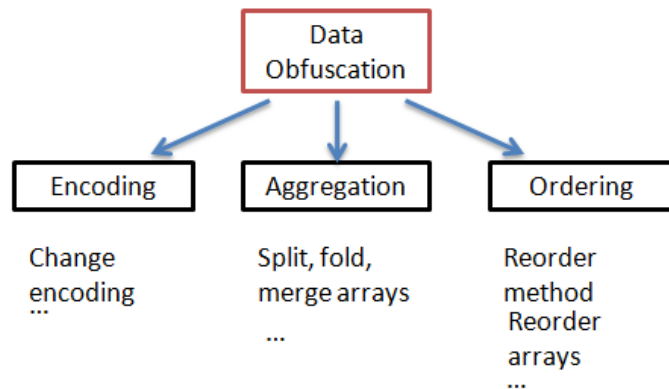


Figure 2.2: Data obfuscation categories.

```

int i = 1;
while ( i < 1000 ) {
  ... A[ i ] ...;
  i++;
}
  
```

⇒

```

int i = 11;
while ( i < 8002 ) {
  ... A[ ( i - 2 ) / 8 ] ...;
  i+=8;
}
  
```

Figure 2.3: An example of a data encoding obfuscation.

Data ordering obfuscations change how data is ordered. The normal way in which an array is used to store a list of integers has the i th element in the list at position i in the array. Instead, we could use a function $f(i)$ to determine the position of the i th element in the list.

2.1.3 Control Flow Obfuscation

Control flow obfuscation affect the contorl flow of the program. Figure 2.4 shows the classification according to what operation they perform.

Control aggregation obfuscations change the way in which program statements are grouped together. For example, it is possible to inline proce-

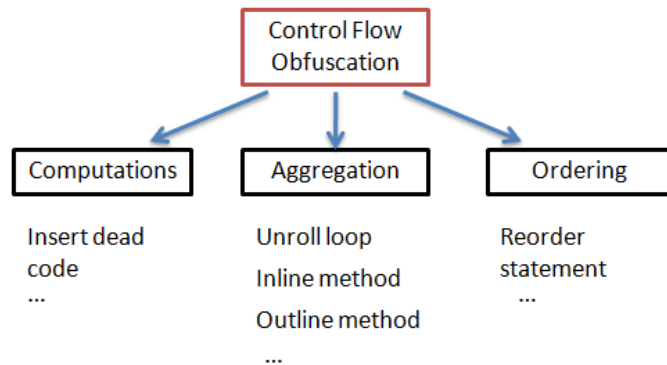


Figure 2.4: Control flow obfuscation categories.

cedure, that is, replacing a procedure call with the statements from the called procedure itself.

Control ordering obfuscations change the order where statements are executed. For example, loops can sometimes be made to iterate backwards instead of forwards.

Control computation obfuscations hide the real control flow in a program. For example, statements which have no effect can be inserted into a program.

2.1.4 Preventive Obfuscation

The main goal is to stop decompilers and deobfuscators from functioning correctly by using the ambiguities and irregularities left in the language [6] [5].

In general, an obfuscator usually contains lots of different obfuscation

methods and provide an interface to let user selects the required level of obfuscation and the maximum execution time/space penalty that the obfuscator is allowed to add to the application [9]. Figure 2.5 shows the internal concept of an obfuscator. In our work, we only propose an obfuscation method instead of implementating a complete obfuscator.

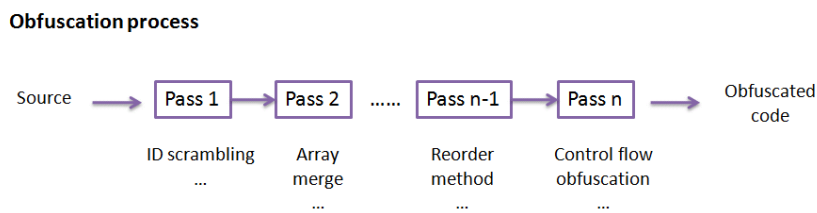


Figure 2.5: Internal concept of an obfuscator.

2.2 Control Flow Flattening

Control flow flattening aims to obscure the control flow logic of a program by "flattening" the control flow graph so that all basic blocks appear to have the same set of predecessors and successors. The actual control flow during execution is guided by a dispatcher variable [14]. Figure 2.6 and Figure 2.7 shows the program that before and after control flow flattening obfuscation.

2.3 Binary Obfuscation Using Signals

Previous obfuscating approaches mostly use program transformation rely on reconstruct original program's structure. For example, they use opaque pred-

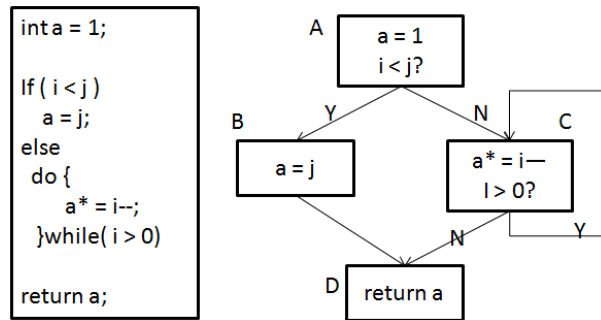


Figure 2.6: Source code and its control flow graph before flattening.

