KTH Computer Science
and Communication

Obfuscate Java bytecode: an evaluation of obfuscating transformations using JBCO

EMILIA HILLERT

Master's Thesis at NADA
Supervisor: Douglas Wikström
Examiner: Johan Håstad

May 2014

Abstract

Today, software is one of the most complex and intriguing technologies around us. Software development companies must be able to protect their products in order to stay strong against its competitors and against other threats. One solution to this problem is *code obfuscation*. This master's thesis investigates how to protect Java source code against *reverse engineering*. Three obfuscating transformations was successfully implemented and evaluated within a specific obfuscator named JBCO. The conclusion is that in many cases, code obfuscation provides a higher level of obscurity to a program, which makes code obfuscation a good supplement to complicate the reverse engineering process.

Referat

Obfuskera Java bytekod: en utvärdering av obfuskeringstransformationer med JBCO

För att ett företag ska vara konkurrenskraftigt krävs det ofta att företaget skyddar sina produkter. Det kan vara förödande för ett företag om en konkurrent får tillgång till deras produkthemligheter och leda till företagets undergång. Detta gäller framförallt företag som utvecklar mjukvara. En lösning på detta problem är *kodobfuskering*. Detta examensarbete undersöker hur man skyddar Java källkod mot *dekompilatorer*. Tre stycken obfuskeringstransformationer implementerades och evaluerades i obfuskeraren JBCO. Slutsatsen är att kodobfuskering i många fall leder till att ett program blir svårare att tyda för en människa. Detta gör obfuskering till ett bra komplement för att försvåra processen att dekompilera ett program.

Acknowledgements

Stefan Larsson, Bitsec, for giving me the opportunity to carry out my master's thesis at Bitsec.

Jonas Renold, Bitsec, for your guidance and feedback.

Douglas Wikström, KTH, for your guidance and input.

Johan Håstad, KTH, for your help and feedback on the report.

Finally, I want to thank my family and friends for their help and support.

Contents

Contents

Part I

Introduction

Chapter 1

Background

1.1 Problem

In order to stay strong against its competitors, a company must be able to protect its business secrets. Business secrets can be anything from design philosophy and ideas to complex solutions. The consequences could be devastating for the company if any of those secrets were revealed to the rest of the world.

Today's software is one of the most complex and intriguing technologies around us. As a company that is developing software, it is important to protect the products against reverse engineering. Reverse engineering in this case means reconstructing source code from a compiled program. To protect the software against reverse engineering, a code obfuscator can be used. Code obfuscation is the technique of creating code that is:

- difficult for humans to read and understand, and
- hard to reverse engineer.

It is important to remember that an obfuscator does not guarantee full protection; it should be seen as a supplement to the code that makes the reverse engineering process difficult.

Some programs are more vulnerable than others. Programs written in the programming language Java is exposed due to the Java bytecode, which normally contains enough information to permit type checking. Therefore it is particularly important to protect programs written in Java.

1.1.1 Goal

The purpose of this master's thesis is to investigate how to protect Java code against theft. More specifically, how to protect the code against reverse engineering (de-

compilation). This is a known problem, a number of tools already exist where the main purpose is to make it harder to decompile Java code. This thesis focuses on the tool JBCO (Java ByteCode Obfuscator). Since it is an open source project and an automatic obfuscation tool, it is easy to further develop and to use. The thesis's main contribution is to implement new obfuscating transformations within the JBCO to contribute to the ongoing research of code obfuscation.

1.1.2 Thesis delimitations

Several improvements could be made to the JBCO tool, but this thesis only focuses on the three different Obfuscating Transformations listed below:

OT1 Outlining conditions.

OT2 Array restructuring.

OT3 Variable restructuring.

1.2 Thesis outline

Chapter 2 serves as an introduction and provides the reader with necessary information for this thesis. The reader is first introduced to code obfuscation and the Java language. This is followed by more advanced background facts such as Java bytecode obfuscators, JBCO, code obfuscation methods and decompilers. Lastley the analysis metrics is presented. In chapter 3 the methods and the approach for the thesis are presented.

In chapter 4, the implementation of each obfuscating transformation is presented followed by the result in chapter 5.

The thesis ends with a discussion in chapter 6 followed by conclusions drawn in chapter 7 based on the result and lastly future research in chapter 8.

Chapter 2

Theory

This chapter presents the necessary theory and concepts to understand this thesis.

2.1 Code obfuscation

Code obfuscation is the technique of creating code that is difficult for humans to read and to reverse engineer. To create this type of obscured code, an obfuscator is used. An obfuscator is a program that obfuscates an input program and outputs an obfuscated program with the same semantics as the input, see figure 2.1.

Figure 2.1: An obfuscator takes a program and obfuscates the code with different obfuscating transformations and outputs a transformed program. The transformed program is semantically equivalent to the original program and it is harder to reverse engineer.

To obscure a program, one or more obfuscating transformations must be applied on the code. Collberg et al. described the notation and definition of obfuscating transformations as follows:

Definition 1 (Obfuscating transformation)
Let $P \xrightarrow{\tau} P'$ be a transformation of a source program P into a target program P'. $P \xrightarrow{\tau} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behaviour*. More precisely, in order for $P \xrightarrow{\tau} P'$ to be a legal obfuscating transformation the following conditions must hold:

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

The term *observable behaviour* is defined loosely as "behaviour as experienced by the user" by Collberg et al. This means that P' may have side effects that P does not, as long as these side effects are not experienced by the user. Side effects such as creating files, sending messages over the Internet etc. It is not required that P and P' are equally efficient, P' is probably slower and may use more memory than P [3].

2.1.1 History

Early obfuscation attempts were made by Cohen [2], which involved rewriting machine-level instructions. He attempted obfuscations such as replacement of instructions, small sequences of instructions, instruction reordering, adding or removing arbitrary jumps and outlining methods. Later on, Collberg et al. presented a more theoretical approach to obfuscations. They defined the term obfuscating transformation and formed a terminology to describe an obfuscation in terms of affect and quality as follows:

Potency: the level of obscurity a specific transformation gives.

Resilience: measures how well an obfuscation holds up against reverse engineering attacks.

Cost: the performance and size penalties incurred by the obfuscation.

Stealth: how hard it is to detect whether the transformation has been applied to a program.

They suggested a number of obfuscating transformations such as false refactoring and method cloning, and array restructuring through splitting or merging. Some years later, Collberg and Thomborson [3] suggested control-flow obfuscation. They used opaque predicates to introduce dead code (dead branches).

2.2 The Java language

The Java language is a high-level and architecture-neutral programming language which was released by Sun Microsystems in 1995. Many of Java's object-oriented features are influenced by the programming language C++. This in turn is a descendant of the programming language C, and it is from C Java inherits its syntax.

Historically, the original impetus for Java was not the Internet. The primary motivation was the need for a platform-independent (architecture-neutral) language that could be used to create software which could be imbedded in various consumer electronic devices. The trouble with C and C++ was that they were designed to be compiled for a specific target. Around the same time when the details of Java were worked out, the World Wide Web took shape. The Internet helped catapult Java into becoming one of the more popular programming languages due to the forward-thinking design approaches, which were unique to the Internet medium [21].

2.2.1 Primary characteristics of Java

The primary characteristics of Java are presented below.

Object-oriented language

Java is an object-oriented language and uses the principles of abstraction, encapsulation and inheritance. Objects and classes are fundamental parts of Java, and a class contains both data (attributes) and executable code (methods).

Multithreading

Java provides a multithreading mechanism, which means that processes can execute simultaneously within an application.

Dynamic linking

In Java, the existence of libraries is verified during compilation, and code is loaded from these libraries when the program is executed. This mechanism is called dynamic loading and means that the size of executables decreases, which makes it possible to optimize the loading of libraries.

Garbage collector

Java has a garbage collector, which is a mechanism that clears memory of all objects that are no longer being used.

Security

The Java virtual machine performs very strict verifications of Java code before it is executed, which means that code cannot bypass the protection mechanisms imposed by the language and the code cannot try to define pointers to directly access memory [20].

Portability

Each kind of machine (e.g. computer) has its own instruction set. It is generally true that a program that runs on one machine will not run on another. The services provided by the operating system, which each system describes in its own unique way, will cause a compatibility and portability problem.

Java gets around this problem by inserting its virtual machine between the application and the environment (the machine and operation system). This results in Virtual Machine code (e.g. Java bytecode) instead of machine code. The main difference between these two is that machine code is specific to the computer system in use and bytecode is platform independent [24, 22].

2.2.2 Java Virtual Machine and Bytecode

Bytecode is a highly optimized set of instructions (stored in a class file) designed to be executed by the Java virtual machine (JVM), which is the Java run-time system. The JVM is a platform-independent execution environment that converts bytecode into machine code and executes it.

When a JVM loads a class file, it gets one stream of bytecodes for each method in the class, these streams are stored in the method area of the JVM. The bytecode for a method is executed when that method is invoked in the program, and can be executed by interpretation, just-in-time compiling, etc.

A method's bytecode stream is a sequence of instructions for the JVM. Each instruction consists of a one-byte opcode (indicates the action to take) followed by zero or more operands. Each type of opcode has a mnemonic, which means that streams of Java bytecode can be represented in the style of assembly language followed by any operand values.

Instructions fall into broad groups which can be seen in table 2.1.

Group	**Example**
Load and store	iload_0, lstore
Arithmetic and logic	iadd, fcmpl
Type conversion	f2b, i2d
Object creation and manipulation	new, putfield
Operand stack management	swap, dup2
Control transfer	ifeq, goto

Method invocation and return	invokespecial, areturn

Table 2.1: Instructions.

Each bytecode opcode is one byte in length and each byte has 256 potential values. The JVM supports seven primitive data types. These can be seen in table 2.2.

Prefix/Suffix	**Operand Type**	**Example**
i	integer	iadd
l	long	ladd
s	short	sadd
b	byte	badd
c	character	cadd
f	float	fadd
d	double	dadd
z	boolean	zadd
a	reference	aadd

Table 2.2: Primitive data types.

These types appear as operands in bytecode streams. All types that occupy more than 1 byte are stored in big-endian order in the bytecode stream, which means higher-order bytes precede lower-order bytes.

Many opcodes push constants onto the stack. There are three different ways to indicate which constant value is to be pushed:

- The constant value is implicit in the opcode itself: when the constant value is implicit in the opcode, it indicates a type and a constant value to push. An example is the iconst_1 opcode which tells the JVM to push integer value 1. Another example is the opcode iconst_m1 which pushes integer value -1, and the opcode aconst_null which pushes a `null` object reference onto the stack.

- The constant value follows the opcode in the bytecode stream as an operand: two opcodes indicate that the constant to be pushed onto the stack has an operand that immediately follows the opcode.

- The constant value is taken from the constant pool: three opcodes push constants from the constant pool. Constants that are stored in the constant pool are associated with a class, such as final variable values. Opcodes that push constants from the constants pool have operands that indicate which constant to push by specifying a constant pool index. This index follows the opcode in the bytecode stream.

All compilation in the JVM are centered on the stack. Since the JVM has no registers for storing arbitrary values, everything has to be pushed onto the stack before it can be used in a calculation. Bytecode instructions primarily operate on the stack [23]. A stack stores stack frames, and a frame is created each time a method is invoked. Each frame consists of three sections: an array of local variables, the operand stack, and a reference to the runtime constant pool of the class of the current method, see figure 2.2. The array of local variables is determined at compile time and is dependent on the number and size of local variables and formal method parameters.

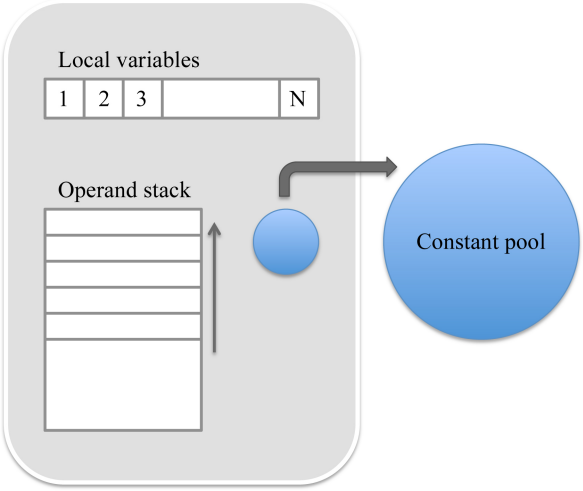

Figure 2.2: A frame containing an array of local variables, the operand stack and a reference to the runtime constant pool of the class of the current method.

There exists a corresponding opcode that pops the top of the stack back onto the local variable for each opcode that pushes a local variable onto the stack [8].

