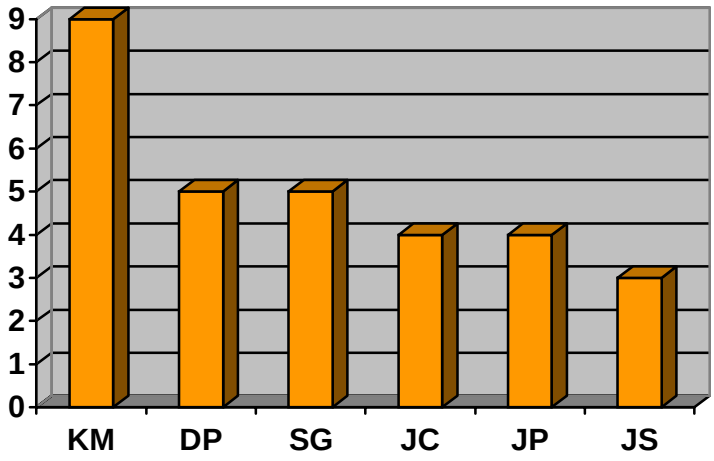
A comparative survey of Java obfuscators available on the Internet



415.780 Project Report

(February 22, 2001)

Author: Hongying Lai

Supervisor: Professor Clark Thomborson

Computer Science Department

The University of Auckland

CONTENTS

Figures, Tables and Programs:

Abstract

The proliferation of decompilers [6] and the rapidly increasing use of Java have triggered a need for Java protectors, obfuscators. With so many obfuscators available on the Internet today, how can one know which is better? This project focuses on a comparative survey of Java obfuscators available on the Internet. Here, I have surveyed 13 obfuscators from the Internet. I used them to obfuscate both my benchmarks and “Logic program” from UCLA’s Philosophy department in America. I also tried decompiling these obfuscated classes by a Java decompiler, SourceAgain. Finally, I used my obfuscation metrics to analyze and compare the above results. I found that various obfuscators provide different security level options to protect Java programs. Many obfuscators work well, but some are not effective.

Acknowledgements

First of all, I would like to thank my supervisor, Professor Clark Thomborson. He has helped me to understand the techniques in my project and performance analysis in evaluating software. He also gave me good advice on data organization and data presentation.

Secondly, I would like to thank Professor David Kaplan who directs the Logic Software project at Philosophy department of UCLA in America and Rob Johnson who is the Author of the Logic Program. They provided me Logic Program and gave me a good opportunity to apply obfuscators into actual work.

Finally, I would like to thank my family for their encouragement and support. Special thanks are given to my husband who helped me set up the experimental environment.

Introduction

It is a well known fact that Java Class files [8] are easily reverse-engineered because Java bytecode [9] contains a lot of the same information as its original source code does. In addition, Java programs, like Mobile codes, have a good reputation as to " write once, run everywhere". This flexibility has many potential advantages in a distributed environment. However, this distributed form increases the risk of malicious reverse engineering [6] attack. Therefore Java code is easy to decompile.

Code obfuscation [1], as a newborn security technique, is currently one of the best protection tools for Java code from reverse engineering. It renders software unintelligible but still functionally equivalent to the original code. It makes programs more difficult to understand, so that it is more resistant to reverse engineering.

Obfuscators [1], which automatically transform a program by code obfuscation, are more and more available on the Internet by the increasing requirement. Various obfuscators provide different security levels to protect Java programs.

1.1 Project Objective

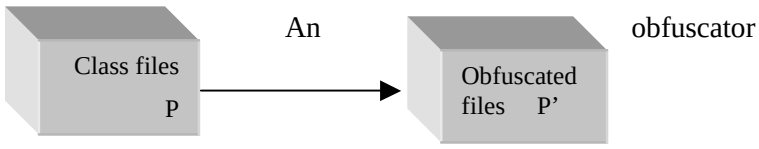
The purpose of the project is firstly to understand the techniques of the obfuscation. Secondly, it is to assess the quality of an obfuscator, and to find out which part of the obfuscation techniques has been applied into the obfuscators available on the Internet. Finally, it is looking at possible techniques which will be used for the future obfuscators and what needs to be improved in my future Obfuscation Metrics.

1.2 General Discussion of Obfuscation and Obfuscators

In above, I have simply described the concept about code obfuscation and an obfuscator. In order to more clearly know about what the obfuscation is and what an obfuscator actually does, I am going to use two examples to discuss them.

Example1:

In Figure 1_1, a set of class files P, through an obfuscator, becomes other set of class files P'. The result is the code of P is not equal to the code of P' and P.code is more different understood than the P'.code, but both function same.



P.code <> P'.code, but P.function = P'.function

Figure 1_1: Obfuscation process.

The following example 2 shows the processes of a simple small Java class file that is obfuscated by an obfuscator, KlassMaster v1.1.

Example 2:

Original Source:

```
class Hello{
   public Hello(){
       int number=1;
   }
   public String getHello(String helloname){
       return helloname;
}
```

Original Bytecode:

```
Compiled from TestOBF.java
class Hello extends java.lang.Object {
    public Hello();
    public java.lang.String getHello(java.lang.String);
```

```
}

Method Hello()
   0 aload_0
   1 invokespecial #4 <Method java.lang.Object()>
   4 iconst_1
   5 istore_1
   6 return

Line numbers for method Hello()
   line 29: 0
   line 30: 4
   line 29: 6

Method java.lang.String getHello(java.lang.String)
   0 aload_1
   1 areturn

Line numbers for method java.lang.String
getHello(java.lang.String)
   line 33: 0
```

After obfuscated by KlassMaster obfuscator, the names in this class had been change. See the following code.

Obfuscated bytecode:

```
Compiled from a.java
class a extends java.lang.Object {
    public static boolean a;
    public a();
    public java.lang.String a(java.lang.String);
}
Method a()
   0 aload_0
   1 invokespecial #4 <Method java.lang.Object()>
   4 iconst_1
   5 istore_1
   6 return
Method java.lang.String a(java.lang.String)
   0 aload_1
   1 areturn
```

From above, we can see that Hello.class has been changed to a.class by obfuscator KlassMaster and their method getHello(java.lang.String) is altered to a(java.lang.String). The method name, “a”, is less understandable than “getHello”.

Comparing the obfuscated bytecode with original bytecode, we can also see that the line numbers had been removed from the Obfuscated bytecode. This gives less information to reverse engineering.

Example2 is a very simple example about code obfuscation. It just scrambles identifiers and removes the line numbers that are generated by compilers. More detail and complicated obfuscated codes can be found in section 6.

1.3 Documentation Organization

This report is organized into 7 sections as indicated.

Section 2 overviews the obfuscator techniques that will be a basic idea for my obfuscation metrics design.

Section 3 introduces some related research that I have learnt from the Internet.

Section 4 explains my obfuscation metric design.

Section 5 presents details about my benchmark design.

Section 6 evaluates 13 obfuscators currently available on the Internet. In my experimental design, my benchmarks and "Logic program" are obfuscated by each obfuscator and the obfuscated files are decompiled by SourceAgain. Finally, the obfuscators are assessed by my metrics and the outputs.

Section 7 displays the comparative results by my obfuscation metrics.

Section 8 discusses the techniques of the obfuscators and the possible improve in the future research.

Overview of Obfuscator Techniques

An obfuscator is a program that applies the techniques of the obfuscating transformation to the program code. Collberg et al. [1] classify the obfuscating transformation into three catalogues: lexical transformations, control transformations and data transformations. Relatively, we can sort the obfuscation techniques into three groups: Layout Obfuscations, Control Obfuscations and Data Obfuscations. See Figure 2_1.

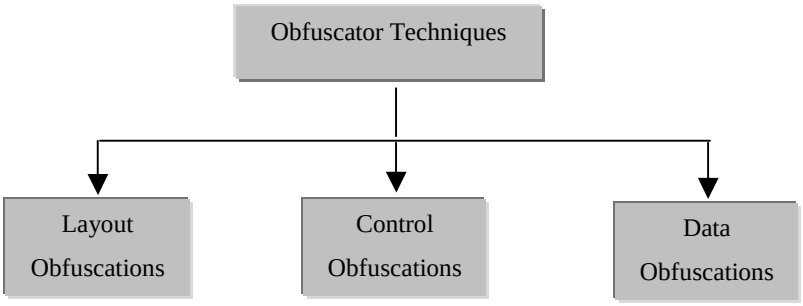


Figure 2_1: Classes of Obfuscations

2.1 Layout Obfuscations

Layout Obfuscations modify the layout structure of the program by two basic methods: renaming identifiers and removing debugging information. They make the program code less information to a reverse engineer. Most Layout Obfuscations cannot be undone because they use one-way function such as changing identifiers by random symbols and removing comments, unused methods and debugging information. Though Layout Obfuscations cannot prevent reverse engineers to understand the program by observing the obfuscated code, they at least consume the cost of reverse engineering.

Layout Obfuscations are the most well studied and widely used field. Almost all Java obfuscators contain this technique.

2.2 Control Obfuscations

Control Obfuscations change the control flow of the program. Collberg et al. introduced an approach to control-altering transformations by opaque predicates [1][3]. These opaque predicates can be inserted to the control flow structure by following three approaches. See Figure 2_1.

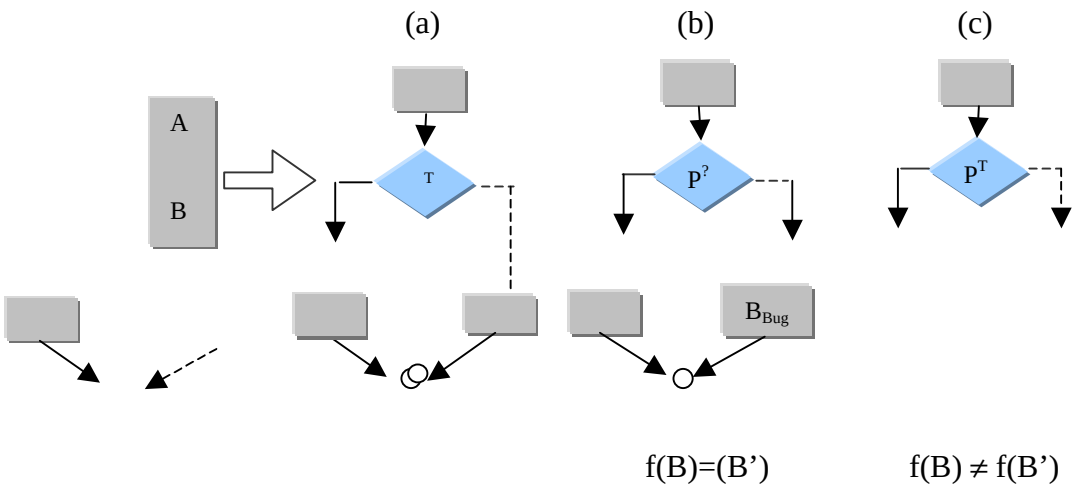


Figure 2_2: Control Obfuscation by opaque predicate insertion

In Figure 2_2(a), the block [A;B] is split up by inserting an opaquely true predicate P^T which makes it appear as if B is only executed some times. In Figure 2_2(b), B is split into two different obfuscated versions B and B'. The opaque predicate $P^?$ selects either of them at runtime. In Figure 2_2(c), P^T always selects B over B_{Bug}, a buggy version of B.

Many Control Obfuscations are similar to these in Figure 2_2.

Douglas Low [4] classifies the control flow obfuscations into three catalogues as following:

1. Control aggregation obfuscations, which change the way in which program statements are grouped together,
2. Control ordering obfuscations, which alters the executed statement order
3. Control computation obfuscations, which hide the real control flow in a program.

These control obfuscations also mainly depend on people's predicates.
The code obfuscated by Control Obfuscations usually cannot be directly decompiled back to the Java language. This makes it more difficult for reverse engineers to decompile codes.

2.3 Data Obfuscations

Data Obfuscations break the data structures used in the program and encrypt literal. Collberg et al. [2] also introduced several approaches to obfuscating data structures such as:

- Modify inheritance relations. For example, producing more classes by extending the inheritance hierarchy tree or false refactoring;
- Restructure arrays. For example, increasing or decreasing the number of an array's dimensions by folding or flattening.
- Clone methods. For example, creating several different versions of a method by applying different sets of obfuscating transformations to the original code.
- Variable Splitting. For example, splitting a boolean variable into two or more variables;

Data Obfuscations change data structure of a program thoroughly. They makes the obfuscated codes so complicated that it is impossible to recreate the original source code.

Related Work

From the Internet, I have found that there already have had examples of evaluating obfuscators.

For examples, in a 1997 “Java tip”, Protect your bytecodes from reverse Engineering/ Decompilation [5], Qusay H. Mahmoud evaluated an obfuscator, Crema and a decompiler, Mocha to learn how Crema obfuscator can protect Java code from decompilers such as Mocha. He also used a simple program as an example to explain it in detail. Firstly, he decompiled the program by Mocha and showed that the decompiled source code was easy to read and understand and still functioned. Then he obfuscated the original program by Crema and decompiled the obfuscated code again. The result is that Mocha had to give up with a java.lang.NullPointerException error. Finally, He concluded that Crema obfuscator could protect Java codes from the Mocha decompiler.