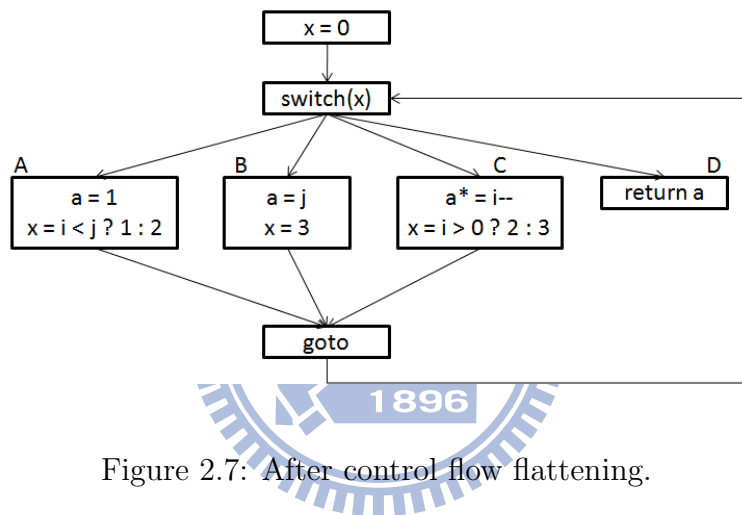


Figure 2.7: After control flow flattening.

icates to hide the control flow transfer and then insert bogus code in untaken path to obfuscate the data flow. Although it can successfully obfuscate the origin program, it break the origin structure of program. So it may cause the compiler can not apply all optimization techniques on it, and further get a notable performance degradation if applied on the whole program [7]. Such obfuscation approach like control flow flattening [14] [15]. To overcome this drawback, researchers recently propose using signal handling as a mechanism for obfuscation.

In *Binary Obfuscation Using Signals* [13], it describes two techniques for obfuscating binaries. The primary technique is to replace control transfer instructions—jumps, calls, and returns by instructions that raise traps at runtime; these traps are then fielded by signal handling code that carries out the appropriate control transfer. The secondary technique is to insert (unreachable) code after traps that contains fake control transfers and that make it hard to find the beginning of the true next instructions. Figure 2.8 shows the summary of how this method work.

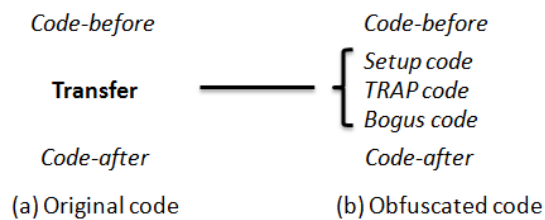
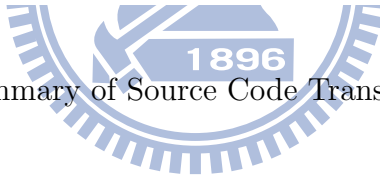


Figure 2.8: Summary of Source Code Transformations [13].



Chapter 3

Motivation

Although *Binary Obfuscation Using Signals* [13] can solve prior obfuscation approaches' problems, it still has two main problems. The first and most important one is this approach may incur high performance overhead due to the high cost of signal handling. For example, it could incur more than 43X performance overhead when obfuscating 90% of the branches [7]. The next problem is that since it is a binary obfuscation, it can only be used on specific environment, that is, it is not portable.

Based on these reasons, we propose a framework that not only can offer high quality of obfuscation but also with moderate increase in execution time and code size.

Chapter 4

Implementation

Our method is similar to *Binary Obfuscation Using Signals* [13], the most difference is we focus our technique on intermediate code instead of binary, so it is not limited by a single environment.

Our main obfuscation technique is to change loops into code sequences that cause runtime exception, then it will jump to a table we called **Transfer Table**. The role of this table is like an address controller, its responsibility is to accept all the input from different runtime exceptions we made, and find the unique output then jump to the corresponding address.

The other obfuscation technique is to add bogus code after the handmade runtime exceptions. Since runtime exception is not easy to be detect at compile time, bogus code may increase the strength of our obfuscation. To further make the added bogus code hard to be decompiled, we propose a technique called **Intersection Loop Obfuscation** and applied on it.

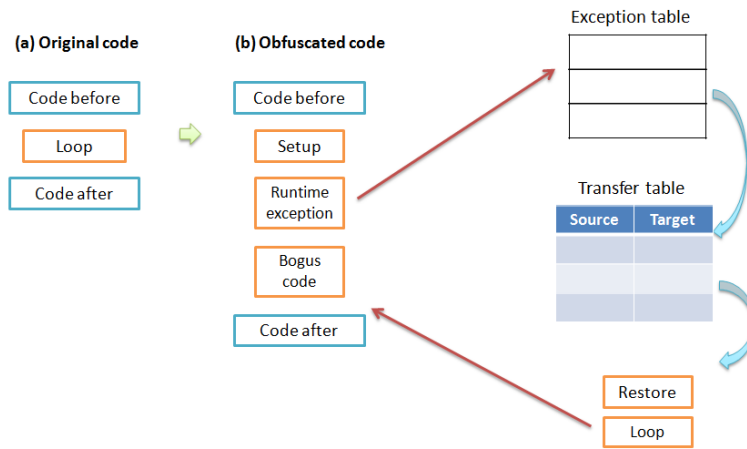


Figure 4.1: Framework of our obfuscation.

Below we give an overview of how these techniques are implemented. Then we describe in detail how to solve the runtime exception collision, how we deal with the implementation of transfer table, and some other issues we faced.

4.1 Overview

Figure 4.1 summarizes the overview of our framework. Figure 4.1(a) contains a fragment of machine code in the origin code. **Loop** is our obfuscation target, and the instructions that is proceeded and followed Loop are indicated by **Code before** and **Code after**.

The reason why we choose loop as our obfuscation target is that it will not cause a notable degradation of performance while applied our framework on the whole program. Imagine, if we consider jump or procedure call to

obfuscate, then a runtime exception will be occurred for each iteration in a loop. Since an exception incurs high overhead, it will lower performance a lot.

4.1(b) contains the corresponding code fragment in the obfuscated code. Loop is replaced by three components, which will be described in the following sections.

When executing the obfuscated code, it causes the runtime exception. Then the runtime exception handler will transfer control to the **Transfer Table** which contains mapping from the address of raising runtime exception to the Loop. Since the source address that raised runtime exception has been saved to a local variable in **Setup**, so we can use this source to find the corresponding target in the Transfer Table and then execute the Loop.

Before discussing in detail for each component, there is one more issue we need to consider about. Since our method will produce so many runtime exceptions, will these exceptions increase too many overhead to performance? This problem can be solved by an optimization called *Exception-Directed Optimization*(EDO) [12] that is developed by IBM. EDO is a feedback-directed dynamic optimization. It attempts to detect hot exception paths, when it find the exception is hot enough, it will inline the hot exception path to the program. So next time it won't raise an exception and save a lot of overhead.