2.3 Java bytecode obfuscators

Several Java bytecode obfuscators exists that are commercial, free or open-source. In the list below, some of the most popular Java bytecode obfuscators are presented.

ProGuard A free Java class file shrinker, optimizer, and obfuscator. ProGuard finds and removes unused classes, fields, methods and attributes. It optimizes bytecode and removes unused instructions and renames the remaining classes, fields and methods by using meaningless names. The Jar files becomes smaller and harder to reverse engineer. The latest version was released March, 2013 [13].

Zelix KlassMaster A Java bytecode obfuscator that modifies Java class files. It changes class, field and method names to meaningless short strings. The obfuscator also performs flow obfuscations, exception obfuscation and Java string encryption. The latest version was released June 23, 2013 [14].

JBCO The Java ByteCode Obfuscator is built on top of the Soot framework and operates on Java class files. JBCO can perform three different types of obfuscations: operator level obfuscations, obfuscating program structure, and exploiting the design gap. The latest version was released January 22, 2012 [16].

yGuard A free Java bytecode obfuscator and shrinker. Renames packages, classes, fields methods according to a selectable name mapping scheme. The latest version was released October 12, 2012 [26].

2.4 JBCO: The Java ByteCode Obfuscator

This thesis only focuses on one obfuscator where new obfuscating transformations were implemented: The Java ByteCode Obfuscator (JBCO). JBCO is a Java bytecode obfuscator built on top of the framework Soot and is developed by Batchelder. It is written in Java, and JBCO is a free open source project. JBCO transforms code to be more complex and esoteric, to make it harder to decompile. To get a better understanding of the JBCO, the framework Soot is presented first.

2.4.1 Soot: A Java Optimization Framework

Soot is a Java optimization framework from the Sable Research Group at McGill University in Quebec, Canada. Soot provides four intermediate representations for analysing and transforming Java bytecode. These four provide different levels of abstraction on the represented code and are targeted for different uses. During execution, Soot transforms Java source code or bytecode into one of the following representations: Baf, Jimple, Grimp or Shimple [5].

Baf is a streamlined stack-based representation of bytecode. It is used to inspect Java bytecode as a stack code, but abstracts away the constant pool and the type dependent variations of instructions into a single instruction. In contrast

to bytecode, which has several instructions for adding primitive data types like `int`, `long` etc., Baf only has one instruction for adding primitive data. This representation is useful for bytecode analyses, optimizations and transformations. Listing 2.1 shows the example method `faculty` in Java source code, and listing 2.2 shows the same method in Baf.

```
1      public int faculty(int n) {
2          int mult = 1;
3          for(int i = 1; i <= n; i++) {
4              mult = mult*i;
5          }
6          return mult;
7      }
```

Listing 2.1: Example method `faculty` in Java source code.

```
1      public int faculty(int)
2      {
3          word r0, i0, b2, i3;
4
5          r0 := @this: Math;
6          i0 := @parameter0: int;
7          push 1;
8          store.b r0;
9          push 1;
10         store.b b2;
11         goto label1;
12
13      label0:
14         load.b r0;
15         load.b b2;
16         mul.b;
17         store.i i3;
18         load.b b2;
19         push 1;
20         add.b;
21         store.i i3;
22
23      label1:
24         load.b b2;
25         load.i i0;
26         ifcmple.b label0;
27
```

```
28            load.b r0;
29            return.b;
30        }
```

Listing 2.2: Example method `faculty` in Baf form. `Math` is the name of the class and the single characters following the dots are the type of the instruction. The instruction `store.i` stores an int.

Jimple is a typed 3-address intermediate representation and a very good foundation for most analyses since it does not need the explicit control flow. The translation from bytecode to Jimple is performed using a naïve translation from bytecode to untyped Jimple. This is made by introducing new local variables for implicit stack locations and using subroutine elimination to remove `jsr` instructions. For local variables the types are inferred in the untyped Jimple and then added. The most important and special part about the transformation to Jimple is that the transformation is a linearization of expressions, so statements only reference at most 3 local variables or constants. Jimple only has 15 different statements, compared to Java bytecode which has more than 200 different instructions. The Jimple statements can be seen in the table 2.3.

Statement	**Type of statement**
NopStmt	Core statement
IdentityStmt	Core statement
AssignStmt	Core statement
IfStmt	Intraprocedural control-flow
GotoStmt	Intraprocedural control-flow
TableSwitchStmt	Intraprocedural control-flow
LookupSwitchStmt	Intraprocedural control-flow
InvokeStmt	Interprocedural control-flow
ReturnStmt	Interprocedural control-flow
ReturnVoidStmt	Interprocedural control-flow
EnterMonitorStmt	Monitor statement
ExitMonitorStmt	Monitor statement
ThrowsStmt	-
BreakpointStmt	-
RetStmt	-

Table 2.3: All Jimple statements.

The Jimple local variables start with a dollar sign ($) and represent stack positions (not local variables in the original program). In listing 2.3 the method `faculty` is showed in Jimple.

```
1     public int faculty(int)
2     {
3         Math r0;
4         int i0, i3, i4;
5         byte b1, b2;
6
7         r0 := @this: Math;
8         i0 := @parameter0: int;
9         b1 = 1;
10        b2 = 1;
11        goto label1;
12
13     label0:
14        i3 = b1 * b2;
15        i4 = b2 + 1;
16
17     label1:
18        if b2 <= i0 goto label0;
19
20        return b1;
21    }
```

Listing 2.3: Example method `faculty` in Jimple form. `Math` is the name of the class.

These two representations are used by the JBCO to analyse and perform the transformations. The two representations presented below are not used by the JBCO, and therefore they are not explained in detail.

Grimp is an aggregated version of Jimple suitable for decompilation and code inspection.

Shimple is a Static Single Assignment-form version of the Jimple representation and guarantees that each local variable has a single static point definition, which simplifies analyses.

2.4.2 The functionality of JBCO

JBCO operates on Java class files or source code and produces obfuscated Baf, Jasmin, or class files [1]. Jasmin is an assembler for the JVM and it takes ASCII descriptions (JVM instruction set syntax) of Java classes and converts them into binary Java class files [17]. The JBCO itself is just a number of Jimple and Baf transformations and each module falls under one of three categories:

Information Aggregator: data from the program such as identifier names, constant usage or local variables to type pairings are collected for other transformations.

Code Analyses: build new forms of information about the code such as control-flow graphs, stack height and type data or use-define chains, which are used to identify where in the program transformations can be applied.

Instrumenters: the algorithms within JBCO that modify the code, which means adding obfuscations or shuffling the code to obscure meaning. There are two different types of transformations: those that operate on the program as a whole and those that operate on one method at a time.

JBCO can be used in two different ways; as a command-line tool or via a graphical user interface. A user can choose which transformation or transformations to use and each transformation has a weight of 0-9, where 0 turns it off and 9 corresponds to applying it everywhere possible. Another mechanism is to limit the obfuscations to specific regions of a program by using regular expressions to specify classes, fields, or methods [1].

2.5 Code obfuscation methods

This section focuses on the code obfuscation methods within JBCO. These obfuscations can be divided into three groups: operator level obfuscations, obfuscating program structure, and exploiting the design gap.

2.5.1 Operator level obfuscations

Operator level obfuscation simply reworks the low-level program logic and does not change the design structure of the program or the control flow of method execution. These kinds of obfuscations are not built to confuse a decompiler, they are made to confuse a human.

Renaming identifiers: classes, methods and fields

Replacing class, field and method names in bytecode can avoid revealing important information, since names of functions often reveals the purpose of the function. Identifiers are renamed with the help of randomly generated sequences, or by taking names from other methods or fields within the program. In JBCO the randomly generated sequences consists of characters which look alike and are difficult to distinguish. These characters are:

[S, 5, $] : a uppercase letter S, a digit five and a dollar sign.

[l, 1, I] : a lowercase letter L, a digit one, a uppercase letter i.

[_] : an underscore.

An example of a renaming of identifiers transformation can be seen in figure 2.3, where both before and after the transformation is listed.

```
1 public void helloWorld(){
2     System.out.print(hello);
3     HelloWorldAgain();
4 }
```

```
1 public void S55$S(){
2     System.out.print(ll1);
3     lI1I1();
4 }
```

Figure 2.3: A Java source code snippet before (listing to the left) and after (listing to the right) a renaming of identifiers' transformation.

Embedding constant values as fields

Constants are often used, and are separately stored in bytecode. Each class file stores constant data in a constant pool and each constant is accessed by its index within the pool. To obfuscate these constants, each constant is moved into a static field and then the references to the constant are changed into references to the field. An example of before and after an 'Embedding constant values as fields' transformation, can be seen in figure 2.4.

```
try {
    i=new FileInputStream(n);
}catch (IOException e){
    System.err.print
    ("File opening failed.");
}
```

```
try {
    i=new FileInputStream(n);
} catch (IOException e){
    System.err.print
        (ObjectA.field1);
}
```

Figure 2.4: A Java source code snippet before (listing to the left) and after (listing to the right) the embedding constant values as fields transformation.

Packing local variables into bitfields

It is possible to combine local variables with primitive types and pack them together into one variable, which has more bits. For each local variable the range of bits are randomly chosen.

Example:
An integer is represented by 32 bits and it is packed into a long between its 9th and 41rd bits.

Converting arithmetic expressions to bit-shifting operations

Complex operations such as multiplication or division can be obscured by converting them into sequences of cheaper operations. JBCO looks for expressions in the form of $v \cdot C$ or v/C, where v is a variable and C is a constant. The largest integer i is extracted from C, where $i < C$, and $i = 2^s$, where $s = floor(log_2(v))$. The remainder is computed $r = v - i$. If s is in the range of $-128 \ldots 127$, the original is converted to $v << s + (v \cdot r)$, and the expression $v \cdot r$ can be further decomposed. To further obscure, a random multiple m is chosen and an equivalent shift value $s' = (byte)(s + (m \cdot 32))$ is computed.

Example:
Before obfuscation: $v \cdot 195$
After obfuscation: first phase: $(v << 7) + (v << 6) + (v << 1) + v$, in the second phase the three shift values are further obfuscated to:
$(v << 39) + (v << 38) + (v << -95) + v$.

2.5.2 Obfuscating program structure

Program structure can be divided into two categories: high-level object-oriented design (moving methods, creating new classes) and the low-level control flow (con-

fusing the control flow of a method). The obfuscation transformations presented in this section operates according to one of these two categories.

Adding dead-code switch statements

This obfuscation adds edges to the control flow graph by inserting a dead switch to complicate flow analysis. The switch construct in Java bytecode offers a control flow obfuscation tool. It is the only way to create control flow graphs that has a node whose successor count is greater than two. The switch is wrapped in an opaque predicate to ensure that it is not executed. All bytecode instructions with a stack height of zero are safe jump targets for cases in the switch. JBCO randomly selects some as targets for the cases switch, which increases the couplings and the complexity of a method.

Finding and reusing duplicate sequences

By finding duplications in a method, and replacing them with a single switched instance, the size of the method can potentially be reduced. There are a number of rules which define whether a sequence is a proper duplicate.

Replacing if(non)null instructions with Try-Catch blocks

This obfuscation exploits two facts of Java. One is that invoking an instance method from a `null` object will always result in a NullPointerException being thrown. The other is that two `null` objects will always be considered *equal* by the `ifacmpeq` instruction, and a non-null object will always be considered *not equal* to a `null` object. Every `ifnull` and `ifnonnull` instruction in a method is considered for this transformation. The `ifnull` instruction being transformed is removed and either replaced with a call to `toString` or an `ifacmpeq` instruction comparing the original object to a null reference. An example of before and after the replacing of the if(non)null instructions with Try-Catch blocks transformation can be seen in figure 2.5.

```
1 if(arg == null) {
2     System.out.print
3         ("Failed.");
4     return;
5 }
```

```
1 try{
2     arg.equals(null);
3 }catch(NullPointerException e){
4     System.out.print("Failed.");
5     return;
6 }
```

Figure 2.5: A Java source code snippet before (listing to the left) and after (listing to the right) the replacing of if(non)null instructions with Try-Catch blocks transformation.

Building API buffer methods

Methods that call direct execution into standard Java libraries cannot be renamed. Instead, the names of the Java library methods can be obscured by indirecting all Java library method calls through intermediate methods with nonsensical identifiers.

Building library buffer classes

This obfuscation attempts to create confusion by adding extra layers, if a class extends a library class. If a class C extends a library class L, then a buffer class B is created and inserted as a child of L and a parent of C. Since B is never used directly in the program, methods over-ridden in C can be defined as nonsense methods in B, which adds confusion.