Similarly, in a 1998 article “Java Decompilation and Reverse Engineering Part2”, Benoit Marchal and Meurrens[6] evaluated six decompilers and three obfuscators to study decompilation of Java classes as a means to learn how to protect applications by obfuscators.

Here are the six decompilers they used:

1. Mocha, the original Java decompiler by Hanpetr van Vliet, beta (summer 1996);
2. Jasmine a patch to Mocha by Tang Hogbo. V1.10 (Jan. 25,1998), SourceTec;
3. WingDis v2.12 by Kathy Ho, (Dec. 13,1997), v2.12, Wingsoft;
4. DeJaVu the decompiler of OEW 2.0.3 (Oct. 2, 1997), v1.0 (1996), Innovative Software;
5. SourceAgain by Paul J. Martino and Grigori Humphreys, v1.10 beta 3.a (Feb. 16, 1998), Ahpah;
6. Jad by Pavel Kouzenetsov, v1.5.3.2 (Feb. 28,1998).

Here are the three obfuscators:

Neil Aggarwal’s Obfuscate;

Zelix KalssMaster;

HoseMocha.

They used the similar way as Qusay H. Mahmoud did: decompiling the original code, obfuscating the original code, decompiling the obfuscated code and finally comparing their results. From the results, they claimed that:

- All the six decompilers could process the code obfuscated by Neil Aggarwal's Obfuscate with its medium mode, but the resulting code was less readable. The code obfuscated by Neil Aggarwal's Obfuscate with its high mode could crash early decompilers such as Mocha. Unfortunately, the classes no longer pass the Java verifier.
- The code obfuscated by Zelix KlassMaster could crash two of the decompilers, Jad and SourceAgain. And the classes are accepted by the verifier.
- The code obfuscated by HoseMocha broke most of the decompilers except Jad and SourceAgain.

At the end, they gave a conclusion "… obfuscators, even if they are unable to prevent decompilation of verifiable classes, can greatly affect the usability of decompilers."

These pervious works used decompilers to measure the security of obfuscators. It is the best and direct way to show how powerful an obfuscator is. So in my project, I also used this approach to one of my obfuscation individual metrics which measure the 13 obfuscators I had collected from the Internet.

Obfuscation Metrics Design

The Obfuscation Metrics are designed according to the theory of obfuscation transformations and the techniques used in the obfuscators available on the Internet. They are also concerned about the actual use of an obfuscator. For example, whether can the obfuscator successfully obfuscate codes? And also whether can these obfuscated codes be decompiled? .

In Section 2, we learn that the obfuscation techniques are classified into three catalogues: Layout obfuscations, Data obfuscations and Control obfuscations. I attempt to apply these theoretical metrics to my practical measure work. Therefore my obfuscator metrics contain these three parts. Furthermore, I added other three parts into my metrics. They are “Decompiling Logic Program”, “Decompiling Obfuscated Benchmark” and “Obfuscating Logic Program”. See table 4_1.

Table 4_1: Obfuscation Metrics

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	1
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		1
	Data Unstructured		1
Control Obfuscation	Change “if…else”, “for” or “while” loop		1
Obfuscating Logic Program			1
Decompiling Obfuscated Benchmarks			1
Decompiling Obfuscated Logic Program			1
Total Score			10

The first part, Layout obfuscations, includes " Scramble Names" which contain modifying public names, protected names and private names and "Remove Line numbers" which deletes the Line numbers generated by a compiler.

The second part, Data obfuscations, contains "Encrypt String" and " Data Unstructured" such as increasing the number of an array's dimensions by folding it.

The third part is Control obfuscations, control-altering transformation through opaque predicates, such as changing “if...else” statements, “while” loop and “for” loop constructs.

The fourth part is “Obfuscating Logic Program”. It is to measure whether an obfuscator can successfully obfuscate the complicated programs, the “Logic Program”.

The fifth part is “Decompiling Obfuscated Benchmarks”. It is to measure whether the obfuscated Benchmark (see Section 5) bytecodes could be decompiled by the decompiler, SourceAgain.

The last part is “Decompiling Obfuscated Logic Program”. It is to measure whether the obfuscated “Logic Program” (see Section 6, 6.3) bytecodes could be decompiled by the decompiler, SourceAgain.

I chose these individual metrics because I could score them objectively, and they spanned all the classes of obfuscations and are possibly found in the obfuscators.

The total number of points of the metrics is 10. There is one point for each individual metric. I give zero point for an individual metric if I observe no obfuscation behavior on this metric in the obfuscated output. For example, on the “Scramble names” metrics, “getArray” in the original Benchmark1is a private method name, after being obfuscated, in the obfuscated output, it still reads “getArray”, so I will assign “0” to the individual metric “Scramble Private name”. However, if it is changed to other name, such as “a”, this metric will be scored one point. In a specific case, if just only parts of private names are changed, I will still score “1” for this metric. This rule is also available for the other “Scramble names” metrics such as “Public” and “protected”.

On the metric, “Remove Line numbers”, if I find there is no “Line numbers” in the obfuscated bytecode, I will score “1”, or I will give “0” to this metric.

On the metric, “Encrypt String”, if a string name in the obfuscated codes has been encrypted, the metric will be assigned “1”, or it will be scored “0”.

On the metric, “Folding one dimension array”, if an one dimension array in the obfuscated code has been change to two or more dimension arrays, I will score the metric “1”, or I will score it “0”.

On the metric, “Change “if…else”, “for” or “while” loop”, if I find one of the construct, “if…else”, “for” loop or “while” loop has been changed, I will score the metric “1”, or I will assign it “0”

On the metric, “Obfuscating Logic Program”, if an obfuscator can successfully obfuscate the “Logic Program” and still function equivalent to the original code, I will score “1” for the metric, or I will assign “0” to it.

On the metric, “Decompile Obfuscated Benchmarks”, if the Benchmarks, which consist of Benchmark1 and Benchmark2, are obfuscated by an obfuscator successfully, and if one of the Benchmarks obfuscated by an obfuscator can not be decompiled, or one set of the decompiled codes does not function equivalent to the original code, I will score “1” for the metric. Other any cases, I will assign “0” to the metric.

On the metric, “Decompiling Obfuscated Logic Program”, if one of the obfuscated Logic Program files cannot be decompiled, I will score “1” for the metric. In any other cases, I will assign “0” to it.

Benchmark Design

Environment:

Window 98

Personal computer

Tools:

JDK1.2.2

Java decompiler, SourceAgain (TM) Personal v1.10h

The Benchmarks are designed to match my Obfuscation Metrics. There are two benchmarks, Benchmark1 and Benchmark2. They are used to measure different metrics. Benchmark1 measures obfuscators' behaviors on Layout Obfuscations and Data Obfuscations. Benchmark2 tests obfuscators' behaviors on Control Obfuscation. In order to simplify the observation and make the metrics to be scored easily, I listed the specific objects from my Benchmarks for my Obfuscation Metrics. See the following Table 5_1.

In Table 5_1, I selected 7 individual metrics from my Obfuscation Metrics. For each individual metric, I chose some specific objects from Benchmark1, which contains TestOBF.class and Hello.class, and Benchmark2, which conations TestControl.class. For example,

- On the individual metrics, "Public Class name" and "Public Method name", I selected public class name "Hello" and public method name "getHello" from Benchmark1 Hello.class for them. I chose public field name "hello" from Benchmark1 TestOBF.class for the metric "Public Class name".
- On the metrics, "Protected Method name" and "Protected Field name", I chose "getGood" and "good" from TestOBF.class for them.
- On the metrics, "Private Method name" and "Private Field name", I chose "getArray" and "number" from TestOBF.class for them.
- On the metrics "Remove Line numbers", "Encrypt String", "Unstructured data", separately I selected "Line numbers", String "GOOD" and one dimension array "number[]" to match them. These tested objects are all from TestOBF.class.

- On the metric, "If…else, For/while loop" I chose tested object from Benchmark2 TestControl.class.

Table 5_1: The original names of the objects under observation in my Benchmark for Obfuscation Metrics

Metrics		**Benchmark1**		**Benchmark2**
		`TestOBF.class`	`Hello.class`	`TestControl.class`
Public	Class name Method name Field name	 hello (instance of class Hello)	Hello getHello	
Protected	Method name Field name	getGood good		
Private	Method name Field name	getArray number		
Remove Line numbers		Line numbers		
Encrypt String		GOOD		
Data Unstructured		One dimension array: number[]		
Change "If…else", "For" loop or "while" loop				If … else For… / While …

Note: Case-sensitive to the public names, protected names, private names and String names.
Blank means that this item is not available.

The Table 5_1 will be used as a concrete standard in Section 6 to measure the obfuscators. The following results are also prepared for the next section, Section 6. Here's how I did for my Benchmark design:

- First of all, I wrote Java code, Benchmarks.

- Secondly, I compiled them with JDK1.2.2 compiler and run the resulting class files to make sure that they function properly. Then I printed their bytecodes by running javap with the c (disassemble), p (include private fields) and l (line number and local tables).
- Thirdly, I decompiled the compiled class files by the decompiler, SourceAgain and compared them with their original source codes.
- Fourthly, I recompile the decompiled source code and run the new code to test whether the decompiled files function properly.
- Finally, I compared all results from every step so that I could clearly understand what my Benchmarks looked like in each step before they were obfuscated.

The detail is as follows.

5.1 **Benchmark1**

Benchmark1 source file is TestOBF.java (911 bytes) and it has two classes, TestOBF.class (1027 bytes) and Hello.class(301 bytes). TestOBF.class is a main class. From my experience, a main class name usually does not need to be obfuscated because it is an important connector to outside. Therefore I designed the other class, Hello.class, for the "Public Class name" and "Public Method name" metrics.

The detail follows:

5.1.1 Benchmark1 Source

The following Program 5_1 shows the Benchmark1 Source code I designed.

Program 5_1: Benchmark1 Source Code

```
/* BenchMark1: TestOBF.java contains two classes
 *             class TestOBF and class Hello.
 * Purpose:    test obfuscators' behaviors on
 *             Layout Obfuscation.
 *                and Data Obfuscations.
 * @date:      14/12/2000
 * @author:    Hongying Lai
 */
public class TestOBF{
    public Hello hello=new Hello();
    protected String good="";
    private int[] number = new int[10];
    public static void main(String args[]){
        new TestOBF();
    }
    public TestOBF(){
```

```
            System.out.println(hello.getHello("HELLO"));
            System.out.println(getGood());
            System.out.println("number="+getArray());
        }
        protected String getGood(){
            good="GOOD";
            return good;
        }
        private int getArray(){
            number[0]=1;
            return number[0];
        }
}

class Hello{
        public Hello(){
            int number=1;
        }
        public String getHello(String helloname){
            return helloname;
        }
}
```

Compile TestOBF.java with JDK1.2.2:

prompt>javac TestOBF.java

Get two classes: TestOBF.class and Hello.class.

These two classes are going to be obfuscated by the obfuscators in Section 6.

Run them to make sure that they function properly:

prompt >java TestOBF

Get result:

```
HELLO
GOOD
number=1
```

Therefore, Benchmark1 function properly before obfuscated.

5.1.2 Benchmark1 Bytecode

Run javap to get bytecode from TestOBF:

prompt>javap -c -p -l TestOBF

See the bytecode listing in Program 5_2 below:

Program 5_2: Benchmark1 Benchmark1 Bytecode_TestOBF

```
Compiled from TestOBF.java
public class TestOBF extends java.lang.Object {
    public Hello hello;
    protected java.lang.String good;
    private int number[];
    public TestOBF();
    private int getArray();
    protected java.lang.String getGood();
    public static void main(java.lang.String[]);
}

Method TestOBF()
   0 aload_0
   1 invokespecial #13 <Method java.lang.Object()>
   4 aload_0
   5 new #5 <Class Hello>
   8 dup
   9 invokespecial #11 <Method Hello()>
  12 putfield #20 <Field Hello hello>
  15 aload_0
  16 ldc #1 <String "">
  18 putfield #19 <Field java.lang.String good>
  21 aload_0
  22 bipush 10
  24 newarray int
  26 putfield #21 <Field int number[]>
  29 getstatic #22 <Field java.io.PrintStream out>
  32 aload_0
  33 getfield #20 <Field Hello hello>
  36 ldc #3 <String "HELLO">
  38 invokevirtual #18 <Method java.lang.String
getHello(java.lang.String)>
  41 invokevirtual #23 <Method void println(java.lang.String)>
  44 getstatic #22 <Field java.io.PrintStream out>
  47 aload_0
  48 invokevirtual #17 <Method java.lang.String getGood()>
  51 invokevirtual #23 <Method void println(java.lang.String)>
  54 getstatic #22 <Field java.io.PrintStream out>
  57 new #9 <Class java.lang.StringBuffer>
  60 dup
  61 ldc #4 <String "number=">
  63 invokespecial #14 <Method
java.lang.StringBuffer(java.lang.String)>
  66 aload_0
  67 invokespecial #16 <Method int getArray()>
  70 invokevirtual #15 <Method java.lang.StringBuffer
append(int)>
  73 invokevirtual #24 <Method java.lang.String toString()>
  76 invokevirtual #23 <Method void println(java.lang.String)>
  79 return

Line numbers for method TestOBF()
   line 14: 0
   line 8: 4
   line 9: 15
   line 10: 21
   line 15: 29
   line 16: 44
   line 17: 54
```

```
   line 14: 79

Method int getArray()
   0 aload_0
   1 getfield #21 <Field int number[]>
   4 iconst_0
   5 iconst_1
   6 iastore
   7 aload_0
   8 getfield #21 <Field int number[]>
  11 iconst_0
  12 iaload
  13 ireturn

Line numbers for method int getArray()
   line 24: 0
   line 25: 7

Method java.lang.String getGood()
   0 aload_0
   1 ldc #2 <String "GOOD">
   3 putfield #19 <Field java.lang.String good>
   6 aload_0
   7 getfield #19 <Field java.lang.String good>
  10 areturn

Line numbers for method java.lang.String getGood()
   line 20: 0
   line 21: 6

Method void main(java.lang.String[])
   0 new #6 <Class TestOBF>
   3 invokespecial #12 <Method TestOBF()>
   6 return

Line numbers for method void main(java.lang.String[])
   line 12: 0
   line 11: 6
```