With help from EDO, our method can be done efficiently with moderate increases in execution time.

4.2 How To Find Loops?

When obfuscating a program, our obfuscator will find loops first. We use a simple heuristic that first appeared in Dynamo [4] to identify loop headers.

After loading a class file, our obfuscator traverses the bytecode program instruction by instruction in each method. Each time a backwards branch instruction is found, our obfuscator put it into a backwards branch array since the destination of that jump is a loop header. When all backwards branches are gathered, the way to find a loop can be classified according to the position of each backwards branch.

4.2.1 Basic Loop

A basic loop is a pattern that there contains no other backwards branches in the scope of a backwards branch and its corresponding loop header. Figure 4.2 shows the concept.

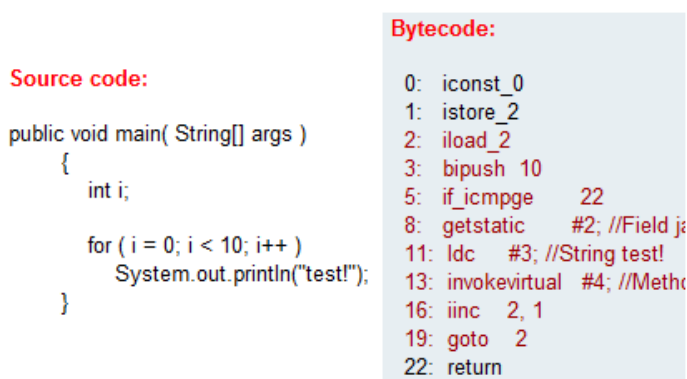


Figure 4.2: Example for basic loop pattern.

4.2.2 Same Loop Header

A same loop header pattern means in the scope of a backwards branch and its corresponding loop header, there exist other backwards branches that share the same loop header. Figure 4.3 shows this pattern.

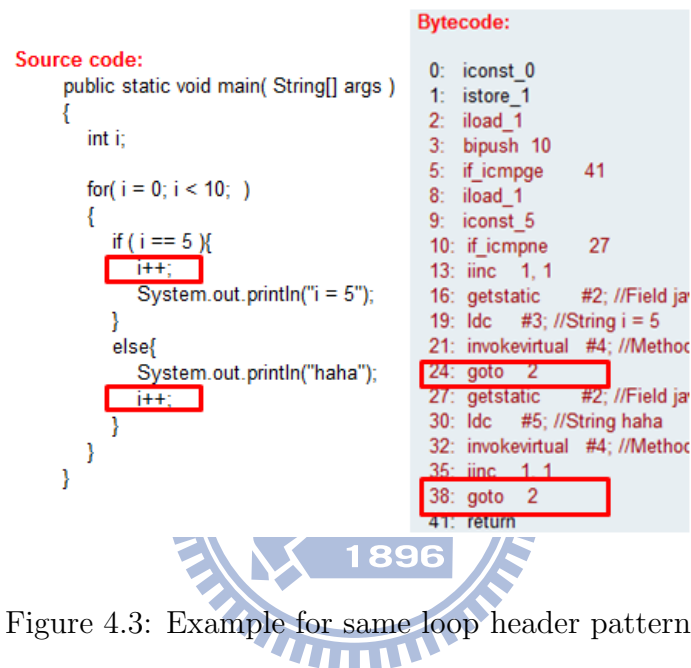


Figure 4.3: Example for same loop header pattern.

4.2.3 Multi-Loop

A multi-loop pattern means for a loop constructed by one backwards branch and its corresponding loop header, there exist another backwards branch and loop header forms a bigger scope and includes the loop. Figure 4.4 shows the concept.

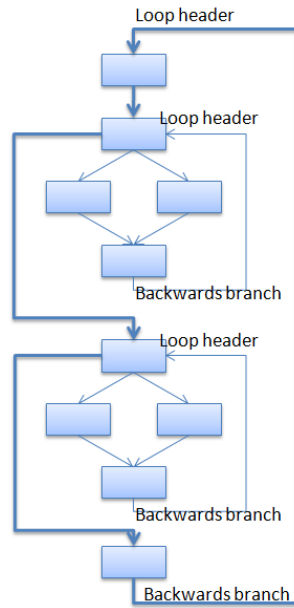


Figure 4.4: Example for multi loop pattern.

4.3 Setup, Restore

After finding loops, we then doing some initialization actions before obfuscating the loop. **Setup** component is responsible to do the initialization. The setup component does three things: (1) store the source address(runtime exception's location) which will be used when building transfer table; (2) push the source address to operand stack and store it to a specified local variable so it can later be used by transfer table; and (3) save operand stack's values to heap so the original program's state can later be restored by **Restore** component to exactly what it was before the Loop.

4.4 Runtime Exception

After initialization, before we can move loop away and insert a runtime exception, there is still one more thing we need to consider about. Will the runtime exceptions we made collision with exceptions in original program? Figure 4.5 shows all possible situations for relationship between loop and original exception.

4.4.1 Runtime Exception Collision Problem

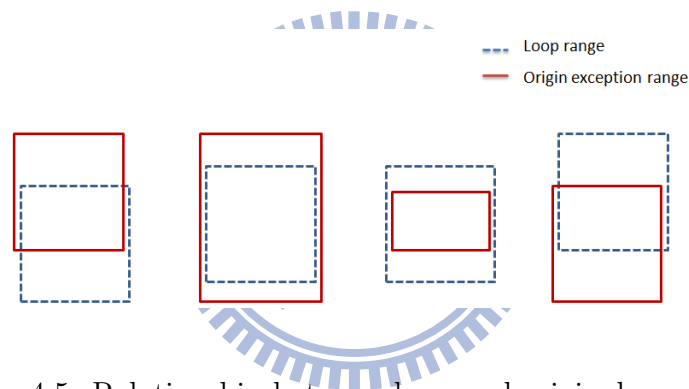


Figure 4.5: Relationship between loop and original exception.