2.5.3 Exploiting the design gap

There are certain gaps between what can be represented in Java source code and what can be represented in bytecode. The obfuscation transformations presented in this section attempt to exploit these gaps.

Converting branches to `jsr` instructions

The `jsr` bytecode (Jump to SubRoutine) is analogous to the `goto` bytecode, except that it pushes a return address on the stack. The return address is normally stored in a register after a `jsr` jump. When the subroutine is complete, the `ret` bytecode is used to return. This Obfuscation replaces `if` and `goto` targets with `jsr` instructions. Each old jump target is prepended by a `pop` in order to throw the return address which is pushed onto the stack. A `goto` is inserted after a jump target if its predecessor in the instruction sequence falls through, which causes a jump directly to the old target. The transformation changes almost all control flow to be `jsr` based.

Reordering `load` instructions above `if` instructions

In situations where a local variable is used directly following both paths of a `if` (the first instruction loads the variable on to the stack), the transformation can be used. The obfuscation moves the `load` instruction above the `if`, removing its clones along both branches.

Disobeying constructor conventions

In the Java source the first statement must be a super call, which means that class constructors always must call either an alternate constructor of the same class or their parent class constructor as the first directive. If no one is specified, *javac* explicitly adds a call to the parent. In bytecode this is not required. This obfuscation

exploits that, and randomly chooses between four approaches to achieve confusion. The four approaches are: wrapping the super call within a try block, taking advantage of classes which are children of java.lang.Throwable, inserting a `jsr` jump and a `pop` directly before the super constructor call, or adding new instructions before the super call.

Partially trapping switch statements

This obfuscation traps sequential sections of bytecode that are not necessarily sequential in Java source code. The Java construct allows only well-nested and structured uses of try-catch blocks, but the bytecode implementation is at a lower abstraction. An example is the switch construct. In Java source, the switch encapsulates different blocks of code as targets of the switch. In bytecode there is nothing explicit tying the `switch` instruction to the different code blocks. This means that if the `switch` is placed within a trap range along with only part of the code blocks which are associated as its targets, then there will be no way for a decompiler to output semantically equivalent code. This obfuscation is limited to those switch constructs which are not already trapped.

Combining Try blocks with their Catch blocks

This obfuscation combines a try-catch block such that both the beginning of the try block and the beginning of the catch block are the same instructions. Try-catch blocks can only be presented in one way in Java source code, with a try block directly followed by one or more catch blocks associated with it. This rule is not applied to bytecode. Try blocks can protect the same code that is used to handle the exceptions it throws, or one of its catch blocks can appear above it in the instruction sequence. To obfuscate, the first unit of the try block prepends with an `if` that branches to the try code or to the catch code depending on an control flow flag.

Indirecting `if` instructions

This obfuscation exploit the fact that *javac* always produces predictable try blocks by indirecting `if` branching through `goto` instructions which are within a certain try block. Since a try block protects gotos, the gotos cannot be removed unless the code can be statically shown to never raise an exception.

`Goto` instructions augmentation

Explicit `goto` statements exist in bytecode, but are not allowed in Java source. This obfuscation splits a method randomly into two sequential parts, then reorders these two parts and inserts two `goto` instructions [10, 1].

2.6 Reverse engineering

The concept of reverse engineering is the process of extracting the knowledge or design from anything made by man. It is often conducted to obtain missing knowledge, ideas and design philosophy when that information is unavailable. There can be several reasons why the technique is used. The information could be owned by someone who is not willing to share it, or perhaps the information has been lost or destroyed. The process to extract desirable information often involves dissecting the product to uncover the secrets of the design. Often the product is examined under a microscope, or is even disassembled in order to examine the parts. The secrets that are retrieved are usually used to make similar or improved products.

In the software world, reverse engineering is a known problem. Software reverse engineering is about understanding of what is happening 'under the hood' of a program. The information gained can be used for security-related or software development-related applications. Reverse engineering is heavily used in connection with malicious software; both malware developers and those developing the antidotes use it. Reverse engineering has been employed in encryption research, involving reversing an encryption product and evaluating the level of security the product provides. The reversing technique is also used when hackers use it to analyse and crack copy protection schemes.

2.6.1 Decompilers

In order to reverse engineer a program, a decompiler can be used. A decompiler takes an executable binary file and attempts to produce readable high-level language code from it. This is done by trying to reverse the compilation process and then hopefully retrieve the original source file or a file similar to it. The output might be structured somewhat differently because of the compiler optimizations. The architecture of a decompiler is similar to a compiler, but works in the reverse order. In a conventional compiler the front-end is the component that parses the source code. In a decompiler, the front-end decodes low-level language instructions and translates them into an intermediate representation. This representation is gradually improved by eliminating useless details and emphasizing valuable details. The improved intermediate representation is then used by the back end to produce a high-level language representation [4].

Decompile Java bytecode

Decompilation of Java bytecode involves transforming Java bytecode to Java source code. This is easier than decompilation of machine code. Java bytecode is easier to decompile since it normally contains enough information to permit type checking. What this means is that bytecode contains explicit abstractions for methods, vari-

ables and the type of each variable. To decompile Java bytecode requires analysis of most of the local variable types. It also requires to flat stack-based instructions and to structure loops and conditionals.

Several Java decompilers exists that are commercial, free or open-source. In this report only a few decompilers were tested, in order to analyse the result of obfuscated Java bytecode. The following decompilers were tested:

Jad Java Decompiler is a free tool for non-commercial use. It is a closed source program, currently unmaintained. Jad is written in the programming language C++ and provides a command-line user interface to extract source code from class files. There is also a graphical user interface: jadClipse, which is a plugin to the Eclipse IDE. The decompilation process is confidential. The latest update for Linux and Windows was in 2001, but in 2006 an OS X version was added [9].

Dava A decompiler which is part of the Soot Java Optimisation Framework from the Sable Research Group at McGill Univerity in Quebec, Canada. Dava is using pattern matching and information obtained through data flow control analyses. Dava is under constant development and the latest version of the Soot framework was released in January 22, 2012 [18].

2.7 Analysis metrics

In this section a qualitative analysis of Java obfuscation is presented, or more precisely, a qualitative method to evaluate how well obfuscation is performed.

2.7.1 Quality of Java obfuscation

Karnick et al. [12] propose a qualitative analysis of Java obfuscation. To evaluate the performance for an obfuscator, the overall performance is denoted as $S_{quality}$, the quality of obfuscation. $S_{quality}$ includes the factors: potency, resilience and cost (as described in section 2.1). In this context, the potency, S_{pot}, refers to how much obscurity needs to be added to the code in order to prevent humans from understanding it. The resilience, S_{res}, is a measure of how strong the program can hold up against reverse engineering attacks. These attacks can be defined as attempts to transform the code back to the original source code by decompilers. The cost, S_{cst}, refers to how much computational overhead is added to an obfuscated program compared to a non-obfuscated program. Both potency and resilience have a positive impact on $S_{quality}$, since they illustrate how well the transformation protects the

code. Resilience is given a higher weight than potency, due to the fact that cognitive ability of a computer program is far inferior to that of humans. Cost have a negative impact on $S_{quality}$. Measuring these factors with different weights will yield a normalized score for the quality. A score of zero means that no obscurity have been added to the program. A higher score gives a higher quality of obfuscation.

$$S_{quality} = 0.4 \cdot S_{pot} + 0.6 \cdot S_{res} - S_{cst}$$

Karnick et al. found that the weights 0.4 and 0.6 gave the most feasible result.

2.7.2 Measurement of potency

Measuring potency is difficult since the analysis is based on human cognitive ability. To make the analysis easier, the potency can be divided into smaller measurements. Karnick et al. breaks potency into the following four complexity areas: nesting complexity, control flow complexity, variable complexity and program length.

Nesting complexity

The nesting complexity measure the number of iterative loops at different hierarchical levels in a program. Hierarchical levels of loops are defined as: a $level_n$ loop is the loop within the body of a $level_{n-1}$ loop. The nesting is calculated as follows:

$$c_{nesting} = \sum_{i=1}^{N} level_i \cdot count_i$$

Where $count_i$ is the number of iterative loops in the nesting level i, and N is the deepest level of nesting in the program.

To finally calculate how well an obfuscation transformation changes the nesting complexity, a ratio is taken to represent a change in the transformation which yields $s_{nesting}$. The total nesting is calculated as follows:

$$s_{nesting} = \frac{c'_{nesting} - c_{nesting}}{c_{nesting}}$$

Where $c'_{nesting}$ and $c_{nesting}$ denote the nesting complexity before and after the obfuscation transformation.

Control flow complexity

The control flow complexity measures non-sequential code. Non-sequential means that a program contains labels and `goto` statements. The control flow complexity score is divided into label score, goto score and unreadable score.

The label score is the percentage of duplicate labels in a transformed program. A duplicate label would be the label appearing in two different methods with the same name. The score increases with the number of duplicate labels. The label score is computed as follows:

$$s_{label} = \frac{l'_{duplicate}}{l'_{total}}$$

Where $l'_{duplicate}$ is a number of duplicate labels and l'_{total} is the total number of labels in the transformed application.

The goto score is the percentage of non-sequential `goto` statements in a transformed program. The goto score is computed as follows:

$$s_{goto} = \frac{g'_{non-seq}}{g'_{total}}$$

Where $g'_{non-seq}$ is the number of non-sequential `goto` statements and g'_{total} is the total number of `goto` statements.

The unreadable score involves the number of lines of code that a decompiler could not translate back to source code. Unreadable in this case means that the decompiler leaves Java instructions that is readable by the virtual machine but not for human interpretation. The unreadable score is the percentage of unreadable lines of code in a transformed program and is proportional to the number of uncompiled lines in the code. The unreadable score is computed as follows:

$$s_{unreadable} = \frac{c'_{undecompiled}}{c'_{LOC}}$$

Where $c'_{undecompiled}$ is the number of unreadable lines of code and c'_{LOC} is the total lines of code in the transformed application.

Variable complexity

The variable complexity has four factors that yield its measurement. The factors are: duplicate variable score, extra variable score, description variable and string encryption.

The duplicate variable score, $s_{duplicate}$, measures the percentage of variables in transformated code with the same name but different meanings. The duplicate variable score is computed as follows:

$$s_{duplicate} = \frac{v'_{duplicate}}{v'_{total}}$$

where $v'_{duplicate}$ is the number of duplicate variables and v'_{total} is the number of total variables in the transformed program.

The extra variable score, s_{extra}, measures variables in transformed code with different names but have the same meaning. The extra variable score is computed as follows:

$$s_{extra} = \frac{v'_{extra}}{v'_{total}}$$

where v'_{extra} is the number of extra variables.

The descriptive variable score, $s_{variable}$, is defined as a Boolean expression, that describes if the transformation has renamed descriptive variables in the original code to non-descriptive ones. There are two possible scenarios that can occur. First scenario is that obfuscation does not conduct any variable renamings no matter if the original code have descriptive variables, and yields a score of 0. Second scenario is that the transformation changes descripted variables into non-descriptive ones, and yields a score of 1.

The string encryption score, s_{string}, has an integer range from 0 to 3, with four possible scenarios. The first one is that no string encryption takes place, which gives a score of 0. The second is that string encryptions occur with a decryption method detected in the same class file, with a score of 1. The third is that string encryptions occur with a decryption method detected in the program, which gives a score of 2. The fourth is that string encryptions occur with no decryption method detected, which gives a score of 3.

Program length

The last measurement is the ratio of how many lines of code (LOC) are added or removed in comparison to the original program length. The program length score is computed as follows:

$$s_{LOC} = \frac{c'_{LOC} - c_{LOC}}{c_{LOC}}$$

where c'_{LOC} and c_{LOC} are the count of the transformed code and of the original one.

Overall measurement of potency

The potency score, S_{pot}, which measures how potent a transformation is, consists of the nine variables described above. It is measured on a scale from 0 points to 100 points, where 100 means extremely potent and zero extremely weak. The nine variables are all weighted, depending on how much they influence the potency. Karnick et al. assigned low weight to the factors that add minor overhead into the original code, which results in modest increased time for deciphering the code. A medium weight implies that it has not changed the top-down structure, but adds more time to decipher the code. A high weight implies that the transformation removes the sequential methodology of the structure and adds much overhead and deciphering time. Karnick et al. classified the nine factors and gave them the values as follows:

Low weight: unreadable, duplicate variable, extra variable, and descriptive variable scores. Weight 6.25.

Medium weight: nesting, string encryption and LOC. Weight 12.50.

High weight: label and goto. Weight 18.75.