Run javap to get bytecode from Hello.class:

prompt >javap -c -p -l Hello

See the bytecode listing in Program 5_3 below:

Program 5_3: Benchmark1 Benchmark1 Bytecode_Hello

```
Compiled from TestOBF.java
class Hello extends java.lang.Object {
    public Hello();
    public java.lang.String getHello(java.lang.String);
}
Method Hello()
   0 aload_0
   1 invokespecial #4 <Method java.lang.Object()>
   4 iconst_1
   5 istore_1
   6 return
Line numbers for method Hello()
```

```
   line 29: 0
   line 30: 4
   line 29: 6
Method java.lang.String getHello(java.lang.String)
   0 aload_1
   1 areturn
Line numbers for method java.lang.String
getHello(java.lang.String)
   line 33: 0
```

From above bytecode, we can see that the class names, method names and field names are the same as those in its source. The bytecode contains Line Numbers generated by a compiler.

5.1.3 Decompiled Benchmark1

Decompile the above classes, TestOBF.class and Hello.class:

In the command line, type:

prompt >srcagain *.class

Get the output: _TestOBF.java , _Hello.java

See the decompiled codes listing in Program 5_4 and Program 5_5 below:

Program 5_4: Decompiled Benchmark1 Code_TestOBF

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//
import java.io.PrintStream;
public class TestOBF {
    public TestOBF()
    {
        System.out.println( hello.getHello( "HELLO" ) );
        System.out.println( getGood() );
        System.out.println( "number=" + getArray() );
    }
    public Hello hello = new Hello();
    protected String good = "";
    private int[] number = new int[10];
    private int getArray()
    {
        number[0] = 1;
        return number[0];
    }
    protected String getGood()
    {
        good = "GOOD";
        return good;
    }
    public static void main(String[] String_1darray1)
    {
        new TestOBF();
    }
```

```
}
```

Program 5_5: Decompiled Benchmark1 Code_Hello

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//
class Hello {
    public Hello()
    {
          Int int1 = 1;
    }
    public String getHello(String String1)
    {
        return String1;
    }
}
```

Comparing the Decompiled Benchmark1 codes (Program 5_4 and Program 5_5) with their original ones (Program 5_1), we can see that these codes are almost the same.

Recompile them to test the function of the Decompiled Benchmark1:

- Rename _TestOBF.java and _Hello.java as TestOBF.java and Hello.java
- Compile them and run the compiled code

Get result:

```
HELLO
GOOD
number=1
```

Therefore the decompiled Benchmark codes run the same function as their original ones.

5.1.4 Summary:

- Before being obfuscated, Benchmark1 function and can be decompiled successfully.
- Comparing Benchmark1 Source with BenchMark1 Bytecode, both have the same identifier names. There are Line numbers in Benchmark1 Bytecode.
- Comparing Benchmark1 Source with Decompiled Benchmark1 Source, they are very similar. The comments in decompiled Benchmark1 Source are missed.
- The decompiled Benchmark1 codes run the same function as their original ones.

5.2 **Benchmark2**

Benchmark2 was designed for measuring the only obfuscators who advertised that they could obfuscate codes by changing “if…else”, “for” loop or “while” loop construct. Because of time limitation, I do not attempt to measure all the obfuscators by it. I will leave it to my future work.

Benchmark2 source file is TestControl.java (608 bytes) and its class file is TestControl.class (719 bytes). The detail follows:

5.2.1 Benchmark2 Source

The following Program 5_6 shows the Benchmark2 Source code I designed.

Program 5_6: Benchmark2 Source Code

```
/* BenchMark2: TestControl.java
 *   Purpose:          test   obfuscators'   behaviors   on   Control
Obfuscations.
 * @date:       14/12/2000
 * @author:     Hongying Lai
 */
public class TestControl{
       public static void main( String args[] ){
      int loopNum=0;
      // test for loop construct
      for ( int i=0; i<2; i++ )
            loopNum++;
      // test if.. else statement
      if ( loopNum>1 )
            System.out.print( " loopNum>1" );
      else
            System.out.println( " loopNum<=1" );
      // test whilt loop construct
      while ( loopNum>1 )
            loopNum=loopNum-1;
            System.out.println( " loopNum="+loopNum );

      }
}
```

Compile TestControl.java with JDK1.2.2:

prompt >javac TestControl.java

Get a class: TestControl.class

This class is going to be obfuscated by the obfuscators.

Run it and get the result:

```
loopNum>1 loopNum=1
```

Therefore, Benchmark2 functions before obfuscated.

5.2.2 Benchmark2 Bytecode

Run javap with the c and p options to get bytecode.

prompt>javap -c -p -l TestControl

See the bytecode listing in Program 5_7 below:

Program 5_7: Benchmark2 Bytecode

```
Compiled from TestControl.java
public class TestControl extends java.lang.Object {
    public TestControl();
    public static void main(java.lang.String[]);
}

Method TestControl()
   0 aload_0
   1 invokespecial #9 <Method java.lang.Object()>
   4 return

Line numbers for method TestControl()
   line 7: 0

Method void main(java.lang.String[])
   0 iconst_0
   1 istore_1
   2 iconst_0
   3 istore_2
   4 goto 13
   7 iinc 1 1
  10 iinc 2 1
  13 iload_2
  14 iconst_2
  15 if_icmplt 7
  18 iload_1
  19 iconst_1
  20 if_icmple 34
  23 getstatic #12 <Field java.io.PrintStream out>
  26 ldc #3 <String " loopNum>1">
  28 invokevirtual #13 <Method void print(java.lang.String)>
  31 goto 49
  34 getstatic #12 <Field java.io.PrintStream out>
  37 ldc #1 <String " loopNum<=1">
  39 invokevirtual #14 <Method void println(java.lang.String)>
  42 goto 49
  45 iload_1
  46 iconst_1
  47 isub
  48 istore_1
  49 iload_1
  50 iconst_1
  51 if_icmpgt 45
  54 getstatic #12 <Field java.io.PrintStream out>
  57 new #7 <Class java.lang.StringBuffer>
  60 dup
```

```
  61 ldc #2 <String " loopNum=">
  63 invokespecial #10 <Method
java.lang.StringBuffer(java.lang.String)>
  66 iload_1
  67 invokevirtual #11 <Method java.lang.StringBuffer
append(int)>
  70 invokevirtual #15 <Method java.lang.String toString()>
  73 invokevirtual #14 <Method void println(java.lang.String)>
  76 return

Line numbers for method void main(java.lang.String[])
   line 9: 0
   line 11: 2
   line 12: 7
   line 11: 10
   line 14: 18
   line 15: 23
   line 14: 31
   line 17: 34
   line 19: 42
   line 20: 45
   line 19: 49
   line 21: 54
   line 8: 76
```

From this program, we can see that there are Line Numbers in the Benchmark2 Bytecode.

5.2.3 Decompiled Benchmark2 Source

Type:

prompt>srcagain TestControl.class

Get output: _TestControl.java

See the decompiled code listing in Program 5_8 below:

Program 5_8: Decompiled Benchmark2 Code

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//
import java.io.PrintStream;
public class TestControl {
    public static void main(String[] String_1darray1)
    {
        int int2 = 0;
        int int3;
        for( int3 = 0; int3 < 2; ++int3 )
            ++int2;
        if( int2 > 1 )
            System.out.print( " loopNum>1" );
        else
            System.out.println( " loopNum<=1" );
        while( int2 > 1 )
            --int2;
        System.out.println( " loopNum=" + int2 );
```

```
        }
    }
```

Comparing decompiled Benchmark2 source codes with their original ones, we can see that these decompiled Benchmark2 source codes and their original source are almost the same in control flow structure. Both have the same “for” loop, “if … else” and “while” loop structures and order.

Recompile them to test the function of the Decompiled Benchmark2:

- Rename _TestOBF.java and _Hello.java as TestOBF.java and Hello.java
- Compile them , run the compiled code and get the result:

```
loopNum>1 loopNum=1
```

Therefore the decompiled Benchmark2 codes run the same function as their original one.

5.2.4 Summary

- Before Obfuscated, Benchmark2 functions and can be decompiled successfully.
- There are Line numbers in Benchmark2 Bytecode.
- Comparing Benchmark2 Source with Decompiled Benchmark2 (Program 5_8), both have the same order, “for” loop, “if …else” then “while” loop.
- The Decompiled Benchmark2 codes run the same result as their original one.

Experimental Design

My Experimental Design is to test the quality of the obfuscators and assess my Obfuscation Metrics. As far as I can, I have collected 13 obfuscators from the Internet. For each obfuscator, I did the following five steps: Installation, Synthetic workload, Logic Program, Decompilation by SourceAgain and Assess Metrics. Following Figure 8_1 is the evaluation flowchart for these five steps:

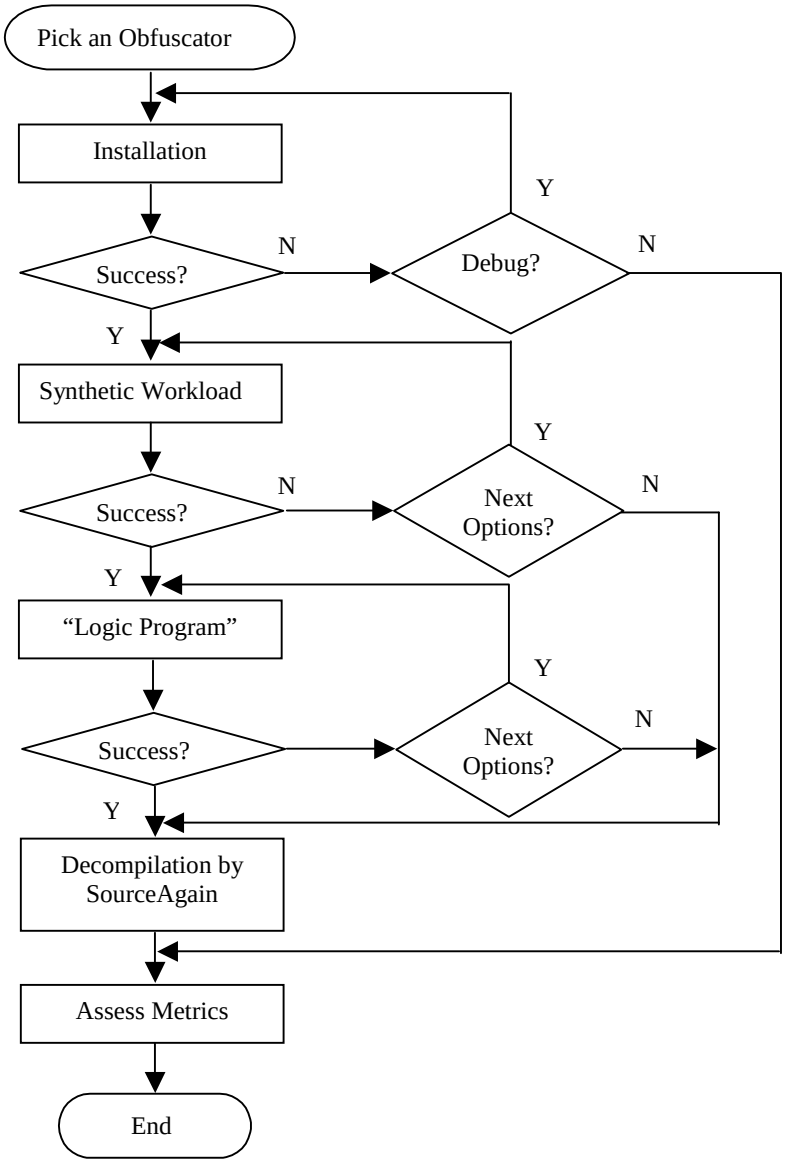


Figure 6_1: Evaluation FlowChart

6.1 Installation

The Installation test is to do the preparation work for the experiment. It is a four-step process, Collection, Download, Installation and Trying Running.

6.1.1 Collection

In this step, I collected the basic data about the obfuscators from the Internet and summed them up in following table:

Table 6_1: The result from the survey of the Obfuscators available on the current Internet.

Obfuscators	Trial	License	Advertised features
2Lkit Obfuscator v1.1	Evaluation	$129	1. Change class, method, field and package names. 2. Removes unused class attributes.
Cloakware			Transforms an executable program's internal data flow, control flow and data structure.
DashO-Pro v2.0	EVALUATION	$895	1. Change class, method and field names. 2. Removes unused class, method and field.
Hashjava	Free	Free	1. Change class, method and field names. 2. Inset confusing debugging information.
JCloak v3.5.3 From Force5	Evaluation	$595.00	1. Change class, method and field names. 2. Removes line numbers and local symbols.
Jproof 1stBarrier MINI v1.1	Evaluation on-line free	$125 ~ $175	Change class names, private method names and private field names.
JOBE – The obfuscator	Free for non Commercial use	a personal: $20.00, a site : $40.	1. Change class, method and field names. 2. Removes debugging information from your java class files.
Jshrinkv1.18	Evaluation		1. Change class, method and field names. 2. Remove unused code, data, and symbolic names from compiled Java class files.
Jzipper v1.08	Evaluation	$35	1. Change class, method and field names. 2. Inset bogus confusing debugging information
Obfuscate™ v1.1	Evaluation		1. Change class, method and field names. 2. Removes debugging information from the java class files.
RetroGuard v1.1	Free	Free	Change public, protected or private names.
SourceGuard 4.0	Evaluation	Enterprise: $1995.00 Professional : $995.00	1. Change class, method and field names. 2. Pruning code
Zelix KlassMaster2.3	Evaluation	Standard: US$299 Small: S$149	1. Change class, method and field names. 2. Remove Line numbers. 3. Obfuscate Control Flow (Limited to one or two methods per class in evaluation version) 4. Encrypt String

The table list order is according to the first letter of the obfuscator names.
The "blank" cell means that there is no this kind information about the obfuscator.

From the Table 6_1, we can learn that 13 Obfuscators were found on the Internet (Some of others might be no longer available such as Crema, some might be not found). Two of the obfuscators are total free and one is free just for non-commercial use. Nine freely provide their evaluations. Cloakware is the only one I could not find the material about its evaluation and product cost. To sum up, there are twelve out of 13 obfuscators can provide their evaluations or their copyright versions freely.

Except for Zelix KlassMaster, other Obfuscators' evaluation versions have the same function as their license versions. The major differences between the Zelix KlassMaster2.3 evaluation version and its commercial version are as follows:

- The evaluation version will flow obfuscate only one or two methods in each class.
- The evaluation version will parse ZKM Script files but will not execute them.
- The evaluation version will not look for or use the `defaultExclude.txt` file.

From their advertisements, most obfuscators can lexically obfuscate the programs such as scrambling the names and removing debugging information. Some also can transform a Java class file by control obfuscation such as obfuscating flow or by data obfuscation such as encrypting String.

6.1.2 Download

From above step, we learnt that there are 12 obfuscators that freely provide their evaluations or copyright versions. However, in this step, when I tried to download these 12 obfuscators, two obfuscators, Hashjava and JOBE-the obfuscator, could not be downloaded. Their links might be out of maintenance. Other ten obfuscators could be downloaded successfully. See table 6_2.

Table 6_2 contains the information about these twelve obfuscators' Version, Downloaded File, Size (in KB), Download Date and URL.

- "Version" shows which product version I downloaded.
- "Download File" presents the names of the files I had downloaded. From the table, we can see that the types of the downloaded files are various. Some of them are "exe", some of them are "zip" and others are "jar".
- "Size" means the download files' sizes. From this table, we could conclude that the download file of the DashO-Pro v2.0 contains the largest size among these

downloaded files. It is 5,226KB. The smallest one is the downloaded file of Obfuscate™, 72KB.

- "Download Date" means the dates when I downloaded these obfuscators.
- "URL" shows the concrete addresses where I downloaded these obfuscators.

Table 6_2: The results of downloading the 12 obfuscators.

Obfuscators	Version	Downloaded File	Size (KB)	Download Date	URL
2Lkit Obfuscator	v1.1	2LkitObf11.exe	666	29/11/2000	www.2Lkit.com/products/2LkitObf
DashO-Pro	V2.0	DASHOPRO_WIN.EXE	5226	12/12/2000	www.preemptive.com
Hashjava				28/11/2000	www.meurrens.org/ip-links/Java/codeEngineering/blackDpen/hashjava.html
JCloak From Force5	V3.5.3	Install.exe	4815	29/11/2000	www.force5.com/pub/Jcloak/install.exe
Jproof 1stBarrier MINI	V1.1	1stBarrier MINI.zip	202	29/11/2000	www.jproof.com
JOBE–Java obfuscator				29/11/2000	www.priment.com
Jshrink	V1.18	Jshrinkinst.exe	100	28/11/2000	www.e-t.com/jshrinkdoc.html
Jzipper	V1.08	Jzip_install_demo.zip	315	28/11/2000	WWW.VEGATECH.NET/JZIPPER
Nei*Aggarwa* Obfuscate™	V1.1	Obfuscate.zip	72	29/11/2000	www.jammconsulting.com/servlets/com.jammconsulting.servlet.JAMMServlet/obfuscateDownloadPage
RetroGuard	V1.1	RetroGuard_v1.1.zip	299	28/11/2000	www.retrologic.com
SourceGuard	V4.0	SG4EvalInstall.jar	3825	28/11/2000	www.4thpass.com/sourceGuard
Zelix KlassMaster	V2.3	ZKMEval.zip	757	28/11/2000	www.zelix.com/klassmaster/

The "blank" cell means that I could not find this information because of unsuccessful download.

6.1.3 Installation

Environment:

JDK1.2.2

Window 98

Personal computer

Installed objects:

Ten obfuscators' files that could be successfully download in above step:

2LkitObf11.exe (*2LKit obfuscator v1.1*), DashoPro_win.exe (*DashO-Pro™ v2.0*), Install.exe (*JCloak v3.5.3*), 1stBarrier MINI.zip (*JProof 1stBarrier MINI v1.1*),

Jshrinkinst.exe (*Jshrink v1.18,Jzipper v1.08*),
Jzip_install_demo.zip (*Nei Aggarwal's Obfuscate™ v1.1*),
RectroGuard_v1.1.zip (*RetroGuard v1.1*) ,SG4EvalInstall.jar (*SourceGuard v4.0*),and ZKMEval.zip (*Zelix KlassMaster v2.3*).

Install: I installed these tested obfuscators by the steps that they provide in their installation document. See these documents that are stored in their download files.

Result:

All ten obfuscators' files could be installed successfully.

6.1.4 Trying Running

Environment:

JDK1.2.2
Window 98
Personal computer

Tested objects:

Ten obfuscators that could be installed successfully:
2LKit obfuscator v1.1, DashO-Pro™ v2.0, JCoak v3.5.3,Jproof 1stBarrier MINI v1.1, Jshrink v1.18, Jzipper v1.08, Nei Aggarwal's Obfuscate™ v1.1, RetroGuard v1.1, SourceGuard v4.0 and Zelix KlassMaster v2.3.

Trying:

In this step, I just simply started these tested obfuscators. If they could start properly, I would report them "Success". Otherwise, I would report them "Failure". When I tried them following their instruction, nine of them could work successfully. The unsuccessful one is *Obfuscate™ v1.1*. I tried it several times. However, it always gave me an error message (See below):

Prompt>java Obfuscate.jar Obfuscate h TestOBF
Exception in thread "main" java.lang.NoClassDefFoundError: Obfuscate/jar

Prompt>java Obfuscate Obfuscate h TestOBF
Exception in thread "main" java.lang.ClassFormatError: Obfuscate (Local variable name has bad constant pool index) ……

Result:

All ten obfuscators were tested in this step. Nine of them could run successfully. One failed.

6.1.5 Summary

Total thirteen Obfuscators had been collected from the Internet. Twelve provide download sites. Ten could be downloaded and installed successfully. Nine passed the Trying Running test. In a word, there were 9 obfuscators could successfully pass the Installation test. See Table 6_3.

Table 6_3: The Summary of Installation Test

Obfuscators	Collection	Download	Installation	Trying Runnin
2Lkit Obfuscator v1.1	Success	Success	Success	Success
Cloakware	Failure			
DashO-Pro v2.0	Success	Success	Success	Success
Hashjava	Success	Failure		
JCloak v3.5.3 From Force5	Success	Success	Success	Success
Jproof 1stBarrier MINI v1.1	Success	Success	Success	Success
JOBE – the obfuscator	Success	Failure		
Jshrinkv1.18	Success	Success	Success	Success
Jzipper v1.08	Success	Success	Success	Success
Obfuscate™ v1.1	Success	Success	Success	Failure
RetroGuard v1.1	Success	Success	Success	Success
SourceGuard 4.0	Success	Success	Success	Success
Zelix KlassMaster2.3	Success	Success	Success	Success

The “blank” cell means that this step is not attempted because a prior step had failed.

In the first step, Collection, Cloakware failed because it did not provide any download sites on the Internet. Therefore I assigned it “Failure” in this step. The others were set “Success”.

In the second step, Download, I gave “Failure” to two obfuscators, Hashjava and JOBE-the obfuscator because I could found download them from the their download sites. The other ten obfuscators, which I could successfully downloaded, were assigned “Success”.

In the third step, Installation, all ten obfuscators, which were “Success” in the second step, were successfully installed. Therefore I gave them “Success”.

In the last step, Trying Running, Obfuscate™ v1.1 run improperly. Therefore I reported it “Failure”. Others could run successfully.

6.2 **Synthetic Workload**

This experiment measures the obfuscators' behaviors on what is known as a "Synthetic Workload" of small Java programs, Benchmark1 and Benchmark2. Benchmark1 is to test the Layout Obfuscation metrics, such as modifying the name, removing debugging information such as Line numbers and data obfuscation metrics such as breaking array, encrypting a string. Benchmark2 is to test the control obfuscation metrics such as changing "if..else" statement, "for" loop or "while" loop construct.

Ten obfuscators, which successfully passed Installation test, were going to be tested in this experiment. There are *2LKit obfuscator v1.1*, *DashO-Pro v2.0, JCloak v3.5.3,JProof 1stBarrier MINI, Jshrink v1.18, Jzipper v1.08, Nei Aggarwal's Obfuscate™ v1.1 RetroGuard v1.1, SourceGuard v4.0* and *Zelix KlassMaster v2.3*.

Here is the process of this test:

```
For each obfuscator o {
    For each benchmark i {
        If I=2 and if obfuscator o not advertises obfuscating control flow{
            Report "N/A"(o,i)
           Try next obfuscator o
           Repeat(o, i)
        }
        Attempt to obfuscate benchmark  i  and with obfuscator o,
        at highest available security options a
        verify that the obfuscated program still function
        if attempt fails, try again with a lower security options a ,
                repeats (o,a,i )
        If attempts fail, report "Failure" or report "Success" (o,a,i)
    }
}
```

Figure 6_2: Evaluation Process for Benchmarks

6.2.1 2LKit Obfuscator v1.1

Test Time: 06/12/2000, 14/12/2000

Platform: Window 98

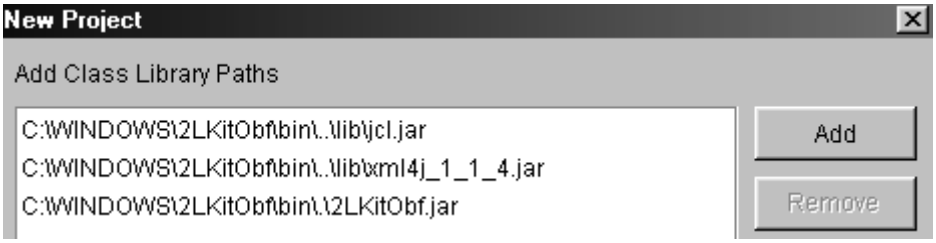
Obfuscating Benchmark1:

Following shows the steps when I obfuscated Benchmark1:

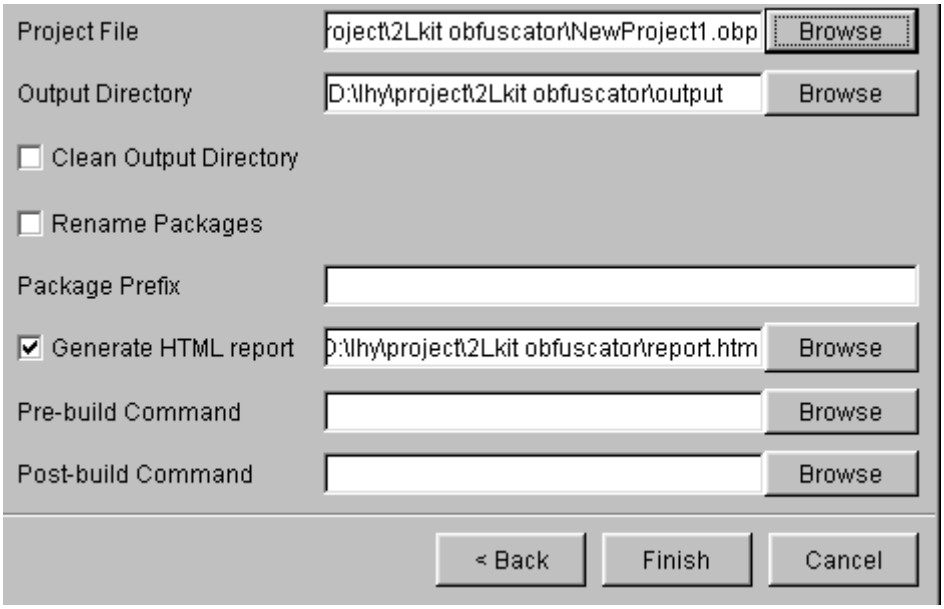
After double clicked “obfuscatorJDK.jar” to start 2LKit Obfuscator, it showed a message:

click “OK”, I choose the path, D:\lhy\project\test source, where my Benchmark1 classes (TestOBF.class and Hello,class) are located.

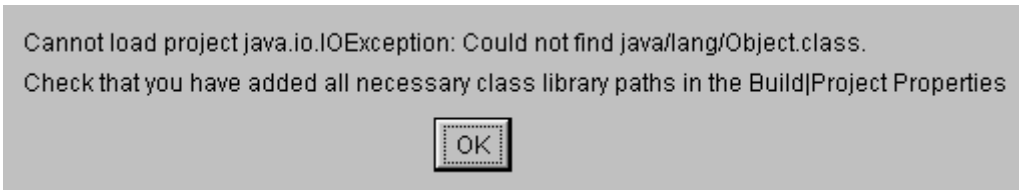
next, came this dialogue:



Then, I selected:



After clicking "finish", finally I got an error message:



I tried to debug it by adding a class path: d:\lhy\project\test Source. I still got the same error message. I tried all approaches that I thought possible. I still the same result. Due to the time limitation for my project, I had to give up.

Result report: Failure

Therefore, *2LKit Obfuscator v1.1* got zero mark for obfuscating my Benchmark1. In other word, it got zero mark on Layout Obfuscation and Data Obfuscation metrics.

Obfuscating Benchmark2:

2LKit Obfuscator v1.1 does not advertise that it can obfuscate programs with control flow (see Table 6_1) such as changing "if…else", "for" loop or "while" loop. So I did not test it and scored "0" for the metric, "Control Obfuscation".

Result report: N/A, not applicable.

Summary:

- *2LKit Obfuscator v1.1* has a friendly, beautiful user interface. However, its "help" information is not detail enough. It is not help me to debug. So I could not successfully obfuscate my Benchmark1 by *2LKit Obfuscator v1.1*.
- It was reported "N/A" for Obfuscating Benchmark2.
- It got zero mark on Layout Obfuscation, Data Obfuscation and Control Obfuscation metrics.

6.2.2 DashO-Pro v2.0

Test Time: 14/12/2000
Platform: Window 98

Obfuscating Benchmark1:

Main steps:

1. Double click "DashOgui.bat" to start DashO-Pro.
2. Input classpath:

    C:\jdk1.2.2\Jre\rt.jar

    C:\DashOPro\DashPro.jar

    C:\DashOPro\Jh.jar

    D:\lhy\project\test source

Note: "D:\lhy\project\test source" stores my Benchmarks' class files.

3. Project type: Application. The main application class is TestOBF
4. Obfuscation options:

    "Remove All" and "Rename Classes_Profix"

This option set is the highest security level in all the option sets that DashO-Pro v2.0 provides

5. Optimization: "None"

Output:

```
TestOBF.class(873 bytes), b.class (238 bytes)
```

Function Test:

Type:

prompt >java TestOBF

Get result:

```
HELLO
GOOD
number=1
```

Therefore the obfuscated class files run the same result as their unobfuscated ones do.

Obfuscated Benchmark1 bytecodes:

Run javap to get obfuscated bytecode from TestOBF.class:

prompt>javap -c -p -l TestOBF

See the bytecode listing in the Program 6_1 below:

Program 6_1: Obfuscated Bytecode_TestOBF of *DashO-Pro v2.0*

```
Compiled from [DashoPro-V2-050200]
public class TestOBF extends java.lang.Object {
    public b a;
    public java.lang.String b;
    public int c[];
    public TestOBF();
    private int b();
    public java.lang.String a();
    public static void main(java.lang.String[]);
}

Method TestOBF()
   0 aload_0
   1 invokespecial #55 <Method java.lang.Object()>
   4 aload_0
   5 new #38 <Class b>
   8 dup
   9 invokespecial #56 <Method b()>
  12 putfield #64 <Field b a>
  15 aload_0
  16 ldc #1 <String "">
  18 putfield #65 <Field java.lang.String b>
  21 aload_0
  22 bipush 10
  24 newarray int
  26 putfield #66 <Field int c[]>
  29 getstatic #67 <Field java.io.PrintStream out>
  32 aload_0
  33 getfield #64 <Field b a>
  36 ldc #2 <String "HELLO">
  38 invokevirtual #57 <Method java.lang.String
a(java.lang.String)>
  41 invokevirtual #58 <Method void println(java.lang.String)>
  44 getstatic #67 <Field java.io.PrintStream out>
  47 aload_0
  48 invokevirtual #59 <Method java.lang.String a()>
  51 invokevirtual #58 <Method void println(java.lang.String)>
  54 getstatic #67 <Field java.io.PrintStream out>
  57 new #41 <Class java.lang.StringBuffer>
```

```
  60 dup
  61 ldc #3 <String "number=">
  63 invokespecial #60 <Method
java.lang.StringBuffer(java.lang.String)>
  66 aload_0
  67 invokespecial #61 <Method int b()>
  70 invokevirtual #62 <Method java.lang.StringBuffer
append(int)>
  73 invokevirtual #63 <Method java.lang.String toString()>
  76 invokevirtual #58 <Method void println(java.lang.String)>
  79 return

Method int b()
   0 aload_0
   1 getfield #66 <Field int c[]>
   4 iconst_0
   5 iconst_1
   6 iastore
   7 aload_0
   8 getfield #66 <Field int c[]>
  11 iconst_0
  12 iaload
  13 ireturn

Method java.lang.String a()
   0 aload_0
   1 ldc #4 <String "GOOD">
   3 putfield #65 <Field java.lang.String b>
   6 aload_0
   7 getfield #65 <Field java.lang.String b>
  10 areturn

Method void main(java.lang.String[])
   0 new #40 <Class TestOBF>
   3 invokespecial #68 <Method TestOBF()>
   6 return
```

Run javap to get obfuscated bytecode from b.class:

prompt>javap -c -p -l b

See the bytecode listing in the Program 6_2 below:

Program 6_2: Obfuscated Bytecode_Hello of *DashO-Pro v2.0*

```
Compiled from [DashoPro-V2-050200]
public class b extends java.lang.Object {
    public b();
    public java.lang.String a(java.lang.String);
}

Method b()
   0 aload_0
   1 invokespecial #14 <Method java.lang.Object()>
   4 iconst_1
   5 istore_1
   6 return

Method java.lang.String a(java.lang.String)
```

```
0 aload_1
1 areturn
```

Result report: Success.

Comparing above obfuscated bytecodes (Progeam 6_1 and Program 6_2) with their unobfuscated bytecodes (Program 5_2 and program 5_3), I scored the following obfuscation objective metrics. See Table6_4.

Table 6_4: The result of obfuscating Benchmark1 by *DashO-Pro v2.0*

Metrics		**Original Names**	**Obfuscation Names**	**Scores**
Public	Class name	Hello	b	1
	Method name	getHello	a	
	Field name	hello	a	
Protected	Method name	getGood	a	1
	Field name	good	b	
Private	Method name	getArray	b	1
	field name	number	c	
Remove Line numbers		Line numbers	Blanked out	1
Encrypt String		GOOD	GOOD	0
Unstructured data: One dimension array		number[]	c[]	0
Score for Layout and Data Obfuscations				4

Note: Case-sensitive

From Table 6_4, we can see that *DashO-Pro v2.0* gets full mark, 4 scores, on the Layout obfuscation metrics. However, it got zero mark on all Data Obfuscation metrics. Therefore *DashO-Pro v2.0* gets total 4 scores for obfuscating my Benchmark1.

Obfuscating Benchmark2:

*DashO-Pro v2.*0 does not advertise that it can obfuscate programs with control flow (see Table 6_1). So I did not test it and scored “0” for the metric, “Control Obfuscation”.

Result report: N/A.

Summary:

- *DashO-Pro v2.0* can lexically obfuscate my Benchmark1. Like its advertisement (see table 6_1), *DashO-Pro v2.0* can change public names, protected names and private names. It also can remove debugging information such as Line numbers.
- My Benchmark1, after obfuscated by *DashO-Pro v2.0*, still function properly.
- I reported "N/A" for Obfuscating Benchmark2.
- Combining the results from Benchmark1 and Benchmark2, I could score most obfuscation individual Metrics for *DashO-Pro v2.0*. See Table 6_5.

Table 6_5: The result of obfuscating Benchmarks by *DashO-Pro v2.0*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	1
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		0
	Folding one dimension array		0
Control Obfuscation	Change "if…else", "for" or "while" loop		0
Score for Layout, Data and Control Obfuscations			4

From this table, we can see that *DashO-Pro v2.0* gets total 4 marks for obfuscating my Benchmarks. All this 4 marks are from Layout Obfuscation Metrics. It gets zero mark for other metrics, Data Obfuscation and Control Obfuscation.

6.2.3 JCloak v3.5.3

Test Time: 15/12/2000

Platform: Window 98

Obfuscating Benchmark1:

Main steps:

1. Double click "JCloakGUI.exe" to start JCloak.
2. Input a single classpath entry for program classes:
   D:\lhy\project\test source
3. Input classpath:

   C:\Program Files\JavaSoft\JRE\1.2\lib\i18n.jar

C:\Program Files\JavaSoft\JRE\1.2\lib\plugprov.jar

C:\Program Files\JavaSoft\JRE\1.2\lib\rt.jar

D:\lhy\project\test source

main() class:'TestOBF'

4. Choose Obfuscation Options:

obfuscate symbols

obfuscate classnames

strip linenumbers

strip local symbols

Output:

TestOBF.class (946 bytes), C0.class(213 bytes)

Function Test:

Type:

prompt >java TestOBF

Get result:

```
HELLO
GOOD
number=1
```

Therefore the obfuscated class files run the same result as their unobfuscated ones do.

Obfuscated Benchmark1 bytecodes:

Run javap to get obfuscated bytecode from TestOBF.class:

prompt>javap -c -p -l TestOBF

See the bytecode listing in the Program 6_3 below:

Program 6_3 Obfuscated Bytecode_TestOBF of *JCloak*

```
Compiled from TestOBF
public class TestOBF extends java.lang.Object {
    public C0 c;
    protected java.lang.String d;
    private int e[];
    public TestOBF();
    private int a();
    protected java.lang.String b();
    public static void main(java.lang.String[]);
}

Method TestOBF()
   0 aload_0
   1 invokespecial #38 <Method java.lang.Object()>
```

```
   4 aload_0
   5 new #39 <Class C0>
   8 dup
   9 invokespecial #25 <Method C0()>
  12 putfield #67 <Field C0 c>
  15 aload_0
  16 ldc #4 <String "">
  18 putfield #21 <Field java.lang.String d>
  21 aload_0
  22 bipush 10
  24 newarray int
  26 putfield #56 <Field int e[]>
  29 getstatic #65 <Field java.io.PrintStream out>
  32 aload_0
  33 getfield #67 <Field C0 c>
  36 ldc #2 <String "HELLO">
  38 invokevirtual #49 <Method java.lang.String
a(java.lang.String)>
  41 invokevirtual #9 <Method void println(java.lang.String)>
  44 getstatic #65 <Field java.io.PrintStream out>
  47 aload_0
  48 invokevirtual #50 <Method java.lang.String b()>
  51 invokevirtual #9 <Method void println(java.lang.String)>
  54 getstatic #65 <Field java.io.PrintStream out>
  57 new #26 <Class java.lang.StringBuffer>
  60 dup
  61 ldc #1 <String "number=">
  63 invokespecial #7 <Method
java.lang.StringBuffer(java.lang.String)>
  66 aload_0
  67 invokespecial #24 <Method int a()>
  70 invokevirtual #44 <Method java.lang.StringBuffer
append(int)>
  73 invokevirtual #42 <Method java.lang.String toString()>
  76 invokevirtual #9 <Method void println(java.lang.String)>
  79 return
Method int a()
   0 aload_0
   1 getfield #56 <Field int e[]>
   4 iconst_0
   5 iconst_1
   6 iastore
   7 aload_0
   8 getfield #56 <Field int e[]>
  11 iconst_0
  12 iaload
  13 ireturn
Method java.lang.String b()
   0 aload_0
   1 ldc #3 <String "GOOD">
   3 putfield #21 <Field java.lang.String d>
   6 aload_0
   7 getfield #21 <Field java.lang.String d>
  10 areturn

Method void main(java.lang.String[])
   0 new #59 <Class TestOBF>
   3 invokespecial #52 <Method TestOBF()>
   6 return
```

Run javap to get obfuscated bytecode from b.class:

prompt>javap -c -p -l b

See the bytecode listing in the Program 6_4 below:

Program 6_4: Obfuscated Bytecode_Hello of *JCloak*