In figure 4.5, we can see that case one and case four are impossible, so we don't have to handle it. Case two means a loop is included by an exception and case three is opposite. There are two steps to handle case two: (1) find the original exception's exception type and handler, then set to the moved loop; and (2) divide the original exception into two parts and insert our runtime exception between them. Figure 4.6 shows the concept. In this way, the collision problem can be solved. For case three, since the exception will be moved together with the loop, so there is no collision can happen. After

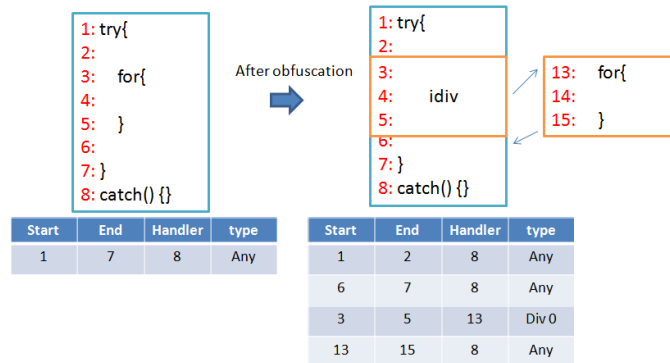


Figure 4.6: Before and after handling exception collision case two.

dealing with collision problem, we can now insert our runtime exception.

4.4.2 Static Initializer

To further confuse attackers, we use some techniques to implement our runtime exception. For example, when an **divide zero** exception want to be raised, before executing an **idiv** bytecode, operand stack must contains a zero value. So our concept is to store values that will be used in runtime exception to heap in advance and then get these values by reference in our runtime exception. Thus, attacker won't easily figure out what values are contained in operand stack.

There are two steps to achieve this goal: (1) write a static initialize method that store the value will be used in runtime exception to heap in advance; and (2) use static initializer to call the static initialize method. Since the code in a static initializer block is executed automatically by the virtual machine when the class is loaded, the values we want to initialize will

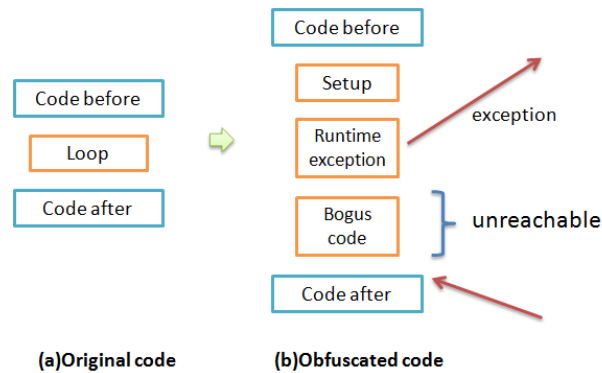


Figure 4.7: Concept of adding bogus code.

be put in heap at the same time.

In this way, the attacker is hard to figure out whether a runtime exception will happen or not because it becomes an inter-procedure data-flow analysis problem.

4.5 Bogus Code

After obfuscating the Loop, we then insert bogus code to further confuse attackers. Figure 4.7 shows the concept of how we insert bogus code after runtime exception.

The main reason why we insert bogus code after runtime exception is to increase the strength of our obfuscation. Although bogus code is unreachable, attacker is not easy to identify since runtime exception is hard to detect at compile time. The other benefit of bogus code is that it can make an attacker think there is another edge in the control flow graph.

Bogus code is composed by a loop that chosen from other place in the

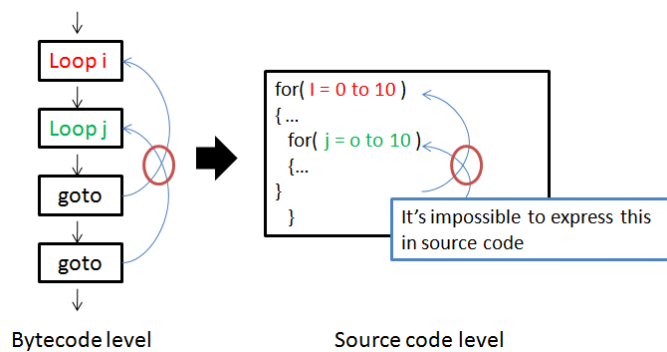


Figure 4.8: Gap between Java source code and Java bytecode.

program. Since previous researches on obfuscation almost focus on control flow, so using loop as bogus code may misleading attackers.

4.5.1 Intersection Loop Obfuscation

It is trivially true that every valid Java source code program must compile to a valid Java class file. A valid class file must pass through the verifier without causing any errors. However, not every Java class file has a direct correspondence to a valid Java source code program. This is because the Java bytecode instruction set supports a richer set of language features than the language Java. These features include goto and subroutine instructions [11].

To further confuse attackers, we use this gap between Java bytecode and Java source code to propose a new technique called **intersection loop obfuscation**. Figure 4.8 shows the concept of our idea. As can be seen in the graph, a intersected loop is not permitted in any high-level language. Therefore, it can make decompilers fail.

Our method aims to transform bogus code into intersected loop. The transformation process contains five steps: (1) tail duplication; (2) duplicate conditional block and put onto the original conditional block; (3) move one of the tail above duplicated conditional block; (4) add a GOTO above whole loop and point to the duplicated conditional block; and (5) retarget the tail's destination to the duplicated conditional block. Figure 4.9 shows an example of how our method work, and Figure 4.10 depict the transformed control flow, as can be seen in Figure 4.10 black line and dotted line form intersection loop.

After intersection loop obfuscation, the bogus code is transformed from a "loop" into a "intersected loop" that can resist decompilers. Although this transformation breaks the origin control flow of the loop and may cause the compiler can not apply all optimization techniques on it then further get a notable performance degradation, since bogus code is unreachable, performance will not be affected.