The final equation for the potency score is computed as follows:

$$S_{pot} = x \cdot s_{nesting} + y \cdot s_{label} + y \cdot s_{goto} + z \cdot s_{unreadable} + z \cdot s_{duplicate} + z \cdot s_{extra} + z \cdot s_{descriptive} + x \cdot s_{string} + x \cdot s_{LOC}$$

where $x = 12.50$, $y = 18.75$ and $z = 6.25$.

2.7.3 Measurement of resilience

The resilience measures how strong the program can hold up against reverse engineering attacks with decompilers. The grading scale for resilience is based on results from decompilers. There are three different scenarios. If no error occurs from decompilation, the score 0 is given. If errors occur during tree parsing, a score of 1 is given. Finally, if the decompilation fails, a score of 2 is given. For a Java program consisting of multiple class files, the score is calculated by averaging the individual scores of the transformation on classes.

2.7.4 Measurement of cost

The cost measures extra resources that an obfuscated program consumes during runtime. Cost is divided into three types: memory, storage space and runtime.

Memory

The memory score is the ratio of additional memory consumption used by an obfuscated program against that of the original one. There are two types of memory: heap memory and non-heap memory. Heap memory is the runtime data area from which memory for all class instances and arrays are allocated. Non-heap memory stores per-class structures such as the runtime constant pool, field and method data. Non-heap memory also stores the code for methods, constructors and interned Strings.

The heap memory score is computed as follows:

$$s_{m-heap} = \frac{p'_{heapmem} - p_{heapmem}}{p_{heapmem}}$$

where $p'_{heapmem}$ is the amount of heap memory consumed by the obfuscated code during runtime, and $p_{heapmem}$ is the amount of heap memory consumed by the original code during runtime.

The non-heap memory score is computed as follows:

$$s_{m-nonheap} = \frac{p'_{nonheapmem} - p_{nonheapmem}}{p_{nonheapmem}}$$

where $p'_{nonheapmem}$ is the amount of non-heap memory consumed by the obfuscated code during runtime and $p_{nonheapmem}$ is the amount of non-heap memory consumed by the original code during runtime.

The total memory score is computed as follows:

$$s_{memory} = a \cdot s_{m-nonheap} + b \cdot s_{m-heap}$$

In the pilot studies made by Karnick et al., they found that $a = 0.375$ and $b = 0.625$ gave the most feasible result. They believed that the heap memory was more critical than the non-heap memory.

Storage

The storage is computed as follows:

$$s_{storage} = \frac{p'_{storage} - p_{storage}}{p_{storage}}$$

where $p'_{storage}$ and $p_{storage}$ denote the file size of the obfuscated program and the original one.

Runtime

The runtime can be very important if a program's purpose is to perform a set of operations quickly and efficiently. The runtime score is computed as follows:

$$s_{runtime} = \frac{p'_{runtime} - p_{runtime}}{p_{runtime}}$$

where $p'_{runtime}$ is the obfuscated program's runtime and $p_{runtime}$ is the original program's runtime.

Overall measurement of cost

The total cost score is computed as follows:

$$S_{cst} = x_2 \cdot s_{memory} + y_2 \cdot s_{storage} + z_2 \cdot s_{runtime}$$

In the pilot studies by Karnick et al., they found that $x_2 = 0.4$, $y_2 = 0.15$, $z_2 = 0.45$ gave the most feasible result. They believed that the memory and runtime are far more expensive resources than the size of the obfuscated program. Negative cost scores represent positive impact on the overall quality of a transformation.

Chapter 3

Methodology

In this chapter the methodology used in this thesis is presented and motived.

3.1 Methods for implementing obfuscating transformations within JBCO

To create new obfuscating transformations some decisions have to be made about how to tackle the implementation. In this section, the choice of intermediate representation within JBCO is motivated, as well as the structure choice of the implementation.

3.1.1 Choice of intermediate representation

To be able to implement obfuscating transformations within JBCO, one needs to be familiar with the framework Soot, which JBCO is build on top of. As mentioned in the previously chapter 2, the JBCO itself is just a number of Jimple and Baf transformations. In this thesis, the obfuscating transformations are created using the representation Jimple. The Jimple representation is used since:

- the typed 3-adress intermediate representation, which makes the representation easy to control and to modify.
- the 15 different statements which makes it easy to handle and use.
- the similarity to Java code, compared with Baf.

3.1.2 Choice of obfuscating transformations

Today, JBCO contains a number of different obfuscating transformations, but there will always be room for more. Batchelder proposed several new transformations

to JBCO. One desirable transformation he proposed was to outline conditions. If statements use conditions to determine if the statement is true or false, also while loops and for loops uses conditions.

JBCO handles a lot of different types of transformations, but has no transformations that obscure arrays and local variables. The first desirable behaviour of an array transformation is to change the structure of it using a scanner array, to confuse a person trying to reverse engineer. The second desirable behaviour of a transformation is to change the structure of a variable, also to confuse a person trying to reverse engineer.

Based on this reasoning, the following three transformations have been chosen:

OT1 Outlining conditions.

OT2 Array restructuring.

OT3 Variable restructuring

Each transformation only handles values of type integer. The delimitation is due to the time constraints since each data type is represented differently in Jimple code. A lot of time is needed to fully understand how the translation from Java source code to Jimple code is done for each data type. Since there is not enough time, the handling of several data types are not prioritized. Instead each transformation only focuses on one data type. The data type integer was chosen because it is commonly used.

3.1.3 Structure

The new obfuscating transformations are stand-alone modules within JBCO, which makes it easy to use only one transformation or to combine multiple. The method used to implement the new obfuscating transformations in JBCO are presented in the list below and can be seen in the figure 3.1.

1. JBCO takes a Java class file containing Java bytecode as input.
2. Soot transforms the Java bytecode to the intermediate Jimple representation.
3. JBCO adds 1 to N obfuscating transformations to the code.
4. Soot transform the obscured intermediate Jimple representation back to Java bytecode.
5. A new Java class file is created as output.

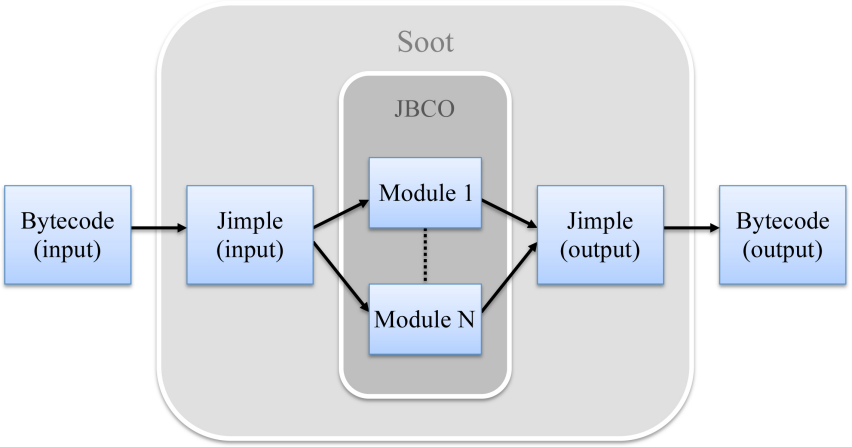

Figure 3.1: A flow chart to describe how the new obfuscating transformations are implemented in Soot. A module corresponds to an obfuscating transformation.

3.2 Analysis methods

To analyse the new transformations, the analysis metrics from chapter 2 are used. The metrics are used to analyze each implemented transformation, despite the fact that the metrics was originally intended to analyze a complete obfuscator. In this section the computer system used and the methods to implement and perform the analysis on the transformations are described.

3.2.1 System

The analytical tests are running on the same computer with the Mac OS X operating system and a Java runtime environment present. This set-up guarantees accurate results because everything uses the same resources. Information about the system can be seen in table 3.1.

Processor	1,8 GHz Intel Core i7
Memory	4 GB 1333 Mhz DDR3
Software	Mac OS Lion 10.7.5

Table 3.1: Information about the computer system used in the analysis.

3.2.2 Quality of Java obfuscating

The metric was calculated according to the equation in section 2.7.1, with the same weights as Karnick et al. used.

3.2.3 Potency

The metric was calculated according to the equation in section 2.7.2, with the same weights as Karnick et al. used on variables x, y and z. The potency score is measured on the Jimple code after the transformation.

Nesting complexity

The nesting complexity score was calculated as it is defined in section 2.7.2 and calculated manually. A negative score (a decrease of nesting complexity) is considered 0.

Control flow complexity

The control flow complexity scores were calculated as they are defined in section 2.7.2. The label score, goto score and unreadable score were calculated manually by hand.

Variable complexity

The variable complexity scores were calculated as they are defined in section 2.7.2. Both the duplicate variable score and the extra variable score score were calculated by hand. The descriptive variable score can only result in 0 or 1, and the score is calculated manually. Retrieve the string encryption score, which can give a score in the range of 0 to 3, is also calculated manually.

Program length

The program length score was calculated as it is defined in section 2.7.2. To get the ratio of how many lines of code are added or removed in comparison to the original program length, an automatic tool was implemented to do this.

3.2.4 Resilience

The resilience score was measured using decompilers. In this thesis, the following two decompilers were used and are described in chapter 2:

- Jad
- Dava

The same grading scale presented in section 2.7.3 was used to rate the result, and the score is given manually by hand. For the interested reader, the result from the two decompilers (Java source code) are also presented, to give the reader a better overview.

3.2.5 Cost

The metric was calculated according to the equation in section 2.7.4, with the same weights as Karnick et al. used on variables x_2, y_2 and z_2.

Memory

The memory scores were calculated as they are defined in section 2.7.4. The total memory score equation used the same values as Karnick et al. used on variables a and b. To get the heap memory score and the non-heap memory score, a profiler tool named YourKit Java Profiler [25] was used to obtain the mean value of heap memory and mean value of non-heap memory consumed by the program during runtime.

Storage

The storage score was calculated as it is defined in section 2.7.4 and is calculated manually by hand.

Runtime

The runtime score was calculated as it is defined in section 2.7.4. An automatic tool was implemented to get the runtime of a program.

3.3 Benchmarks

The benchmarks have been culled from open source projects and well known algorithms. Each project is written in Java source language and is compiled with *Javac*. The benchmarks represent different projects with different purposes and sizes. In the analysis, each program is obfuscated and analysed. In the list below a brief description of each program's key features is presented.

Game solvers: A program that can solve mazes and Sudoku games.

Jadretro: A class transformer tool which helps to successfully decompile Java classes by modern Java compilers (Java 1.4 or later), by transforming the class files into files which could be processed by an old Java decompiler (Java 1.3 or earlier) [15].

MARS: An encryption algorithm. MARS is a shared-key block cipher that works with a block size of 128 bit and a variable key size [7].

Mathematical algorithms: A program that can handle matrix operations such as adding two matrices together, multiply two matrices, subtract one matrix from another matrix, finding the cofactor, determinant, inverse and transpose. The program can also calculate prime numbers, compute the discrete Fourier transform (DFT) and its inverse with the fast Fourier transform (FFT) algorithm, and solve continuous-space linear programming problems by the simplest method.

ProjectEuler: A couple of solutions to Project Euler problems. Project Euler problems are different mathematical problems [11].

Sorting algorithms: Program that sorts elements with the sorting algorithm Quicksort and Radix sort.

Turing Machine: A Turing Machine program that manipulates symbols according to a table of rules.

Virus Scanner: A program that scans for viruses in files [19].

In the analysis all programs are analysed as one single program, which means that the programs are analysed together instead of being analysed separately. This composite program is called *Test_Program* and consists of approximately 10000 lines of Java source code, including blank lines and comments.

Part II

Implementation

Chapter 4

Implementation

In this chapter the new obfuscating transformations that has been developed are presented and the theory learned in the Introduction part of this thesis will come in use.

4.1 The new obfuscating transformations

The main task of this master's thesis was to implement three new obfuscating transformations in JBCO, these three transformations are motivated in chapter 3 and are as follows:

OT1 Outlining conditions.

OT2 Array restructuring.

OT3 Variable restructuring

Each one of these transformations was implemented as a stand-alone module within the JBCO. This means that each transformation can be activated independently and depending on the severity of obfuscation desired, each transformation can also be weighted independently using the weighting mechanism. All transformations use a weighting mechanism. This mechanism can be set within the range of 0-9. If the weight is set to 0, the transformation will not be applied at all, and if it is set to 9 the transformation will be applied everywhere possible. If no weight is specified, a weight of 9 will be used by default. To ensure that the transformations produce output that is correct, none of the transformations can be applied on constructors, fields or main methods in classes.