```
Compiled from C0
class C0 extends java.lang.Object {
    public C0();
    public java.lang.String a(java.lang.String);
}
Method C0()
   0 aload_0
   1 invokespecial #7 <Method java.lang.Object()>
   4 iconst_1
   5 istore_1
   6 return
Method java.lang.String a(java.lang.String)
   0 aload_1
   1 areturn
```

Result report: Success.

Comparing above obfuscated bytecodes (Progeam 6_3, Program 6_4) with their unobfuscated bytecodes (Program 5_2, Program 5_3), I scored the following obfuscation objective metrics. See Table6_6.

Table 6_6 The result of obfuscating Benchmark1 by *JCloak*

Metrics		**Original Names**	**Obfuscation Names**	**Score**
Public	Class name	Hello	C0	1
	Method name	getHello	a	
	Field name	hello	c	
Protected	Method name	getGood	b	1
	Field name	good	d	
Private	Method name	getArray	a	1
	field name	number	e	
Remove Line numbers		Line numbers	Blanked out	1
Encrypt String		GOOD	GOOD	0
Unstructured data: One dimension array		number[]	e[]	0
Score for Layout and Data Obfuscations				4

Note: Case-sensitive

From Table 6_6, we can see that *JCloak v3.5.3*gets full mark, 4 scores, on the Layout obfuscation metrics. However, it got zero mark on all Data Obfuscation metrics. Therefore *JCloak v3.5.3* gets total 4 points for obfuscating my Benchmark1.

Obfuscating Benchmark2:

JCloak v3.5.3 does not advertise that it can obfuscate programs with control flow (see Table 6_1). So I did not test it and scored "0" for the metric, "Control Obfuscation".

Result report: N/A.

Summary:

- *JCloak v3.5.3* can lexically obfuscate my Benchmark1. Like its advertisement (see table 6_1), *JCloak v3.5.3* can change public names, protected names and private names. It also can remove debugging information such as Line numbers.
- My Benchmark1, after obfuscated by *JCloak v3.5.3*, still function properly.
- I reported "N/A" for Obfuscating Benchmark2.
- Combining the results from Benchmark1 and Benchmark2, I could score most obfuscation individual Metrics for *JCloak v3.5.3*. See Table 6_7.

Table 6_7: The result of obfuscating Benchmarks by *JCloak v3.5.3*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	1
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		0
	Folding one dimension array		0
Control Obfuscation	Change "if…else", "for" or "while" loop		0
Score for Layout, Data and Control Obfuscations			4

From this table, we can see that *JCloak v3.5.3* gets total 4 marks for obfuscating my Benchmarks. All these 4 marks are from Layout Obfuscation Metrics. It got zero mark for other metrics, Data Obfuscation and Control Obfuscation.

6.2.4 Jproof 1stBarrier MINI v1.1

Test Time: 15/12/2000

Platform: Window 98

Obfuscating Benchmark1:

Jproof 1stBarrierMINI v1.1 supports only .JAR file. So before obfuscating, I archived Benchmark1 class files, TestOBF.class and Hello.class, into an archive, testclass.jar:

prompt >jar cvf testclasses.jar TestOBF.class Hello.class

Main steps:

1. Double click "1stBarrierMINI.jar" to start *Jproof 1stBarrierMINI v1.1*.
2. Input source name: testclass.jar.
3. Enter "main" class: TestOBF.class.
4. Choose Obfuscation Options: Default, it is the highest security level

Output: testclass.jar.

Type:

prompt >jar xvf testclasses.jar TestOBF.class Hello.class

Get two classes: TestOBF.class and dl.class

Function Test:

Type:

prompt >java TestOBF

Get result:

```
HELLO
GOOD
number=1
```

Therefore the obfuscated class files run the same result as their unobfuscated ones do.

Obfuscated Benchmark1 bytecodes:

Run javap to get obfuscated bytecode from TestOBF.class:

prompt>javap -c -p -l TestOBF

See the bytecode listing in the Program 6_5 below:

Program 6_5 Obfuscated Bytecode_TestOBF of *Jproof 1stBarrierMINI v1.1*

```
No sourcepublic class TestOBF extends java.lang.Object {
   public dl hello;
   protected java.lang.String good;
    private int Lt[];
    public TestOBF();
    private int lt();
    protected java.lang.String getGood();
    public static void main(java.lang.String[]);
}

Method TestOBF()
   0 aload_0
   1 invokespecial #13 <Method java.lang.Object()>
   4 aload_0
   5 new #5 <Class dl>
   8 dup
   9 invokespecial #11 <Method dl()>
  12 putfield #20 <Field dl hello>
  15 aload_0
  16 ldc #1 <String "">
  18 putfield #19 <Field java.lang.String good>
  21 aload_0
  22 bipush 10
  24 newarray int
  26 putfield #21 <Field int Lt[]>
  29 getstatic #22 <Field java.io.PrintStream out>
  32 aload_0
  33 getfield #20 <Field dl hello>
  36 ldc #3 <String "HELLO">
  38 invokevirtual #18 <Method java.lang.String
getHello(java.lang.String)>
  41 invokevirtual #23 <Method void println(java.lang.String)>
  44 getstatic #22 <Field java.io.PrintStream out>
  47 aload_0
  48 invokevirtual #17 <Method java.lang.String getGood()>
  51 invokevirtual #23 <Method void println(java.lang.String)>
  54 getstatic #22 <Field java.io.PrintStream out>
  57 new #9 <Class java.lang.StringBuffer>
  60 dup
  61 ldc #4 <String "number=">
  63 invokespecial #14 <Method
java.lang.StringBuffer(java.lang.String)>
  66 aload_0
  67 invokespecial #16 <Method int lt()>
  70 invokevirtual #15 <Method java.lang.StringBuffer
append(int)>
  73 invokevirtual #24 <Method java.lang.String toString()>
  76 invokevirtual #23 <Method void println(java.lang.String)>
  79 return
Method int lt()
   0 aload_0
   1 getfield #21 <Field int Lt[]>
   4 iconst_0
   5 iconst_1
   6 iastore
   7 aload_0
```

```
   8 getfield #21 <Field int Lt[]>
  11 iconst_0
  12 iaload
  13 ireturn
Method java.lang.String getGood()
   0 aload_0
   1 ldc #2 <String "GOOD">
   3 putfield #19 <Field java.lang.String good>
   6 aload_0
   7 getfield #19 <Field java.lang.String good>
  10 areturn

Method void main(java.lang.String[])
   0 new #6 <Class TestOBF>
   3 invokespecial #12 <Method TestOBF()>
   6 return
```

Run javap to get obfuscated bytecode from b.class:

prompt>javap -c -p -l b

See the bytecode listing in the Program 6_6 below:

Program 6_6: Obfuscated Bytecode_Hello of *Jproof 1stBarrierMINI v1.1*

```
No sourceclass dl extends java.lang.Object {
   public dl();
   public java.lang.String getHello(java.lang.String);
}
Method dl()
   0 aload_0
   1 invokespecial #4 <Method java.lang.Object()>
   4 iconst_1
   5 istore_1
   6 return
Method java.lang.String getHello(java.lang.String)
   0 aload_1
   1 areturn
```

Result report: Success.

Comparing above obfuscated bytecodes (Progeam 6_5, Program 6_6) with their unobfuscated bytecodes (Program 5_2, Program 5_3), I scored the following obfuscation objective metrics. See Table6_8.

From Table 6_8, we can see that *Jproof 1stBarrierMINI v1.1* gets 3 scores. They were from obfuscating public class names, private names and removing Line numbers. *Jproof 1stBarrierMINI v1.1* got zero mark on all Data Obfuscation metrics and renaming protected metric. Though *Jproof 1stBarrierMINI v1.1* did not obfuscate public method names and public field names but public class names, I had to score 1 mark on the “Public” names metric by my marking standard rule.

Table 6_8: The result of obfuscating Benchmark1 by *Jproof 1stBarrierMINI v1.1*

Metrics		**Original Names**	**Obfuscation Names**	**Score**
Public	Class name	Hello	dl	1
	Method name	getHello	getHello	
	Field name	hello	hello	
Protected	Method name	getGood	getGood	0
	Field name	good	good	
Private	Method name	getArray	lt	1
	field name	number	Lt	
Remove Line numbers		Line numbers	Blanked out	1
Encrypt String		GOOD	GOOD	0
Unstructured data: One dimension array		number[]	Lt[]	0
Score for Layout and Data Obfuscations				3

Note: Case-sensitive

Obfuscating Benchmark2:

Jproof 1stBarrierMINI v1.1 did not advertise that it could obfuscate programs with control flow (see Table 6_1). So I did not test it and scored "0" for the metric, "Control Obfuscation".

Result report: N/A.

Summary:

- *Jproof 1stBarrierMINI v1.1 could* obfuscatedmy Benchmark1. Like its advertisement (see table 6_1), *Jproof 1stBarrierMINI v1.1* changed public class names, private method names and private methods. It also can removed debugging information such as Line numbers.
- My Benchmark1, after being obfuscated by *Jproof 1stBarrierMINI v1.1*, still function properly.
- I reported "N/A" for Obfuscating Benchmark2.
- Combining the results from Benchmark1 and Benchmark2, I could score most obfuscation individual Metrics for *Jproof 1stBarrierMINI v1.1*. See Table 6_9.

Table 6_9: The result of obfuscating Benchmarks by *Jproof 1stBarrierMINI v1.1*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	0
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		0
	Folding one dimension array		0
Control Obfuscation	Change "if…else", "for" or "while" loop		0
Score for Layout, Data and Control Obfuscations			3

From this table, we can see that: *Jproof 1stBarrierMINI v1.1*gets total 3 marks for obfuscating my Benchmarks. All this 3 marks are from Layout Obfuscation Metrics but it lost 1 mark here. It gets zero mark for Data obfuscation and Control Obfuscation.

6.2.5 Jshrink v1.18

Test Time: 15/12/2000

Platform: Window 98 command line (MS-DOS window)

Obfuscating Benchmark1:

Main steps:

1. Put the tested files, TestOBF.class and Hello.class into a folder
2. On the command line, type:

   prompt >jshrink *.class

Output:

TestOBF.class (890 bytes) and Hello.class (223 bytes)

Function Test:

Type:

prompt >java TestOBF

Get result:

```
HELLO
GOOD
number=1
```

Therefore the obfuscated class files run the same result as their unobfuscated ones do.

Obfuscated Benchmark1 bytecodes:

Run javap to get obfuscated bytecode from TestOBF.class:

prompt>javap -c -p -l TestOBF

See the bytecode listing in the Program 6_7 below:

Program 6_7: Obfuscated Bytecode_TestOBF of *Jshrink v1.18*

```
Compiled from TestOBF
public class TestOBF extends java.lang.Object {
    public Hello hello;
    protected java.lang.String good;
    private int 0[];
    public TestOBF();
    private int 0();
    protected java.lang.String getGood();
    public static void main(java.lang.String[]);
}
Method TestOBF()
   0 aload_0
   1 invokespecial #13 <Method java.lang.Object()>
   4 aload_0
   5 new #5 <Class Hello>
   8 dup
   9 invokespecial #11 <Method Hello()>
  12 putfield #20 <Field Hello hello>
  15 aload_0
  16 ldc #1 <String "">
  18 putfield #19 <Field java.lang.String good>
  21 aload_0
  22 bipush 10
  24 newarray int
  26 putfield #21 <Field int 0[]>
  29 getstatic #22 <Field java.io.PrintStream out>
  32 aload_0
  33 getfield #20 <Field Hello hello>
  36 ldc #3 <String "HELLO">
  38 invokevirtual #18 <Method java.lang.String
getHello(java.lang.String)>
  41 invokevirtual #23 <Method void println(java.lang.String)>
  44 getstatic #22 <Field java.io.PrintStream out>
  47 aload_0
  48 invokevirtual #17 <Method java.lang.String getGood()>
  51 invokevirtual #23 <Method void println(java.lang.String)>
  54 getstatic #22 <Field java.io.PrintStream out>
  57 new #9 <Class java.lang.StringBuffer>
  60 dup
  61 ldc #4 <String "number=">
```

```
  63 invokespecial #14 <Method
java.lang.StringBuffer(java.lang.String)>
  66 aload_0
  67 invokespecial #16 <Method int 0()>
  70 invokevirtual #15 <Method java.lang.StringBuffer
append(int)>
  73 invokevirtual #24 <Method java.lang.String toString()>
  76 invokevirtual #23 <Method void println(java.lang.String)>
  79 return
Method int 0()
   0 aload_0
   1 getfield #21 <Field int 0[]>
   4 iconst_0
   5 iconst_1
   6 iastore
   7 aload_0
   8 getfield #21 <Field int 0[]>
  11 iconst_0
  12 iaload
  13 ireturn
Method java.lang.String getGood()
   0 aload_0
   1 ldc #2 <String "GOOD">
   3 putfield #19 <Field java.lang.String good>
   6 aload_0
   7 getfield #19 <Field java.lang.String good>
  10 areturn
Method void main(java.lang.String[])
   0 new #6 <Class TestOBF>
   3 invokespecial #12 <Method TestOBF()>
   6 return
```

Run javap to get obfuscated bytecode from Hello.class:

prompt>javap -c -p -l Hello

See the bytecode listing in the Program 6_8 below:

Program 6_8 Obfuscated Bytecode_Hello of *Jshrink v1.18*