4.6 Transfer Table

After obfuscating each Loop and inserting Bogus code, all information that needs to build transfer table are gathered. A transfer table act like a controller, when handmade runtime exception occurs, the exception handler will guide the next execution instruction to our transfer table and then the table will be traversed to find the corresponding target address and set this address to PC. After doing that, execution can be continued from the beginning of

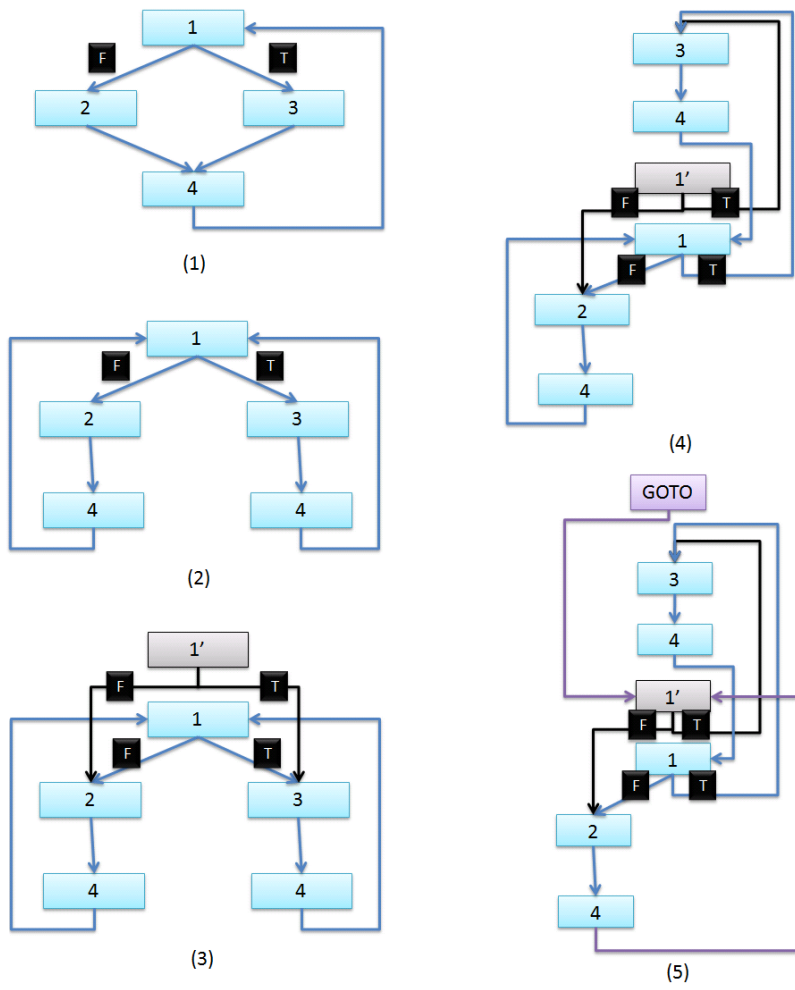


Figure 4.9: Intersection loop obfuscation example. (1) is the original control flow and the rest is the transformation process for each step.

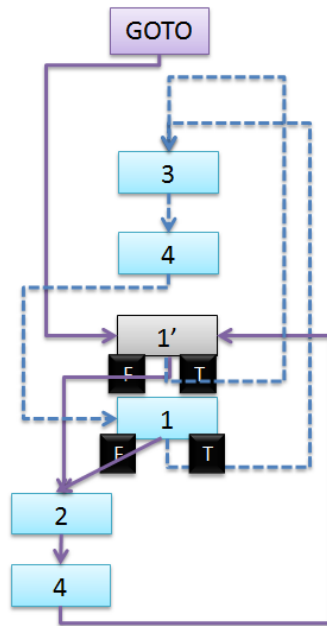


Figure 4.10: Loop after applying intersection loop obfuscation.

the moved loop.

To make it hard to reverse engineer the contents of transfer table, before store the source address into it, we use a hash function and store the hashed value into transfer table. This not only hides the value of the source address but also complicate the transfer table because source addresses do not appear in the obfuscated program directly. The concept is summarized in Figure 4.11.

After getting the corresponding target from transfer table, the next question is how to put the target into PC. The process is depicted in Figure 4.12.

Ideally, it contains two steps: (1) store the target to the local variable;

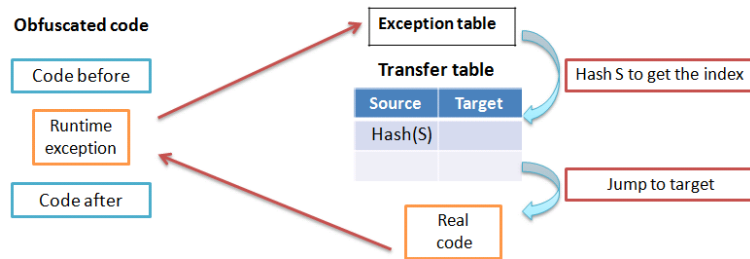


Figure 4.11: Concept of hashing source address.

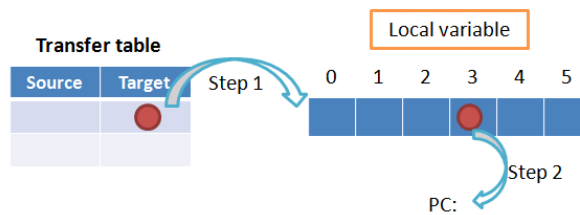


Figure 4.12: Process of transfer target to PC.

and (2) push the value in the local variable into PC. Unfortunately, neither of these two steps can be achieved because JVM does not supply such bytecodes that can store a reference type value (target's type is present as **reference** in JVM) into local variable and none of bytecodes can do indirect jump. So we choose switch to build our transfer table. Figure 4.13 shows the concept how a transfer table looks like. The drawback of using switch to implement our transfer table is that for each case in switch, the target is fixed. This means we can only play tricks on source address but not on target address to make attackers confused.

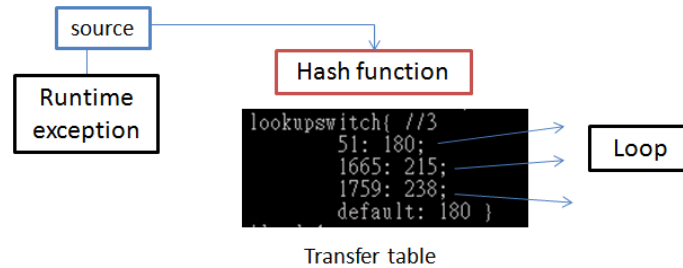


Figure 4.13: Use switch to build transfer table.