The modules are written in the programming language Java and the modules were developed in the integrated development environment Eclipse [6].

These new obfuscating transformations can be applied on a program by using the command-line. The transformations are enabled by adding command-line options, see table 4.1:

Obfuscating transformation	**Command-line option**
OT1	`-t`:W: `wjtp.jbco_de`
OT2	`-t`:W: `wjtp.jbco_ar`
OT3	`-t`:W: `wjtp.jbco_vr`

Table 4.1: Command-line options for the new obfuscating transformations OT1, OT2 and OT3, where W is a weight.

To obscure a program in the command-line, the expression below is used:

`java soot.jbco.Main -w -t`:W: `wjtp.jbco`_PN `-app` Example

Where W is a weight, PN is the phase name and Example is the class with the entry point (main method).

4.2 Outlining conditions (OT1)

This obfuscating transformation takes the condition of an if statement, a while loop or a for loop and replaces it with a method call to a new method where the condition has been extracted to. The transformation is designed so that each subcondition of a condition becomes a new method. The new method always returns a Boolean value. This transformation does the opposite of an inline expansion, lets define the term outlining. A pseudo code example of how the outlining condintions transformation works can be seen in listing 4.1 and 4.2.

```
1 [if|while|for](x0 RO_0 y0 LO_0 .. LO_N-1 xN RO_N yN){
2     body
3 }
```

Listing 4.1: Outlining conditions pseudo code example before the transformation. RO stands for Relational operator and LO stands for Logical operator.

```
[if|while|for](newMethod0(x0,y1) LO_0 .. LO_N-1 newMethodN(xN,yN)){
    body
}

boolean newMethod(x0,y0){
    return x0 RO_0 y0
}
.
.
boolean newMethodN(xN,yN){
    return xN RO_N yN
}
```

Listing 4.2: Outlining conditions pseudo code example after the transformation. RO stands for Relational operator and LO stands for Logical operator.

4.2.1 The type of the Outlining conditions transformation

As mentioned in chapter 2 obfuscating transformation can be divided into three categories. This Outlining transformation belongs to the *Obfuscating program structure* category because of the confusing control flow of if statements, while loops and for loops. The code structure after the transformation will make the code harder to reverse engineer.

4.2.2 The Outlining conditions transformation algorithm

The first step in the algorithm is to identify where the transformation can be applied in the program to be obscured. The algorithm scans one method at a time in each class in the program and when the algorithm finds an if statement, a while loop or a for loop where a relational operator exists, the outlining transformation starts. There are several relational operators in the Java language. The various relational operands identified by the transformation can be seen in table 4.2.

Operator	Description
==	Checks if the value of two operands are equal, if yes then the condition becomes true.
!=	Checks if the value of two operands are not equal, if yes then the condition becomes true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then the condition becomes true.

<	Checks if the value to left operand is less than the value of right operand, if yes then the condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then the condition becomes true.

Table 4.2: The relational operands the outlining transformation can identify.

The data types used in a condition can be clustered into three different groups. Each group is handled differently by Soot, due to the Jimple code which differs for each one of them. The three data types groups can be seen in table 5.9. Only variables in the first group is transformed by the Outlining conditions transformation, which means that both variables in the condition must belong to group one.

Category	**Data types**	**Jimple statements used**
C1	int, byte, char, short, boolean	IfStmt
C2	long, float, double	AssignStmt(Comparison expression) followed by IfStmt
C3	String	AssignStmt(Virtual Invoke expression) followed by IfStmt

Table 4.3: The three data type groups the Outlining conditions transformation can handle. For more information about the Jimple statements, see table 2.3.

The algorithm, which makes the transformation possible, operates as follows:

1. Let E be the expression to be transformed if: E is an instance of IfStmt, does not contain a local Jimple variable, both variables belongs to group one and the weighting mechanism allows the transformation.

2. Let C be the container that stores the variables: `goto` and destination d.

3. Create a new method m with the information found in C. The method name is chosen in the same way as in the *Renaming Identifiers* transformation. The new method returns the Boolean value of E.

4. Create a new expression which calls m and inserts the expression before E.

5. Create another expression which checks if the return value of m is `true` or `false`, and let the `goto` destination be d. Insert the expression before E and remove E.

4.2.3 A concrete example of the Outlining conditions transformation

To demonstrate how the outlining transformation works in reality, a hands-on example is presented. In listing 4.3 a Java source code snippet before the transformation is shown and in listing 4.4 a Java source code snippet after the transformation can be seen. The if statement condition in the example consists of two subconditions. For each subcondition a method will be created.

```
1 public void convertG(int gLoop, int gValue, int gType){
2     while(gLoop >= 0){
3         if(gValue > 1000 && gType != 1){
4             createG4367();
5         }
6     }
7 }
```

Listing 4.3: A Java source code snippet before obfuscation with the outlining transformation.

```
1  public void convertG(int gLoop, int gValue, int gType){
2  while(________(gLoop, 0)){
3     if(S$5$5(gValue, 1000) && ll1Il(gType, 1))
4          createG4367();
5      }
6  }
7
8  private boolean ________(int i0, int i1){
9      return i0 >= i1;
10 }
11
12 private boolean S$5$5(int i0, int i1){
13     return i0 > i1;
14 }
15
16 private boolean ll1Il(int i0, int i1){
17     return i0 != i1;
```

```
18 }
```

Listing 4.4: A Java source code snippet after obfuscation with the outlining transformation. For each subcondition a new method is created.

4.3 Array restructuring (OT2)

This obfuscating transformation changes the structure of an integer array, to make it more difficult for an attacker to understand its structure at runtime. The transformation creates a scanner array to each array and the scanner arrays are used to get the right order of each array. A pseudo code example of how the transformation works can be seen in listing 4.5 and 4.6.

```
1 int[] array = {a, b, c, d, e};
2 for(int i = 0; i < array.length; i++){
3     do something with array[i];
4 }
```

Listing 4.5: Array restructuring pseudo code example before obfuscating transformation.

```
1 int[] array = {b, e, a, d, c};
2 int[] scanner = {w, u, y, x, v};
3 for(int i = 0; i < array.length; i++){
4     do something with array[scanner[i]];
5 }
```

Listing 4.6: Array restructuring pseudo code example after obfuscating transformation.

4.3.1 The type of the Array restructuring transformation

As mentioned in chapter 2 obfuscating transformation can be divided into three categories, this Array restructuring transformation belongs to the *Operator level obfuscations* category due to the restructuring of the low-level logic. The code structure after the transformation will make the code harder to reverse engineer.

4.3.2 The Array restructuring transformation algorithm

The algorithm consists of two phases. For all phases, let P be the program to be obscured and let A be the container which holds the array names of all arrays which

have been modified in P.

The first phase of the algorithm identifies where the transformation can be applied and more important, where the transformation cannot be applied in the program to be obscured. It searches for arrays of the type integer and scans one method at a time in each class in the program. The algorithm for the first phase works as follows:

1. Let M be the method being examined. If M returns a variable of type Array where the values are of type `int`, dismiss the method. If not, read M line by line.
2. Let H be a container that stores arrays which will not be transformed and let A be an array of type `int` in M.
3. If a field is assigned the value of A, add A to H. If a method call exist with parameter A or a method call with return type Array where the values are of type `int`, dismiss the method.

The second phase detects approved arrays, modifies them and creates a scanner to each array as follows:

1. Let OA be the array to be transformed in P, if the examined expression is an instance of AssignStmt, the right side of the statement is an instance of Array Type, does not exist in H and the type is an instance of `int` and the weighting mechanism allows the transformation.
2. Let C be a container that stores variable names and the size s of OA.
3. Create a new scanner array SA of size s in C and randomly shuffle the elements in SA.
4. Create a new empty array NA of size s in C which later will contain the elements from OA, but in a different order so the elements can be accessed by SA.
5. Iterate through each position in OA and store the elements in NA with the new position which can be accessed by SA. Give NA the same variable name as OA. Add NA to P.
6. Add the variable name of NA to A.
7. Add SA to P.

In the second phase the algorithm finds all occurrences where the old arrays were used. The algorithm uses A to find them. If the algorithm finds an occurrence, then the pointer to NA is redirected to SA, which will give the correct value of NA.

4.3.3 A concrete example of the Array restructuring transformation

To demonstrate how the Array restructuring transformations works in reality, a hands-on example is presented. In listing 4.7 a Java source code snippet before the transformation is shown and in listing 4.8 a Java source code snippet after the transformation can be seen. The array in this example contains the first 10 prime numbers and the method only prints them in the correct order.

```
1 public void primeNumbers(){
2     int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
3     for(int i = 0; i < primes.length; i++){
4         System.out.println(primes[i]);
5     }
6 }
```

Listing 4.7: A Java source code snippet before obfuscation with the Array restructuring transformation.

```
1 public void primeNumbers(){
2     int[] primes = {19, 3, 13, 29, 7, 17, 23, 2, 11, 5};
3     int[] scanner = {7, 1, 5, 9, 3, 6, 8, 0, 4, 2};
4     for(int i = 0; i < primes.length; i++){
5         System.out.println(primes[scanner[i]]);
6     }
7 }
```

Listing 4.8: A Java source code snippet after obfuscation with the Array restructuring transformation.

4.4 Variable restructuring (OT3)

This obfuscating transformation changes the structure of an integer variable, to make it more difficult for an attacker to understand the value at runtime. The Variable transformation takes an integer variable and creates a new char array of the same size containing random characters. For each element in the new char array a digit is added and in the end the char array contains the same value as the integer value. The only difference is that the integer value is now hidden within the char array.

A pseudo code example of how the transformation works can be seen in listing 4.9 and 4.10.

```
int variable = i;    //i = i0i1i2...iN
```

Listing 4.9: Variable restructuring pseudo code example before obfuscating transformation. Each ix, where $0 \leq x \leq N$, represents a subnumber of x.

```
char[] temp = {c0, c1, c2, ..., cN};
temp[0] += y0;
temp[1] += y1;
temp[3] += y2;
.
.
temp[c.length] += yN;
int variable = Integer.parseInt(new String(temp));
```

Listing 4.10: Variable restructuring pseudo code example after obfuscating transformation. Each yx, where $0 \leq x \leq N$, represents a value which is connected to c (see section 4.4.2 to understand how they are connected).

4.4.1 The type of the Variable restructuring transformation

As mentioned in chapter 2 obfuscating transformation can be divided into three categories, this Variable restructuring transformation belongs to the *Operator level obfuscations* category due to the restructuring of the low-level logic. The code structure after the transformation will make the code harder to reverse engineer.

4.4.2 The Variable restructuring transformation algorithm

The first step in the algorithm is to identify where the transformation can be applied in the program to be obscured. The algorithm scans one method at a time in each class in the program and when the algorithm finds an variable of type `int`, then the Variable restructuring algorithm transforms the variable. Let P be the program to be obscured. The algorithm works as follows:

1. Let i be the variable of type `int`, if the examined expression is an instance of AssignStmt and of type `int`.

2. Let C be a container that stores the variable name of i and number of characters n that the variable consists of (units = 1, tens = 2, hundreds = 3, and so on).

3. Create an char array O of i and a char array R of length n. Populate R with random char elements within the range of 65 to 122. The interval is chosen

based on the decimal values that correspond to the characters: a-z, A-Z, [,], ˆ, _, ‘. For better understanding see Ascii table 4.4. Add the initiation of R to P.

4. For each element at position j in R, create an expression u which adds a specific digit to the element, to regain the same value that can be find in the same position j in O. The digit is obtained by the subtraction: $R[j] = O[j] - R[j]$. Add each u to P.

5. Create an expression f which converts R to type `int`, let g be the integer value. Add f to P. The value of g is not explicitly visible in f, it is hidden within f, which complicates the process to see the true value of g. The variable g is equal to i.

Dec	Char	Dec	Char	Char	Dec	Char	Dec	Dec	Char
65	A	77	M	89	Y	101	e	113	q
66	B	78	N	90	Z	102	f	114	r
67	C	79	O	91	[	103	g	115	s
68	D	80	P	92	\	104	h	116	t
69	E	81	Q	93	]	105	i	117	u
70	F	82	R	94	ˆ	118	j	112	v
71	G	83	S	95	_	119	k	112	w
72	H	84	T	96	‘	108	l	120	x
73	I	85	U	97	a	109	m	121	y
74	J	86	V	98	b	110	n	122	z
75	K	87	W	99	c	111	o		
76	L	88	X	100	d	112	p		

Table 4.4: Ascii table that the variable restructuring transformation uses, with the exception of character "\" with decimal value 92.