```
Compiled from Hello
class Hello extends java.lang.Object {
    public Hello();
    public java.lang.String getHello(java.lang.String);
}
Method Hello()
   0 aload_0
   1 invokespecial #3 <Method java.lang.Object()>
   4 iconst_1
   5 istore_1
   6 return
Method java.lang.String getHello(java.lang.String)
   0 aload_1
   1 areturn
```

Result report: Success.

Comparing above obfuscated bytecodes (Progeam 6_7, Program 6_8) with their unobfuscated bytecodes (Program 5_2, Program 5_3), I scored the following obfuscation objective metrics. See Table 6_10.

From Table 6_10, we can see that *JShrink v1.18* gets 2 scores. They are from obfuscating private names and removing Line numbers. On other metrics, *JShrink v1.18* gets zero mark.

Table 6_10 The result of obfuscating Benchmark1 by *JShrink v1.18*

Metrics		**Original Names**	**Obfuscation Names**	**Score**
Public	Class name	Hello	Hello	0
	Method name	getHello	getHello	
	Field name	hello	hello	
Protected	Method name	getGood	getGood	0
	Field name	good	good	
Private	Method name	getArray	0	1
	field name	number	0	
Remove Line numbers		Line numbers	Blanked out	1
Encrypt String		GOOD	GOOD	0
Unstructured data: One dimension array		number[]	0[]	0
Score for Layout and Data Obfuscations				2

Note: Case-sensitive

Obfuscating Benchmark2:

JShrink v1.18 did not advertise that it could obfuscate programs with control flow (see Table 6_1). So I did not test it and scored "0" for the metric, "Control Obfuscation".

Result report: N/A

Summary:

- *JShrink v1.18 can* obfuscate my Benchmark1. Like its advertisement (see table 6_1), *JShrink v1.18*change private method names and private methods. It also can remove debugging information such as Line numbers.
- My Benchmark1, after obfuscated by *JShrink v1.18*, still function properly.

- I reported “N/A” for Obfuscating Benchmark2.
- Combining the results from Benchmark1 and Benchmark2, I could score most obfuscation individual Metrics for *JShrink v1.18*. See Table 6_11.

Table 6_11: The result of obfuscating Benchmarks by *JShrink v1.18*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	0
		Protected	0
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		0
	Folding one dimension array		0
Control Obfuscation	Change “if…else”, “for” or “while” loop		0
Score for Layout, Data and Control Obfuscations			2

From this table, we can see that *JShrink v1.18* gets total 2 marks for obfuscating my Benchmarks. These 2 marks are from obfuscating private names and removing Line numbers.

6.2.6 Jzipper v1.08

Test Time: 16/12/2000

Platform: Window 98

Obfuscating Benchmarks:

When I double clicked “Jzipper.bat” to start *Jzipper v1.08*.

It showed the following message:

```
JZipper software has expired, please return to its website to
retrieve the latest version.
```

I had downloaded it again and tried running it for three times. However, I still got the same result. I tried using it obfuscate my codes. It could obfuscate my Benchmark1. But when I run the obfuscated code, it turned out:

```
Exception in thread "main" java.lang.VerifyError: (class: JZ7,
 method: <init> signature: ()V) Bad type in putfield/putstatic.
```

Result report: Failure

Summary:

- *Jzipper v1.08* could obfuscate my Benchmark1 (I did not obfuscate Benchmark2).
- Its obfuscated codes could not function improperly.
- I reported it failure and gave zero mark for its Layout, Data and Control Obfuscation metrics.

6.2.7 RetroGuard_v1.1

Test Time: 16/12/2000

Platform: Window 98 command line

Obfuscating Benchmarks:

RetroGuard_v1.1.zip supports only .JAR file. So before obfuscating, I archived Benchmark1 class files, TestOBF.class and Hello.class, into an archive, test.jar:

prompt >java cvf test.jar TestOBF.class Hello.class

Main steps:

1. Type **prompt** >java RetroGuard Test.jar to get output out.jar
2. De-archive out.jar to get obfuscated files: a.class and b.class

Function Test:

Running obfuscated classes, I got a message:

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

Result report: Failure

Summary:

- *RetroGuard_v1.1.zip* supports only .JAR file.
- It runs on command line with no options.
- It could obfuscate my Benchmark1 (I did not obfuscate Benchmark2).
- Its obfuscated codes could not function improperly.
- I reported it failure and gave zero mark for its Layout, Data and Control Obfuscation metrics.

.

6.2.8 SourceGuard v4.0

Test Time: 17/12/2000

Platform: Window 98

Obfuscating Benchmark1:

Main steps:

Choose Obfuscation Options:

Protect all

Modify Bytecode Range

Hide loops

Remove SourceFile Attribute

Remove Line number Table Attribute

Remove Local variable Table Attribute

Remove Innerclass Attribute

Remove underlined Attribute

Remove Constant Pool Constant matching class names

Output:

TestOBF.class (829 bytes), a.class(187 bytes)

Function Test:

Type:

prompt >java TestOBF

Get result:

```
HELLO
GOOD
number=1
```

Therefore the obfuscated class files run the same result as their unobfuscated ones do.

Obfuscated Benchmark1 bytecodes:

Run javap to get obfuscated bytecode from TestOBF.class:

prompt>javap -c -p -l TestOBF

See the bytecode listing in the Program 6_9 below:

Program 6_9 Obfuscated Bytecode_TestOBF of *SourceGuard v4.0*

```
No sourcepublic class TestOBF extends java.lang.Object {
    private int a[];
    protected java.lang.String c;
    public a b;
    public static void main(java.lang.String[]);
    protected java.lang.String b();
    private int a();
    public TestOBF();
}

Method void main(java.lang.String[])
   0 new #10 <Class TestOBF>
   3 invokespecial #18 <Method TestOBF()>
   6 return

Method java.lang.String b()
   0 aload_0
   1 ldc #1 <String "GOOD">
   3 putfield #26 <Field java.lang.String c>
   6 aload_0
   7 getfield #26 <Field java.lang.String c>
  10 areturn

Method int a()
   0 aload_0
   1 getfield #13 <Field int a[]>
   4 iconst_0
   5 iconst_1
   6 iastore
   7 aload_0
   8 getfield #13 <Field int a[]>
  11 iconst_0
  12 iaload
  13 ireturn

Method TestOBF()
   0 aload_0
   1 invokespecial #24 <Method java.lang.Object()>
   4 aload_0
   5 new #8 <Class a>
   8 dup
   9 invokespecial #20 <Method a()>
  12 putfield #15 <Field a b>
  15 aload_0
  16 ldc #3 <String "">
  18 putfield #26 <Field java.lang.String c>
  21 aload_0
  22 bipush 10
  24 newarray int
  26 putfield #13 <Field int a[]>
  29 getstatic #14 <Field java.io.PrintStream out>
  32 aload_0
  33 getfield #15 <Field a b>
  36 ldc #4 <String "HELLO">
  38 invokevirtual #35 <Method java.lang.String
a(java.lang.String)>
  41 invokevirtual #25 <Method void println(java.lang.String)>
  44 getstatic #14 <Field java.io.PrintStream out>
```

```
  47 aload_0
  48 invokevirtual #30 <Method java.lang.String b()>
  51 invokevirtual #25 <Method void println(java.lang.String)>
  54 getstatic #14 <Field java.io.PrintStream out>
  57 new #5 <Class java.lang.StringBuffer>
  60 dup
  61 ldc #2 <String "number=">
  63 invokespecial #33 <Method
java.lang.StringBuffer(java.lang.String)>
  66 aload_0
  67 invokespecial #29 <Method int a()>
  70 invokevirtual #27 <Method java.lang.StringBuffer
append(int)>
  73 invokevirtual #34 <Method java.lang.String toString()>
  76 invokevirtual #25 <Method void println(java.lang.String)>
  79 return
```

Run javap to get obfuscated bytecode from a.class:

prompt>javap -c -p -l a

See the bytecode listing in the Program 6_10 below:

Program 6_10 Obfuscated Bytecode_Hello of of *SourceGuard v4.0*

```
No sourceclass a extends java.lang.Object {
    public java.lang.String a(java.lang.String);
    public a();
}

Method java.lang.String a(java.lang.String)
   0 aload_1
   1 areturn

Method a()
   0 aload_0
   1 invokespecial #4 <Method java.lang.Object()>
   4 iconst_1
   5 istore_1
   6 return
```

Result report: Success.

Comparing above obfuscated bytecodes (Progeam 6_9, Program 6_10) with their unobfuscated bytecodes (Program 5_2, Program 5_3), I scored the following obfuscation objective metrics. See Table6_12.

From Table 6_12, we can see that *SourceGuard v4.0* gets full mark, 4 scores, on the Layout obfuscation metrics. However, it got zero mark on all Data Obfuscation metrics. Therefore *SourceGuard v4.0* gets total 4 points for obfuscating my Benchmark1.

Table 6_12 The result of obfuscating Benchmark1 by *SourceGuard v4.0*

Metrics		Original Names	Obfuscation Names	Score
Public	Class name	Hello	a	1
	Method name	getHello	a	
	Field name	hello	b	
Protected	Method name	getGood	b	1
	Field name	good	c	
Private	Method name	getArray	a	1
	field name	number	a	
Remove Line numbers		Line numbers	Blanked out	1
Encrypt String		GOOD	GOOD	0
Unstructured data: One dimension array		number[]	a[]	0
Score for Layout and Data Obfuscations				4

Note: Case-sensitive

Obfuscating Benchmark2:

SourceGuard v4.0 does not advertise that it can obfuscate programs with control flow (see Table 6_1). So I did not test it and scored "0" for the metric, "Control Obfuscation".

Result report: N/A.

Summary:

- *SourceGuard v4.0* could lexically obfuscate my Benchmark1. Like its advertisement (see table 6_1), *SourceGuard v4.0 was able to* change public names, protected names and private names. It also could remove debugging information such as Line numbers.
- My Benchmark1, after being obfuscated by *SourceGuard v4.0*, still functioned properly.
- I reported "N/A" for Obfuscating Benchmark2.
- Summing the results from Benchmark1 and Benchmark2, I could score most obfuscation individual Metrics for *SourceGuard v4.0*. See Table 6_13.

Table 6_13: The result of obfuscating Benchmarks by *SourceGuard v4.0*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	1
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		0
	Folding one dimension array		0
Control Obfuscation	Change "if…else", "for" or "while" loop		0
Score for Layout, Data and Control Obfuscations			4

From this table, we can see that *SourceGuard v4.0* got total 4 marks for obfuscating my Benchmarks. All these 4 marks are from Layout Obfuscation Metrics. It got zero mark for other metrics, Data Obfuscation and Control Obfuscation.

6.2.9 KlassMaster v2.3

Test Time: 17/12/2000

Platform: Window 98

Obfuscating Benchmark1:

Main steps:

1. Double click "ZKM.jar" to start ZKM
2. Choose Obfuscation Options:
   - rename and remove all
   - obfuscate Control Flow: aggressive
   - Encrypt String Literal: flow obfuscate.

Output:

b.class (1,170 bytes), a.class(256 bytes)

Function Test:

Type:

prompt >java b

Get result:

```
HELLO
GOOD
number=1
```

Therefore the obfuscated class files run the same result as their unobfuscated ones do.

Obfuscated Benchmark1 bytecodes:

Run javap to get obfuscated bytecode from b.class:

prompt>javap -c -p -l TestOBF

See the bytecode listing in the Program 6_11 below:

Program 6_11: Obfuscated Bytecode_TestOBF of *KlassMaster **v2.3***

```
Compiled from b.java
public class b extends java.lang.Object {
    public a a;
    protected java.lang.String b;
    private int c[];
    static int d;
    public b();
    private int a();
    protected java.lang.String b();
    public static void main(java.lang.String[]);
    private static java.lang.String a(java.lang.String);
}

Method b()
   0 getstatic #61 <Field int d>
   3 istore_1
   4 aload_0
   5 invokespecial #13 <Method java.lang.Object()>
   8 aload_0
   9 new #5 <Class a>
  12 dup
  13 invokespecial #11 <Method a()>
  16 putfield #20 <Field a a>
  19 aload_0
  20 ldc #1 <String "">
  22 putfield #19 <Field java.lang.String b>
  25 aload_0
  26 bipush 10
  28 newarray int
  30 putfield #21 <Field int c[]>
  33 getstatic #22 <Field java.io.PrintStream out>
  36 aload_0
  37 getfield #20 <Field a a>
  40 ldc #3 <String "]  ~'X">
  42 invokestatic #92 <Method java.lang.String
a(java.lang.String)>
  45 invokevirtual #18 <Method java.lang.String
a(java.lang.String)>
  48 invokevirtual #23 <Method void println(java.lang.String)>
  51 getstatic #22 <Field java.io.PrintStream out>
  54 aload_0
  55 invokevirtual #17 <Method java.lang.String b()>
```