4.7 Local Variable Inconsistent

Local variable inconsistent problem is a verify error, it is occurred when java verifier verifies our obfuscated program. Let's explain this problem in Figure 4.14. Figure 4.14.(a) is the original program which contains three for loops. After our obfuscation, these three loops will be replaced by three runtime exceptions and a transfer table will contains the target information for each source. Figure 4.14.(b) depict the concept. When we run the obfuscated code, the java verifier will be raised to verify the whole program, when it encounters the first runtime exception, it jumps to the transfer table. Since the transfer table is made by switch, java verifier won't have any idea what the correct case is, so it will traverse every cases one by one. Then when it verifies Figure 4.14.(2), a local variable inconsistent problem will be occurred because local variable `j` has not been declared.

To solve this problem, we can simply copy the declaration of each local variable that will be used in current method and put it on top of the method. In this way, when Java verifier verifies the obfuscated code, all the local vari-

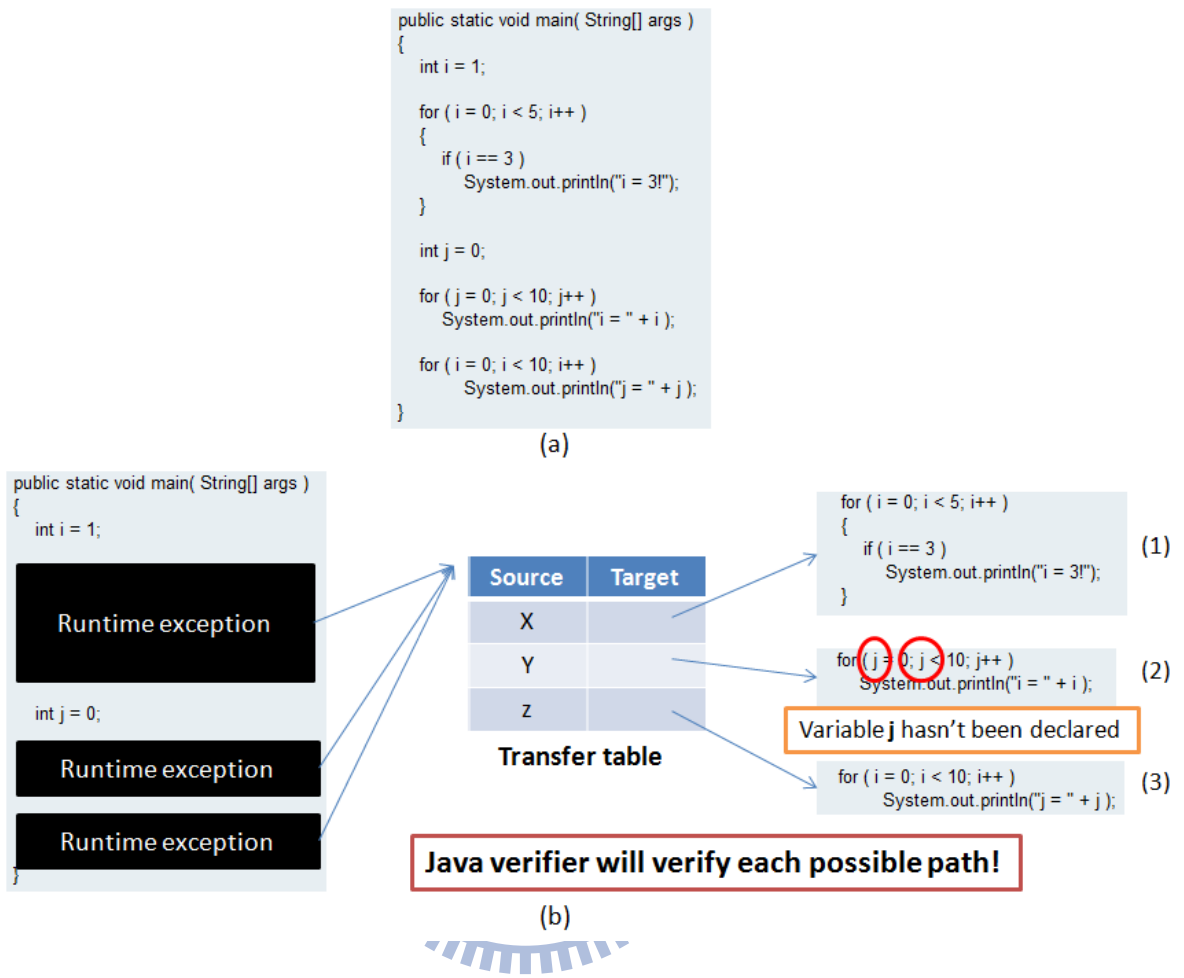


Figure 4.14: How local variable inconsistent occur.

ables are initialized in advance before executing the first runtime exception we made, so local variable inconsistent problem can be avoided.

Chapter 5

Experiment

We evaluated our efficiency of our approach using seven programs from the SPCEJVM2008 benchmark suite. Our experiment were run under windows 7 on Intel Core 2 Duo CPU E7400, with 4GB RAM. We used the Sun Java HotSpot Client VM, 16.3-b01 mixed mode with the default Just-In-Time compiler turned on. Each evaluated on the program includes iteration and warmup. An iteration goes on for a certain duration, by default 240 seconds. During this time the program will be called several times, one by one as soon as previous program completed. It will never abort a program, but wait until a program is completed for stopping. The first iteration is a warmup iteration, run for 120 seconds by default. The result of the warmup iteration is not included in the benchmark result. The result for each profile is a score on each workload.

5.1 Performance of the Obfuscated Program

The following data was obtained by applying our obfuscation to SPECJVM2008 benchmark programs. Seven SPEC programs are used in this experiment. Figure 5.1 shows the result. We tested each benchmark program by obfuscating 0%, 10%, 50% and 80% of the program. As can be seen in Figure 5.1, derby decrease a lot.