4.4.3 A concrete example of the Variable restructuring transformation

To demonstrate how the Variable restructuring transformations works in reality, a hands-on example is presented. In listing 4.11 a Java source code snippet before the transformation is shown and in listing 4.12 a Java source code snippet after the transformation can be seen. The method in the example adds two integer variables together and returns the value.

```
1 public int getPrice(){
2     int price1 = 1495;
```

```
3       int price2 = 69;
4       return price1 + price2;
5   }
```

Listing 4.11: A Java source code snippet before obfuscation with the Variable restructuring transformation.

```
1   public int getPrice(){
2       char[] temp0 = {'d', 'L', 'B', 'H'};
3       temp0[0] += -51;
4       temp0[1] += -24;
5       temp0[2] += -9;
6       temp0[3] += -19;
7       int price1 = Integer.parseInt(new String(temp0));
8
9       char[] temp1 = {'F', '_'};
10      temp1[0] += -16;
11      temp1[1] += -38;
12      int price2 = Integer.parseInt(new String(temp1));
13
14      return price1 + price2;
15  }
```

Listing 4.12: A Java source code snippet after obfuscation with the Variable restructuring transformation.

Part III

Results

Chapter 5

Results

In this chapter the results are presented. For details on how the different scores are calculated and obtained, read section 2.7 in chapter 2 and section 3.2 in chapter 3.

5.1 Outlining conditions (OT1)

Number of transformation occurrences:	**444**

The Outlining conditions transformation was applied 444 times to the *Test_Program*, which means that 444 new methods were created. Each new method consists of 14 lines of Jimple code and one extra line of Jimple code to call the new method.

5.1.1 Potency

The result of the nesting complexity (nesting score) for the Outlining condition transformation was 0, since no such nesting changes were made.

In table 5.2, the result of the control flow complexity (label score, goto score and unreadable score), variable complexity (duplicate variable score, extra variable score, descriptive variable score and string encryption score), program length (length score) and overall potency score is presented for the Outlining condition transformation. Most scores were 0, since there were no duplicated labels or variables, non-seq goto, unreadable lines, extra variables, descriptive variables or string encryption.

	Transformed
Number of labels	3276
Number of duplicate labels	0
Label score	**0**
Total number of goto	3768
Number of non-seq goto	0
Goto score	**0**
Total number of lines	35031
Number of unreadable lines	0
Unreadable score	**0**
Total number of variables	7008
Number of duplicate variables	0
Duplicate variable score	**0**
Total number of variables	7008
Number of extra variables	0
Extra variable score	**0**
Descriptive variable score	**0**
String encryption score	**0**
Total number of lines in original	24432
Total number of lines in transformed	35031
Length score	**0.4338**
Overall potency score	**5.42**

Table 5.2: The result for all potency scores.

5.1.2 Resilience

In table 5.3, the result after using the decompilation tools Dava and Jad are presented for the Outlining condition transformation. The result is based on the Java source code after the transformation.

	Original	**Dava**	**Jad**
Number of labels	0	80	146
Number of duplicate labels	0	0	10
Total number of goto	0	0	120
Total number of lines	7921	12916	15646
Number of unreadable lines	0	2275	200
Total number of variables	1801	2168	2559
Number of extra variables	0	2	0

Table 5.3: The result for the Outlining condition transformation after using the decompilation tools Dava and Jad, in an attempt to reproduce the Java source code again.

Both Dava and Jad gave the same resilience score of 1 and the two decompilers are therefore not treated separately. Since both tools gave the same result, it does not matter which decompilation tool is used.

5.1.3 Cost

In table 5.4, the result of the cost (overall memory score, storage score, runtime score and overall cost score) is presented for the Outlining condition transformation.

	Original	*Transformed*
Mean value of heap memory (MB)	72.64	67.03
Mean value of non-heap memory (MB)	9.09	9.71
Heap memory score	-0.0772	
Non-heap memory score	0.0686	
Overall memory score	**-0.0225**	
Size (KB)	272	779
Storage score	**1.8636**	
Runtime (s)	72	76
Runtime score	**0.0556**	
Overall cost score	**0.30**	

Table 5.4: The result for all cost scores.

5.1.4 Quality of obfuscation

The quality of obfuscation is 2.47 for the Outlining condition transformation.

5.2 Array restructuring (OT2)

Number of transformation occurrences:	**4**

The Array restructuring transformation was applied 4 times to the *Test_Program*, which means that 4 new arrays of were created. The number of new lines of Jimple code is $n+2$ for each integer array, where n is the number of elements in the array.

Each time an element in the array was used, 2 new lines of Jimple code was added to the code to access the right element through the scanning array.

5.2.1 Potency

The result of the nesting complexity (nesting score) for the the Array restructuring transformation was 0, since no such nesting changes were made.

In table 5.6, the result of the control flow complexity (label score, goto score and unreadable score), variable complexity (duplicate variable score, extra variable score, descriptive variable score and string encryption score), program length (length score) and overall potency score is presented for the Array restructuring transformation. Most scores were 0, since there were no duplicated labels or variables, non-seq goto, unreadable lines, extra variables, descriptive variables or string encryption.

	Transformed
Number of labels	2832
Number of duplicate labels	0
Label score	**0**
Total number of goto	3324
Number of non-seq goto	0
Goto score	**0**
Total number of lines	29150
Number of unreadable lines	0
Unreadable score	**0**
Total number of variables	6231
Number of duplicate variables	0
Duplicate variable score	**0**
Total number of variables	6231
Number of extra variables	0
Extra variable score	**0**
Descriptive variable score	**0**
String encryption score	**0**
Total number of lines in original	24432
Total number of lines in transformed	29150
Length score	**0.1931**
Overall potency score	**2.41**

Table 5.6: The result for all potency scores.

5.2.2 Resilience

In table 5.7, the result after using the decompilation tools Dava and Jad are presented for the Array restructuring transformation. The result is based on the Java source code after the transformation.

	Original	**Dava**	**Jad**
Number of labels	0	80	146
Number of duplicate labels	0	0	10
Total number of goto	0	0	118
Total number of lines	7921	10320	11077
Number of unreadable lines	0	2275	200
Total number of variables	1801	1532	1693
Number of extra variables	0	2	3

Table 5.7: The result for the Array restructuring transformation after using the decompilation tools Dava and Jad, in an attempt to reproduce the Java source code again.

Both Dava and Jad gave the same resilience score of 1 and the two decompilers are therefore not treated separately. Since both tools gave the same result, it does not matter which decompilation tool is used.

5.2.3 Cost

In table 5.8, the result of the cost (overall memory score, storage score, runtime score and overall cost score) is presented for the Array restructuring transformation.

	Original	*Transformed*
Mean value of heap memory (MB)	72.64	63.97
Mean value of non-heap memory (MB)	9.09	9.03
Heap memory score	-0.1192	
Non-heap memory score	-0.0058	
Overall memory score	**-0.0767**	
Size (KB)	272	705
Storage score	**1.5917**	
Runtime (s)	72	75
Runtime score	**0.0417**	
Overall cost score	**0.23**	

Table 5.8: The result for all cost scores.

5.2.4 Quality of obfuscation

The quality of obfuscation is 1.26 for the Array restructuring transformation.

5.3 Variable restructuring (OT3)

Number of transformation occurrences:	**811**

The Variable restructuring transformation was applied 811 times to the *Test_Program*. The number of new lines of Jimple code is $n \cdot 7 + 6$ for each integer variable, where n is the length of the variable.

5.3.1 Potency

The result of the nesting complexity (nesting score) for the the Variable restructuring transformation was 0, since no such nesting changes were made.

In table 5.10, the result of the control flow complexity (label score, goto score and unreadable score), variable complexity (duplicate variable score, extra variable score, descriptive variable score and string encryption score), program length (length score) and overall potency score is presented for the variable restructuring transformation. Most scores were 0, since there were no duplicated labels or variables, non-seq goto, unreadable lines, extra variables, descriptive variables or string encryption.

	Transformed
Number of labels	2832
Number of duplicate labels	0
Label score	**0**
Total number of goto	3324
Number of non-seq goto	0
Goto score	**0**
Total number of lines	38670
Number of unreadable lines	0
Unreadable score	**0**
Total number of variables	9857
Number of duplicate variables	0

Duplicate variable score	**0**
Total number of variables	9857
Number of extra variables	0
Extra variable score	**0**
Descriptive variable score	**0**
String encryption score	**0**
Total number of lines in original	24432
Total number of lines in transformed	38670
Length score	**0.5828**
Overall potency score	**7.29**

Table 5.10: The result for all potency scores.

5.3.2 Resilience

In table 5.11, the result after using the decompilation tools Dava and Jad are presented for the Variable restructuring transformation. The result is based on the Java source code after the transformation.

	Original	**Dava**	**Jad**
Number of labels	0	80	150
Number of duplicate labels	0	0	11
Total number of goto	0	0	119
Total number of lines	7921	13078	18879
Number of unreadable lines	0	2275	200
Total number of variables	1801	3288	2582
Number of extra variables	0	210	0

Table 5.11: The result for the Variable restructuring transformation after using the decompilation tools Dava and Jad, in an attempt to reproduce the Java source code again.

Both Dava and Jad gave the same resilience score of 1 and the two decompilers are therefore not treated separately. Since both tools gave the same result, it does not matter which decompilation tool is used.

5.3.3 Cost

In table 5.12, the result of the cost (overall memory score, storage score, runtime score and overall cost score) is presented for the Variable restructuring transformation.

	Original	*Transformed*
Mean value of heap memory (MB)	72.64	194.40
Mean value of non-heap memory (MB)	9.09	8.07
Mean value of heap memory score	1.6762	
Mean value of non-heap memory score	-0.1103	
Overall memory score	**1.0062**	
Size (KB)	272	995
Storage score	**2.66**	
Runtime (s)	72	138
Runtime score	**0.9167**	
Overall cost score	**6.95**	

Table 5.12: The result for all cost scores.

5.3.4 Quality of obfuscation

The quality of obfuscation is 3.22 for the Variable restructuring transformation.

Chapter 6

Discussion

In this chapter the results are discussed and analysed thoroughly. First the potency is discussed, then the resilience and lastly the cost for each obfuscating transformation is discussed. In this chapter, the abbreviations for each obfuscating transformation name from section 4.1 are used in the figures.

6.1 Overall analysis

All obfuscating transformations could be applied to the *Test_Program*[1]. The results for each transformation can be seen in figure 6.1. Variable restructuring had the highest number of transformation occurrences and Array restructuring had the lowest number of occurrences.

The Outlining conditions transformation was applied 444 times to the program. The large number can be explained by the frequent use of control statements in programs, which includes if statements, while loops and for loops.

The Array restructuring transformation was only applied 4 times to the program, which means that it was not a useful transformation in this particular case, since a small number of transformations do not create as much confusion in the obfuscated code as a larger number of transformations would do. The transformation could have been dismissed. Many factors must be met, for the transformation to be applied in the code. The most important factor is that the transformation looks for an integer array that is declared and initialized at the same time. The second most important factor is that the array is not a field and is not created in the constructor or in the main method. According to the test program, an array that can meet both requirements seems uncommon.

[1]Read about *Test_Program* in section 3.3 in chapter 3.

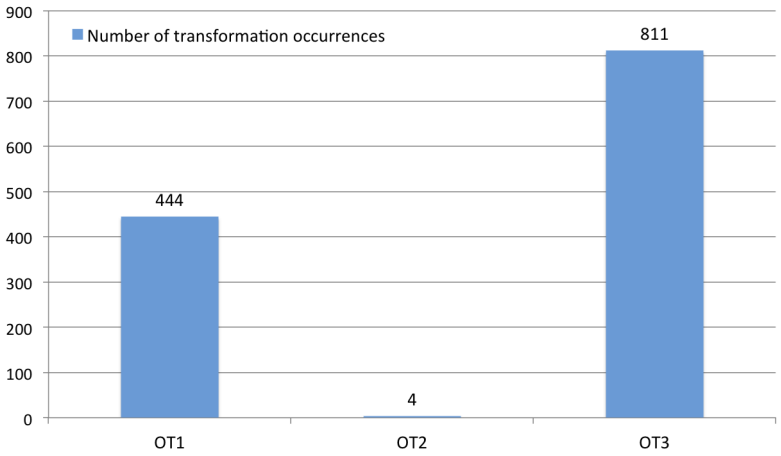

Figure 6.1: The number of transformation occurrences for each transformation.

The Variable restructuring transformation was applied 811 times to the program. The large number can be explained due to the large number of integer variables.

6.2 Potency

The overall potency score for each obfuscating transformation can be seen in figure 6.2. Variable restructuring gave the highest score and Array restructuring gave the lowest score.

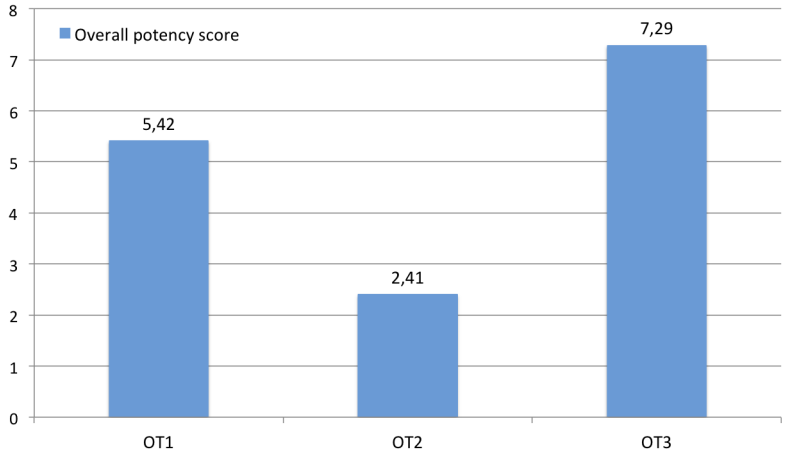

Figure 6.2: The overall potency score for each transformation.

6.2.1 Outlining conditions

The overall potency score for Outlining conditions was 5.42 out of 100, which means that the transformation gave a positive level of obscurity to the code.

Outlining conditions did not affect the nesting complexity in the program, since the transformation does not create any new loops, only changes the conditions of if statements, while loops and for loops. Neither did any duplicate labels, non-sequence gotos, unreadable lines, duplicate variables or descriptive variables appear, which was as expected. Also, no string encryption was added to the program.

The main difference between the original Jimple code compared with the Jimple code after the transformation, was the code size. When an Outlining conditions transformation is made, the code size grows. In *Test_Program* the transformation was applied 444 times, which means that 444 new methods were created, each one with 14 lines of Jimple code and one extra line to call the new method. On this assumption, the total size of the transformed program would be the original size (24432), see table 5.2, plus number of transformations (444) multiplied with the number of new lines $(14 + 1)$. This returns the result of 31092 lines of code. The calculated result for the total number of lines after transformation (31092) does not match the result from the survey in this thesis (35031), which can be seen in table 5.2. There is a difference of 3939 lines of code, which means that JBCO increases the size of the code. The extra code is probably added when the original bytecode is disassembled and converted to Jimple code and then reassembled to bytecode again. The translation back to bytecode from Jimple code is probably done differently, which gives the bytecode another structure.

6.2.2 Array restructuring

The overall potency score for Array restructuring was 2.41 out of 100, which means that the transformation gave a positive level of obscurity to the code, but the score was close to zero.

The Array restructuring did not affect the nesting complexity in the program, since the transformation only change integer arrays. Neither did any duplicate labels, non-sequence gotos, unreadable lines, duplicate variables or descriptive variables appear, which was as expected. Also, no string encryption was added to the program.

As for the Outlining conditions transformation, the main difference between the original Jimple code compared with the Jimple code after the transformation, was also the code size. Since the Array restructuring transformation only was applied 4 times, the code should have grown only a little in size. According to the result in table 5.6, the total number of lines in the original program was 24432 and the

total number of lines in the transformed program was 29150. That is a difference of 4718 lines of code. The code is expected to grow, but not that drastically, there is approximately 4000 lines of additional code. The Outlining conditions transformation also had about 4000 lines of extra code, that seems to correlate with the Array restructuring transformation. In the previous section 6.2.1 these extra lines of code are discussed and explained.

6.2.3 Variable restructuring

The overall potency score for Variable restructuring was 7.29 out of 100, which means that the transformation gave a positive level of obscurity to the code.

The Variable restructuring did not affect the nesting complexity in the program, since the transformation only change integers. Neither did any duplicate labels, non-sequence gotos, unreadable lines, duplicate variables or descriptive variables appear, which was as expected, due to the inner workings of the transformation. Also, no string encryption was added to the program.

As was the case with the Outlining conditions and Array restructuring transformations, the main difference between the original Jimple code compared with the Jimple code after the transformation, was also the code size. When a Variable restructuring transformation is made, the code size grows. It is hard to calculate how much the size grows, because it depends on how large each number is. A program that contains many large integer numbers will after the transformation consist of more lines of code than a program with few large integer numbers.

According to the result in table 5.10 the total number of lines in the original program was 24432 and the total number of lines in the transformed program was 38670, which gives a difference of 14238 lines of code. The transformation was applied 811 times to the program, and the number of new lines of Jimple code is $n \cdot 7 + 6$ for each integer variable, where n is the length of the variable. There are often many numbers of length 1 in program, but of course there are also large numbers in the program. To get the exact number of extra lines, the average integer length was calculated to 1.13 for the *Test_Program*. If the mean value of each integer in the program was of length 1.13, the number of extra lines of code would be $1.13 \cdot 7 + 6$ multiplied with 811, which is 11281. The total number of lines would be $24432 + 11281$, which is 35713. The calculated result for the total number of lines after transformation (35713) does not match the result from the survey in this thesis (38670) in table 5.10. There is a difference of 2957 lines of code. The Outlining conditions and Array restructuring did also have lines of extra code, which seems to be a correlation for all three transformations. In the section 6.2.1 these extra lines of code are discussed and explained.

6.3 Resilience

All obfuscating transformations with both Dava and Jad have the same resilience score 1 out of 2, which means that error occurred during tree parsing. It is safe to say that the problem regarding the decompilation is not due to the transformation. Both decompilers failed to decompile the same number of lines in each obfuscating transformation. Dava failed to decompile 2275 lines of code and Jad failed to decompile 200 lines of code, see tables 5.3, 5.7 and 5.11.

Despite the fact that Jad manages to decompile more lines of code than Dava, the quality of the Java source code was lower. The code decompiled with Jad would require more work compared with the code decompiled with Dava, to make the test program executable. It is hard to understand how Jad works since there is no documentation about how the decompilation process works. Dava is a newer decompiler and is probably a more advanced tool than Jad since it uses flow-analysis information. That is probably the reason why Dava performs better than Jad.

The resilience score in the qualitative analysis is determined by how well the decompiler manages to decompile the Java code. It is only based on whether there were no errors from decompilation, if errors occur during tree parsing, or if the decompilation fails. The resilience score does not take into account the number of errors or how the code is structured, which in this case do not affect the choice of decompiler. Therefore, no further investigation was made regarding why Dava and Jad behave differently apart from the short analysis below.

For the interested reader, the test program decompiled with Jad contained more lines of code, more labels and `goto` statements compared with the test program decompiled with Dava, for all transformations. For both the Outlining conditions transformation and the Array restructuring transformation, the code decompiled with Jad contained more variables than the code decompiled with Dava. It is the opposite for the Variable restructuring transformation, where the code decompiled with Dava contained more variables than the code decompiled with Jad. This probably means that Dava treats arrays differently than Jad.

6.4 Cost

The overall cost score for each obfuscating transformation can be seen in figure 6.3. The cost score does not have an upper limit, but a positive score has negative impact on the overall quality of a transformation. Variable restructuring gave the highest score and Array restructuring gave the lowest score.

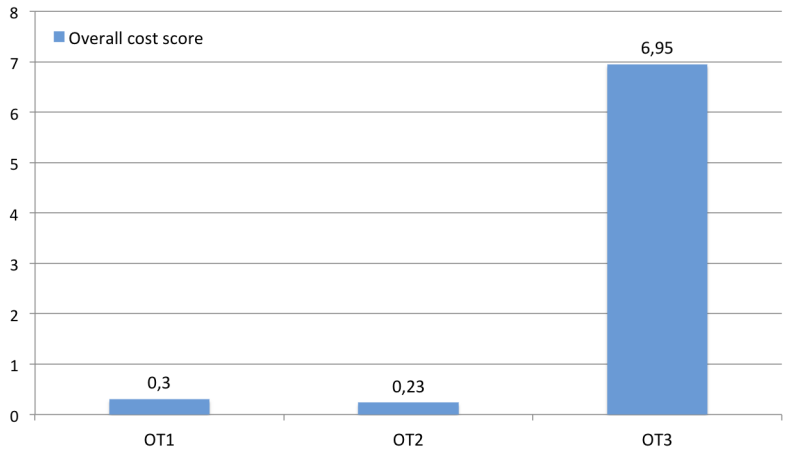

Figure 6.3: The overall cost score for each transformation.

6.4.1 Outlining condition

The overall cost score for Outlining conditions was 0.30, which has a negative impact on the quality.

The memory score was -0.0225 and a negative score has a positive impact on the overall quality. The mean value of heap memory was higher for the original test program compared to the transformed program, but the difference was very small. Probably some optimizations were made by the JBCO, when the code was converted between different representations. The mean value of non-heap memory was lower for the original program compared to the transformed program, which was as expected, because the transformed test program consists of more methods than the original test program.

The storage score was 1.8636, which has a negative impact on the overall quality. As expected, the size of the transformed test program was larger than the size of the original test program. The transformed test program is larger because a new method is added each time the transformation is applied.

The runtime score was 0.0556, which has a negative impact on the overall quality. As expected the runtime for the original program was shorter than the runtime for the transformed program, due to that the transformed program consists of more methods and method calls.

6.4.2 Array restructuring

The overall cost score for Array restructuring was 0.23, which has a negative impact on the quality.

The memory score was −0.0767, which have a positive impact on the overall quality. The mean value of heap memory was higher for the original test program compared to the transformed program. As discussed in the precious section 6.4.1, some optimizations were probably made by the JBCO. The non-heap memory score was higher for the original program compared to the transformed program. Once again, it has to do with the optimizations made by the JBCO.

The storage score was 1.517, which has a negative impact on the overall quality. The storage score was not expected, due to the fact that the transformation only was applied 4 times to the code. Learnt from section 6.2, the JBCO adds extra lines of code to a program, which explains why the transformed program is of larger size. Even though extra lines of code are added to the transformed program, the size should not be twice the original program's size.

The runtime score was 0.0417, which has a negative impact on the quality. The runtime for the original program was shorter than the runtime for the transformed program, which can be explained by the extra lines of code that have been added to the transformed program.

6.4.3 Variable restructuring

The overall cost score for Variable restructuring was 6.95, which has a negative impact on the quality.

The memory score was 1.0062, which have a negative impact on the overall quality. The mean value of heap memory was higher for the transformed test program compared to the original program. The result is explained due to the restructuring of all integer variables, which requires more memory. The mean value of non-heap memory was lower for the transformed program compared to the original program, since there were no new method calls and due to optimization by the JBCO.

The storage score was 2.66, which has a negative impact on the overall quality. The size was expected to be larger after the transformation, due to the restructuring of all integer variables.

The runtime score was 0.9167, which has a negative impact on the quality. The runtime for the original program was shorter than the runtime for the transformed program, which was expected due to the restructuring of all integer variables.

6.4.4 Quality of obfuscation

The quality for each obfuscating transformation can be seen in figure 6.4. The quality score does not have an upper limit, but a positive score increases the level of obfuscation. The resilience result for Dava and Jad was the same for all transformations, and they are therefore not treated separately in the figure 6.4, where the result for the quality is presented for each transformation.

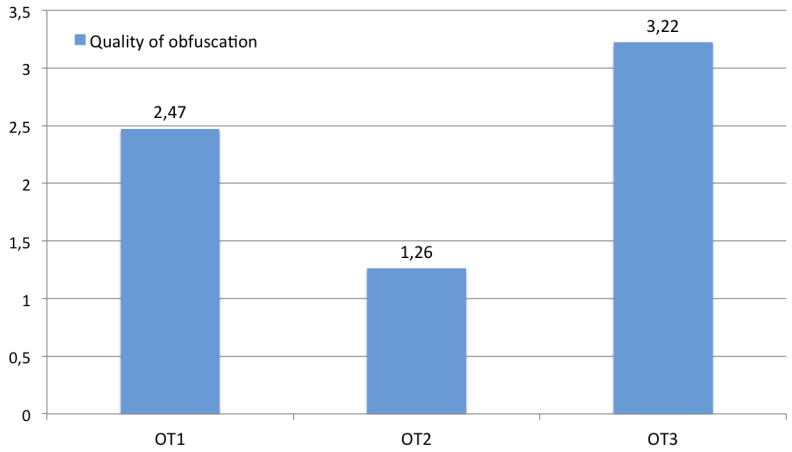

Figure 6.4: The result for the quality of each obfuscating transformation.