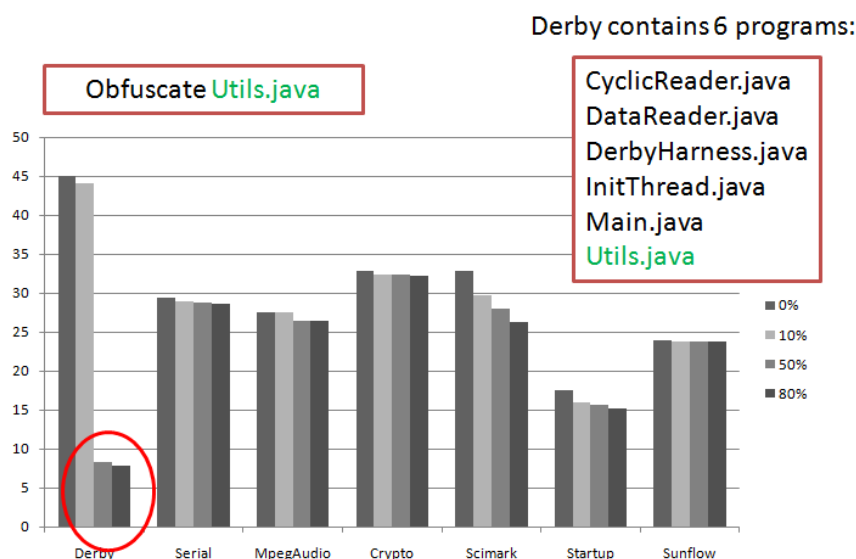


Figure 5.1: Performance on SpecJVM2008.

Derby contains six programs. After tracing it, we found that there are four programs will call `Utils.java` when they execute. Figure 5.2 depict the relationship among the four programs and `Utils.java`. After obfuscating, loops in `Utils.java` become runtime exceptions. Since exception increase lots of overhead, it decrease performance a lot when the four programs keep calling `Utils.java` at runtime. This is the main reason why derby got bad

performance after obfuscation.

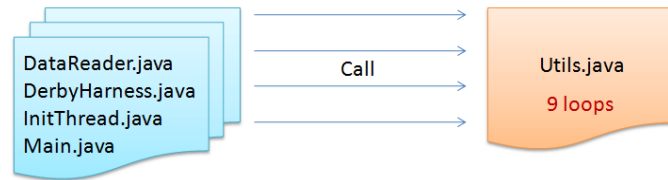


Figure 5.2: Relationship among Utils.java and other programs.

Although Utils.java is hot, it is not necessarily to be importance code. This means weather we obfuscate Utils.java or not, an attacker won't be able to get important algorithms from the obfuscated program, so we choose not to obfuscate it. Figure 5.3 shows the result. As can be seen, all the decrease on performance is less then 21%.

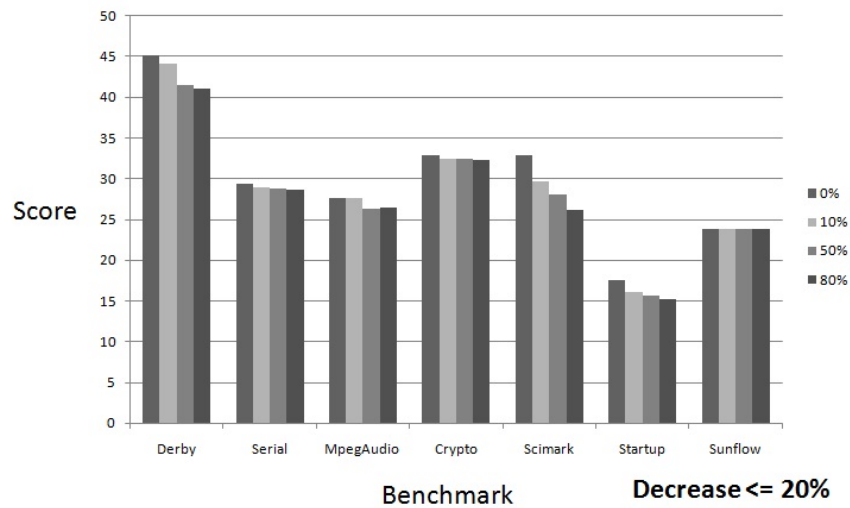


Figure 5.3: Performance on SpecJVM2008 without obfuscate Utils.java.

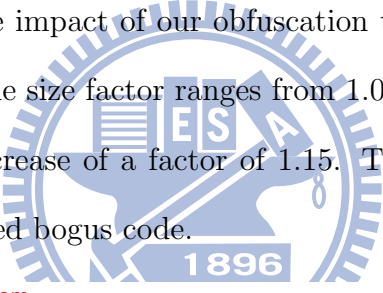
5.2 Code Size

Since dynamic class loading is one of the important features of JVM, so to keep increased code size down is a major issue for obfuscation.

To decrease the impact on code size after obfuscation, we remove the `LineNumberTable` and `LocalVariableTable` from class file after obfuscation. These two attributes are optional and may be used by debuggers to get more debugging information from class file. So removing these two attributes not only can save much space but also make debuggers hard to work.

Figure 5.4 shows the impact of our obfuscation techniques on code size. We can see that the code size factor ranges from 1.07 (Crypto) to 1.28 (Scimark), with a mean increase of a factor of 1.15. The increase of code size mainly comes from added bogus code.

Obfuscate whole program.

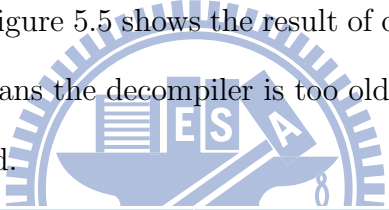


Benchmark	Original	Obfuscated	Increase
Derby	37461	43087	15.0%
Serial	39547	44102	11.5%
MpegAudio	4642	5455	17.5%
Crypto	17080	18345	7.4%
Scimark	37660	48484	28.7%
Startup	6584	7137	8.4%
Sunflow	3114	3670	17.8%

Figure 5.4: Code size increased after obfuscation.