The Variable restructuring gave the highest quality of obfuscation and Array restructuring gave the lowest. This was expected, because the Variable restructuring transformation was applied 811 times to the program compared to the Array restructuring transformation, which only was applied 4 times to the program. It is important to keep in mind that the quality result may differ depending on which program is to be obscured.

The level of obscurity was low for all obfuscating transformations, but if each transformation is combined with all the other transformations, the level of obfuscation will increase. It is important to remember that the intention is not to run each transformation separately; the intention is to use each transformation together with other transformations in JBCO, which hopefully will give a higher and more satisfying level of obscurity to the program.

Chapter 7

Conclusions

In this chapter the conclusions of the thesis are presented. The conclusions are based on the results presented in chapter 5 and chapter 6.

7.1 Quality of obfuscation

The results presented in chapter 5 indicate that obfuscating transformations increases the level of obscurity of the *Test_Program*[1]. The transformations implemented in this thesis are not as general as many of the already existing transformations in JBCO (presented in chapter 2). More general transformations will most likely give a higher quality score than transformations that are more specific. The purpose of this thesis was not to create transformations that could be compared with the existing transformations in JBCO that are very general. It would be unfair to compare them against each other because they are different in the aspect of generality. The purpose was to add new obfuscating transformations to the JBCO, transformations that would offer extra value to the tool, which the new transformations did.

As previously mentioned, it is important to remember that an obfuscator does not guarantee one hundred percent protection. Given the right amount of time and effort, the code can be reverse engineered. A person trying to reverse engineer *Test_Program* will first encounter problems when trying to decompile the code with a decompilation tool, such as Dava and Jad. Subsequently, errors will occur during the tree parsing phase, which will result in non-executable code or result in no code at all. The successfully decompiled code will be messy and difficult to read, and will complicate the reverse engineer process.

From an analytical perspective, it is hard to determine how difficult it is to reverse engineer obfuscated code because it is a subjective process. Regarding how difficult

[1] Read about *Test_Program* in section 3.3 in chapter 3.

it is, different people would have different opinions. This makes it cumbersome to test the quality of the obfuscations in a quantitatively and scientific way. A thorough evaluation of such a survey would require a large number of participants and it could be hard to find qualified participants and it could also be costly. Unfortunately, this kind of survey could not be carried out in this thesis due to the reasons mentioned above. Instead, the analysis metric used in this thesis are partly based on decompilers, which gives a good indication of how hard it is to decompile code, without using any participants. The drawback with the method used is that an evaluation with decompilers does not reflect the reality to the same extent as a survey with people would do. Despite this fact, decompilers should not be underestimated; they are excellent tools which have been very useful and have provided credible results.

A problem with obfuscation is that the transformed programs often tend to increase in size and memory usage. In many cases, new code is added to the program after the transformation, such as adding dead-code to the program to complicate flow analyses. The obfuscating transformations this thesis focuses on (Outlining conditions, Array restructuring and Variable restructuring) are no exceptions. As can be seen in the result, the size and memory usage increases. This is not necessarily bad; it depends on how the program should be used and which delimitations that exist. One solution to minimize the size would be to reduce the weight of the transformation, which means that the transformation is applied less frequently in the code. This is a setting that the user can experiment with in order to find the best result to fulfil his or her purpose.

The analysis metrics used in this thesis were originally intended to analyse a complete obfuscation, not to analyse one transformation at a time. This fact explains the low scores obtained in chapter 5 and why many of the factors used in the analysis yielded a value of zero. Despite the low scores from the analysis metrics, comparing the factors with values greater than zero gave valuable information about each obfuscating transformation. However, a value of zero is still a valid result and a descriptive outcome.

7.2 Outlining conditions

The Outlining conditions transformation increased the level of obscurity to the program. The transformation changed the flow control of the program, which in this case means that the conditions of if statements, while loops, and for loops were outlined in new methods.

This transformation increases both size and memory usage of the program, which the person using the transformation have to keep in mind in order to get the best

result. A good advice is to set a low weight to the transformation if the size of the program after the transformation is considered too big compared to the original program size.

7.3 Array restructuring

As the Array restructuring transformation is designed right now, it is weak and many factors must be met in order for the transformation to be applied on the code. The transformation changes the structure of integer arrays to make it more difficult for an attacker to understand their structures at runtime.

Despite the fact that the transformation was only applied 4 times to the *Test_Program*, the transformation can be useful in programs that contains many integer arrays. Compared to Outlining conditions and Variable restructuring, this Array restructuring transformation is not as general as they are.

7.4 Variable restructuring

The Variable restructuring transformation increased the level of obscurity to the program. The transformation changes the structure of integer variables, to make it more difficult for an attacker to understand the values at runtime.

This transformation also increases size and memory usage of the program, which can be seen in the results. The size can be reduced by changing the weight of the transformation to a lower weight, which results in lower obscurity of the program.

Chapter 8

Future work

This chapter discusses improvements and future work for each transformation.

8.1 Improvements of obfuscations

There is always room for improvement, and several improvements can be made to enhance the quality of each obfuscating transformation. These improvements are presented in this section.

8.1.1 Outlining conditions

The greatest improvement that can be made is to make the Outlining conditions transformation more general. At the moment, the transformation can only handle conditions of type Integer. This means that conditions of other types will not be obfuscated, which leads to lower level of obscurity. If the Outlining conditions transformation could handle types such as long, float, double and String, the quality for the transformation would be higher.

Another improvement would be if the transformation could be applied in the constructor and in the main method and still guarantee that the transformed program is semantically equivalent to the original program.

The quality of the transformation would also increase if the conditions could be more obscured.

8.1.2 Array restructuring

There are several improvements that can be made in the Array restructuring transformation. The biggest problem in this study was that the transformation was only

applied 4 times on the test program, which is not a good result. If the transformation was able to transform more types such as long, float, double and String, the transformation would be more general and perform better.

As for the Outlining transformation in section 8.1.1, the transformation would be improved if the transformation could be applied in the constructor and in the main method, which would provide a better coverage in terms of obfuscation. The quality of the transformation would also be improved if the transformation could be applied on fields.

8.1.3 Variable restructuring

The Variable restructuring transformation can be improved by making the transformation more general as described above in the Outlining condition and Array restructuring sections in this chapter. Also the transformation could be improved if the transformation could be applied in the constructor and in the main method and on fields as described in the section 8.1.2.

As can be seen in table 5.10 in section 5.3, there is a big difference between the total number of lines in the original program compared in the transformed program. An improvement would be to try reducing the length of the transformed program by minimizing the Variable restructuring transformation's output. As the transformation works now, an integer of length n will increase the code length with $n \cdot 7 + 6$ new lines. A reduction of the code length would result in a better overall cost score, which would increase the quality of the obfuscating transformation.

8.2 New obfuscations

The best way to improve the level of obfuscation is to combine different transformations, as discussed in section 6.4.4 in chapter 6. JBCO already consist of many different obfuscating transformations, there are still areas within bytecode and the JVM which JBCO does not cover.

Batchelder [1] discussed new transformations and transformations which can be further obscured. He discussed transformations such as the thread locking mechanism of Java and embedding constant values in fields, packing local variables into bitfileds and converting arithmetic expressions to bit-shifting operations could be more obscured with the addition of opaque predicates.

Batchelder also discussed improving obfuscation-point decisions through the static detection of hot spots. A hot spot is a region of a program where a high proportion of executed instructions occur or where most time is spent during the program's

execution. Instead of applying obfuscating transformation on a whole program, the transformation could be applied only on the hot spots in the program, which can improve the quality of an obfuscated program.

Combining these new obfuscating transformations with already existing transformation will hopefully give a high level of obscurity to the program.

8.3 Improvements of analysis methods

One way of improving the accuracy of the overall quality score for each obfuscating transformation is to use participants instead of decompilers to calculate the resilience score. It would require numerous participants and the goal for each participant would be to reverse engineer obfuscated code. A participant's perception for how difficult it was could be graded and act as the result. A survey like this would give a better understanding of the reverse engineering process and yield a more justified resilience score. Unfortunately, this type of survey is not feasible since it requires a lot of work.

8.4 Final words

It is certain to say that code obfuscation is a good supplement to make the reverse engineering process more difficult. It is especially important to protect Java source code, because it normally contains enough information to permit type checking.

A drawback of code obfuscation is that the efficiency of the program often is reduced, at least when it comes to the three transformations presented in this thesis. Some might argue that the efficiency of a program is much more important than the obscurity of a program. Usually it is about finding the balance between these two parameters. The balance is most likely determined by the program. Factors such as the program's main purpose, how the program will be used, and which limitations that exist; these are factors that have a significant impact on how the balance is divided between efficiency and obscurity.

Finding the right balance can be hard but in order for a company to stay strong against competitors, code obfuscation can be the answer to the problem.

Bibliography

[1] Michael R. Batchelder. Java bytecode obfuscation. Master's thesis, McGill University, Montréal, 2007.

[2] Fredrick Cohen. Operating system protection through program evolution. Technical report, 1992. Accessed online 2013-05-20.

[3] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, 1997. Accessed online 2013-05-17.

[4] Eldad Eilam. *Reversing: Secrets of Reverse Engineering.* Wiley Publishing, Inc., 2005.

[5] Árni Einarsson and Janus Dam Nielsen. A survivor's guide to java program analysis with soot. Technical report, Department of Computer Science, 2008. Accessed online 2013-05-17.

[6] The Eclipse Foundation. Eclipse. `http://www.eclipse.org/`, 2013. Accessed online 2013-07-20.

[7] Reto Galli. Mars encryption. `http://reto.orgfree.com/us/projectlinks/MARSReport.html`, 2000. Accessed online 2013-08-07.

[8] Peter Haggar. Java bytecode: Understanding bytecode makes you a better programmer. `http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/`, 2001. Accessed online 2013-05-17.

[9] James Hamiltion and Sebastian Danicic. An evaluation of current java bytecode decompilers. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 129–136, Edmonton, Alberta, Canada, 2009. IEEE Computer Society.

[10] Laurie Hendren and Michael Batchelder. Obfuscating java: The most pain for the least gain*. Technical report, School of Computer Science, 2007. Accessed online 2013-05-17.

[11] Colin Hughes. Project euler. `http://jadretro.sourceforge.net/`. Accessed online 2013-08-07.

[12] Matthew et al. Karnick. A qualitative analysis of java obfuscation. In *10th IASTED International Conference SOFTWARE ENGINEERING AND APPLICATIONS*, pages 166–171, Dallas, TX, USA, 2006.

[13] Eric Lafortune. Proguard. `http://proguard.sourceforge.net/`, 2013. Accessed online 2013-07-11.

[14] Zelix Pty Ltd. Zelix klassmaster - heavy duty protection. `http://www.zelix.com/klassmaster/index.html`, 2012. Accessed online 2013-07-11.

[15] Ivan Maidanski. Jadretro. `http://jadretro.sourceforge.net/`, 2008. Accessed online 2013-08-07.

[16] McGill. Jbco: the java bytecode obfuscator. `http://www.sable.mcgill.ca/JBCO/`, 2012. Accessed online 2013-07-11.

[17] Jonathan Meyer and Daniel Reynaud. Jasmin. `http://jasmin.sourceforge.net/`, 2005. Accessed online 2013-05-17.

[18] Jerome Miecznikowski. New algorithms for a java decompiler and their implementation in soot. Master's thesis, McGill University, Montréal, 2003.

[19] Pravin H. Rane. Java-sourcecode. `http://jadretro.sourceforge.net/`. Accessed online 2013-08-07.

[20] TechMetric Research. Java appliction servers report. Technical report, 1999. Accessed online 2013-05-17.

[21] Herbert Schildt. *Java: The Complete Reference*. McGraw-Hill, 2011.

[22] Hank Shiffman. Boosting java performance: Native code and jit compilers. `http://www.disordered.org/Java-JIT.html`, 1996. Accessed online 2013-05-17.

[23] Bill Venners. Bytecode basics - a first look at the bytecodes of the java virtual machine. `http://www.javaworld.com/javaworld/jw-09-1996/jw-09-bytecodes.html`, 1996. Accessed online 2013-05-17.

[24] Peter J.L. Wallis. *Portable Programming*. John Wiley and Sons Inc, 1982.

[25] YourKit. Yourkit java profiler. `http://www.yourkit.com/`. Accessed online 2013-08-23.

[26] yWorks. yguard - java bytecode obfuscator and shrinker. `http://www.yworks.com/en/products_yguard_about.htm`, 2013. Accessed online 2013-07-11.