5.3 Decompiler

To proof our obfuscation technique can resist decompiler, we use 9 decompilers and one deobfuscator to test the obfuscated code. The way how we test decompilers contains four steps: (1) use javac to compile the java source code to bytecode; (2) apply our obfuscation technique on the bytecode; (3) use decompiler to decompile the obfuscated bytecode; and (4) use javac to compile the produced source code again. If javac shows compile error, we say that the tested decompiler is fail and mark a X on the blank of the corresponding decompiler. Figure 5.5 shows the result of our testing. Blanks with "Version mismatch" means the decompiler is too old to support new version of class file that we used.



Decompiler	Decompilable
Jad	X
Cavaj	X
DJ Java Decompiler	X
JD-GUI	X
Mocha	Version mismatch
JreversePro	Version mismatch
JODE	X
Dacafe	Version mismatch
JCavaj	X

Deobfuscator	Decompilable
JDO	X

Figure 5.5: Test result of decompilation obfuscated program.

5.4 Visualizing the Effects of Obfuscation

To give an intuitive view on the effect of our obfuscation techniques, we use `c1Visualizer` [2] to visualize the control flow graph of both the original program and obfuscated one. For easy understanding, we use a small program that contains three for loops.

Before visualize the obfuscated program, we apply Java Deobfuscator (JDO) [3] on it. As can be seen in Figure 5.6, JDO could not eliminate most of the faked control flow edges. Hence, we can see a dramatically change to the obfuscated code.

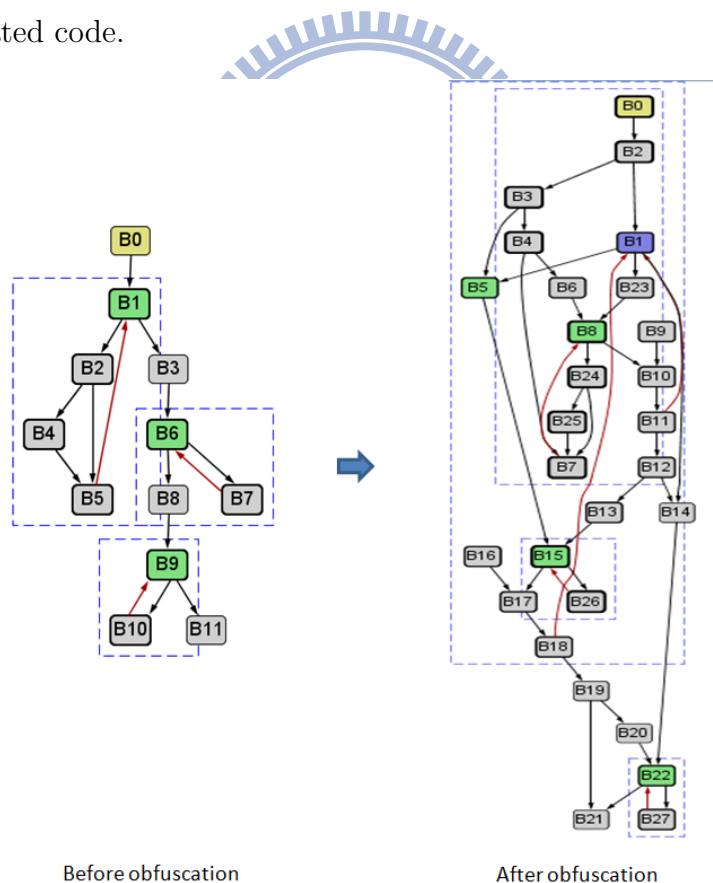


Figure 5.6: Intuitive view on the effect of our obfuscation method.

Chapter 6

Conclusion

The problem of protecting software from attackers is an important issue. To protect intellectual property, obfuscation is one of the easiest and efficient way to achieve this goal.

In this paper, we has described a new approach to obfuscating java bytecode and evaluated its effectiveness and code size on programs in SPECJVM2008 benchmark suite. In our framework, we replace loops by some bytecode that cause runtime exception, then use a transfer table to response the mapping of runtime exception and loop, and insert bogus code to further confuses decompilers.

The experiment results show that the average effect on performance is less than 21%, and the mean increase of code size is 1.15X. We also use 9 decompilers and 1 deobfuscator to test the strength of our obfuscation method. The experiment shows that these tools can not reverse the obfuscated code

to source code.

Since we propose a obfuscation method that can be done efficiently with moderate increase in execution time and code size, our method can be combine with more other obfuscation methods to obtain better obfuscation strength.



Bibliography

- [1] <http://jakarta.apache.org/bcel/>.
- [2] <https://c1visualizer.dev.java.net/>.
- [3] <http://www.softpedia.com/get/programming/debuggers-decompilers-dissassemblers/java-deobfuscator.shtml>.
- [4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *ACM SIGPLAN Notices*, pages 1–12, 2000.
- [5] Jien-Tsai Chan and Wu Yang. Advanced obfuscation techniques for java bytecode. *J. Syst. Softw.*, 71(1-2):1–10, 2004.
- [6] Jien-Tsai Chan, Wu Yang, and Jing-Wei Huang. Traps in java. *J. Syst. Softw.*, 72(1):33–47, 2004.
- [7] Haibo Chen, Liwei Yuan, Xi Wu, Binyu Zang, Bo Huang, and Pen chung Yew. Control flow obfuscation with information flow tracking.

- In *Proc. MICRO-42 Microarchitecture 42nd Annual IEEE/ACM Int. Symp*, pages 391–400, 2009.
- [8] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. 28(8):735–746, 2002.
- [9] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, July 1997.
- [10] Daniel Dolz and Gerardo Parra. Using exception handling to build opaque predicates in intermediate code obfuscation techniques. *JCS&T*, 8(2), 2008.
- [11] Douglas Low. Java control flow obfuscation. Technical report, 1998.
- [12] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. A study of exception handling and its dynamic optimization in java. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 83–95, New York, NY, USA, 2001. ACM.
- [13] Igor V. Popov, Saumya K. Debray, and Gregory R. Andrews. Binary obfuscation using signals. In *SS'07: Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.
- [14] Chenxi Wang, J. Davidson, J. Hill, and J. Knight. Protection of software-based survivability mechanisms. In *Proc. [Organically Assured*

and Survivable Information Systems] Foundations of Intrusion Tolerant Systems, pages 273–282, 2003.

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