```
  58 invokevirtual #23 <Method void println(java.lang.String)>
  61 getstatic #22 <Field java.io.PrintStream out>
  64 new #9 <Class java.lang.StringBuffer>
  67 dup
  68 ldc #4 <String "{9"(rgq">
  70 invokestatic #92 <Method java.lang.String
a(java.lang.String)>
  73 invokespecial #14 <Method
java.lang.StringBuffer(java.lang.String)>
  76 aload_0
  77 invokespecial #16 <Method int a()>
  80 invokevirtual #15 <Method java.lang.StringBuffer
append(int)>
  83 invokevirtual #24 <Method java.lang.String toString()>
  86 invokevirtual #23 <Method void println(java.lang.String)>
  89 iload_1
  90 ifeq 107
  93 getstatic #74 <Field boolean a>
  96 ifeq 103
  99 iconst_0
 100 goto 104
 103 iconst_1
 104 putstatic #74 <Field boolean a>
 107 return

Method int a()
   0 aload_0
   1 getfield #21 <Field int c[]>
   4 iconst_0
   5 iconst_1
   6 iastore
   7 aload_0
   8 getfield #21 <Field int c[]>
  11 iconst_0
  12 iaload
  13 ireturn

Method java.lang.String b()
   0 aload_0
   1 ldc #2 <String "R~`">
   3 invokestatic #92 <Method java.lang.String
a(java.lang.String)>
   6 putfield #19 <Field java.lang.String b>
   9 aload_0
  10 getfield #19 <Field java.lang.String b>
  13 areturn

Method void main(java.lang.String[])
   0 new #6 <Class b>
   3 invokespecial #12 <Method b()>
   6 return

Method java.lang.String a(java.lang.String)
   0 aload_0
   1 invokevirtual #87 <Method char toCharArray()[]>
   4 astore_1
   5 aload_1
   6 arraylength
   7 istore_2
   8 iconst_0
   9 istore_3
```

```
  10 iload_2
  11 iconst_1
  12 if_icmpgt 89
  15 aload_1
  16 iload_3
  17 dup2
  18 caload
  19 iload_3
  20 iconst_5
  21 irem
  22 tableswitch 0 to 3: default=72
         0: 52
         1: 57
         2: 62
         3: 67
  52 ldc #76 <Integer 21>
  54 goto 74
  57 ldc #77 <Integer 76>
  59 goto 74
  62 ldc #78 <Integer 79>
  64 goto 74
  67 ldc #79 <Integer 74>
  69 goto 74
  72 ldc #80 <Integer 23>
  74 ixor
  75 i2c
  76 castore
  77 iinc 3 1
  80 iload_2
  81 ifne 89
  84 aload_1
  85 iload_2
  86 goto 17
  89 iload_3
  90 iload_2
  91 if_icmplt 15
  94 new #81 <Class java.lang.String>
  97 dup
  98 aload_1
  99 invokespecial #91 <Method java.lang.String(char[])>
 102 areturn
```

Run javap to get obfuscated bytecode from a.class:

prompt>javap -c -p -l a

See the bytecode listing in the Program 6_12 below:

Program 6_12 Obfuscated Bytecode_Hello of *KlassMaster v2.3*

```
Compiled from a.java
class a extends java.lang.Object {
    public static boolean a;
    public a();
    public java.lang.String a(java.lang.String);
}
Method a()
   0 aload_0
   1 invokespecial #4 <Method java.lang.Object()>
```

```
   4 iconst_1
   5 istore_1
   6 return
Method java.lang.String a(java.lang.String)
   0 aload_1
   1 areturn
```

Result report: Success.

Comparing the obfuscated bytecodes (Progeam 6_11, Program 6_12) with their unobfuscated bytecodes (Program 5_2, Program 5_3), I scored the following obfuscation objective metrics. See Table6_14.

Table 6_14: The result of obfuscating Benchmark1 by *KlassMaster v2.3*

Metrics		**Original Names**	**Obfuscation Names**	**Score**
Public	Class name	Hello	a	1
	Method name	getHello	a	
	Field name	hello	a	
Protected	Method name	getGood	b	1
	Field name	good	b	
Private	Method name	getArray	a	1
	field name	number	c	
Remove Line numbers		Line numbers	Blanked out	1
Encrypt String		GOOD	R˜ `	1
Unstructured data: One dimension array		number[]	c[]	0
Score for Layout and Data Obfuscations				5

Note: Case-sensitive

From Table 6_14, we can see that *KlassMaster v2.3* gets full mark, 4 scores, on the Layout obfuscation metrics and 1 mark on the Encrypt String. It gets zero mark on Unstructure Data metric. Therefore *KlassMaster v2.3* gets total 5 points for obfuscating my Benchmark1.

6.2.9.2 Obfuscating Benchmark2:

I used the same options as I did in obfuscating Benchmark1 to obfuscate Benchmark2. The obfuscating result is 0.class.

Run it and get the result:

```
loopNum>1 loopNum=1
```

Therefore the obfuscated class files run the same result as their unobfuscated ones do.

See the bytecode listing in the Program 6_13 below:

Program 6_13 Obfuscated Bytecode_TestControl of *KlassMaster v2.3*

```
Compiled from 0.java
public class 0 extends java.lang.Object {
    public static boolean 0;
    public static boolean 1;
    public 0();
    public static void main(java.lang.String[]);
    private static java.lang.String 0(java.lang.String);
    private static java.lang.String 1(java.lang.String);}
Method 0()
   0 aload_0
   1 invokespecial #9 <Method java.lang.Object()>
   4 return
Method void main(java.lang.String[])
   0 getstatic #72 <Field boolean 1>
   3 istore 4
   5 getstatic #49 <Field boolean 0>
   8 istore_3
   9 iconst_0
  10 istore_1
  11 iconst_0
  12 istore_2
  13 iload_3
  14 iload 4
  16 ifne 29
  19 ifeq 28
  22 iinc 1 1
  25 iinc 2 1
  28 iload_2
  29 iconst_5
  30 if_icmplt 22
  33 iload 4
  35 ifne 25
  38 iload_3
  39 iload 4
  41 ifne 53
  44 ifne 25
  47 iload_1
  48 iload 4
  50 ifne 95
  53 iconst_3
  54 if_icmple 80
  57 getstatic #12 <Field java.io.PrintStream out>
  60 ldc #3 <String "V•˚C(8^•k">
  62 invokestatic #79 <Method java.lang.String(java.lang.String)>
  65 invokestatic #69 <Method java.lang.String
0(java.lang.String)>
  68 invokevirtual #13 <Method void print(java.lang.String)>
  71 iload_3
  72 iload 4
  74 ifne 108
  77 ifeq 107
  80 getstatic #12 <Field java.io.PrintStream out>
  83 ldc #1 <String "V•˚C(8^•eE">
```

```
  85 invokestatic #79 <Method java.lang.String
1(java.lang.String)>
  88 invokestatic #69 <Method java.lang.String
0(java.lang.String)>
  91 invokevirtual #14 <Method void println(java.lang.String)>
  94 iload_3
  95 iload 4
  97 ifne 108
 100 ifeq 107
 103 iload_1
 104 iconst_1
 105 isub
 106 istore_1
 107 iload_1
 108 iconst_3
 109 if_icmpgt 103
 112 getstatic #12 <Field java.io.PrintStream out>
 115 new #7 <Class java.lang.StringBuffer>
 118 dup
 119 ldc #2 <String "V•˚C8^•">
 121 invokestatic #79 <Method java.lang.String
1(java.lang.String)>
 124 invokestatic #69 <Method java.lang.String
0(java.lang.String)>
 127 invokespecial#10<Method java.lang.StringBuffer
java.lang.String>
 130 iload_1
 131 invokevirtual #11 <Method java.lang.StringBuffer
append(int)>
 134 invokevirtual #15 <Method java.lang.String toString()>
 137 invokevirtual #14 <Method void println(java.lang.String)>
 140 return
Method java.lang.String 0(java.lang.String)
   0 getstatic #72 <Field boolean 1>
   3 istore 4
   5 aload_0
   6 invokevirtual #63 <Method char toCharArray()[]>
   9 astore_1
  10 aload_1
  11 arraylength
  12 istore_2
  13 iconst_0
  14 istore_3
  15 iload_2
  16 iconst_1
  17 iload 4
  19 ifne 108
  22 if_icmpgt 106
  25 aload_1
  26 iload_3
  27 dup2
  28 caload
  29 iload_3
  30 iconst_5
  31 irem
  32 tableswitch 0 to 3: default=84
         0: 64
         1: 69
         2: 74
         3: 79
  64 ldc #52 <Integer 47>
```

```
  66 goto 86
  69 ldc #53 <Integer 121>
  71 goto 86
  74 ldc #54 <Integer 63>
  76 goto 86
  79 ldc #55 <Integer 93>
  81 goto 86
  84 ldc #56 <Integer 32>
  86 ixor
  87 i2c
  88 castore
  89 iinc 3 1
  92 iload_2
  93 ifne 106
  96 aload_1
  97 iload_2
  98 iload 4
 100 ifne 27
 103 goto 27
 106 iload_3
 107 iload_2
 108 if_icmplt 25
 111 new #57 <Class java.lang.String>
 114 dup
 115 aload_1
 116 invokespecial #67 <Method java.lang.String(char[])>
 119 areturn
Method java.lang.String 1(java.lang.String)
   0 aload_0
   1 invokevirtual #63 <Method char toCharArray()[]>
   4 astore_1
   5 aload_1
   6 arraylength
   7 istore_2
   8 iconst_0
   9 istore_3
  10 iload_2
  11 iconst_1
  12 if_icmpgt 89
  15 aload_1
  16 iload_3
  17 dup2
  18 caload
  19 iload_3
  20 iconst_5
  21 irem
  22 tableswitch 0 to 3: default=72
         0: 52
         1: 57
         2: 62
         3: 67
  52 ldc #73 <Integer 89>
  54 goto 74
  57 ldc #74 <Integer 14>
  59 goto 74
  62 ldc #75 <Integer 88>
  64 goto 74
  67 ldc #76 <Integer 113>
  69 goto 74
  72 ldc #77 <Integer 120>
  74 ixor
```

```
75 i2c
76 castore
77 iinc 3 1
80 iload_2
81 ifne 89
84 aload_1
85 iload_2
86 goto 17
89 iload_3
90 iload_2
91 if_icmplt 15
94 new #57 <Class java.lang.String>
97 dup
98 aload_1
99 invokespecial #67 <Method java.lang.String(char[])>
102 areturn
```

Result report: Success

Comparing the obfuscated bytecodes (Program 6_13) with its unobfuscated bytecodes (Program 5_4), we can see that Program 6_13 is more complicated than Program 5_4. Program 6_13 had changed the "if …else" statement, "for" loop and "while" loop constructs.
Therefore I scored "1" mark for this metric.

Summary:

- KlassMaster v2.3 could successfully obfuscate my Benchmarks. Like its advertisement (see table 6_1), it not only could scramble names, remove debugging information such as Line numbers and encrypt String, but also can obfuscate codes with control flow.
- My Benchmarks, after being obfuscated by *KlassMaster v2.3* still functioned properly.
- Summing the results from Benchmark1 and Benchmark2, I could score most obfuscation individual Metrics for *KlassMaster v2.3* See Table 6_15.

Table 6_15 The result of obfuscating Benchmarks by *KlassMaster v2.3*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	1
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		1
	Folding one dimension array		0
Control Obfuscation	Change "if…else", "for" or "while" loop		1
Score for Layout, Data and Control Obfuscations			6

From this table, we can see that *KlassMaster v2.*3 gets total 6 marks for obfuscating my Benchmarks. Four marks are from Layout Obfuscation Metrics. One mark is from Date Obfuscation metrics and another one mark is from Control Obfuscation metrics.

6.2.10 Summary

Total nine Obfuscators were used to obfuscate my Benchmarks. See Table 6_16.

Table 6_16 The Summary of Synthetic Workload Test

Obfuscators	Benchmark1		Benchmark2		Score
	Obfuscated	Run	Obfuscated	Run	
2LKit Obfuscator v1.1	Failure		Failure		0
DashO-Pro v2.0	Success	Success	N/A	N/A	4
JCloak v3.5.3 From Force5	Success	Success	N/A	N/A	4
Jproof 1stBarrier MINI	Success	Success	N/A	N/A	3
Jshrink v1.18	Success	Success	N/A	N/A	2
Jzipper v1.08	Success	Failure	N/A	N/A	0
RetroGuard v1.1	Success	Failure	N/A	N/A	0
SourceGuard 4.0	Success	Success	N/A	N/A	4
Zelix KlassMaster2.3	Success	Success	Success	Success	6

The “blank” cell means that this step is not attempted because a prior step had failed.

All these nine obfuscators were tested by Benchmark1. Only one obfuscators were tested by Benchmark2.

Six obfuscators could not only successfully obfuscate my Benchmarks but also keep their original functions. So these six obfuscators are “Success” obfuscators in this Synthetic Workload test . Two obfuscators, Jzipper and RetroGuard, could obfuscate Benchmark1 (I did not try them by Benchmark2). However, their obfuscated codes could not pass the function test. Therefore they are unsuccessful obfuscators by my experiment. I scored them “0” mark. Only one obfuscator, 2Lkit Obfuscator v1.1, failed to obfuscate my Benchmarks. So I scored it “0”.

In my Synthetic Workload test, KlassMaster got the highest mark 6 in these 9 obfuscators. 2Lkit, Jzipper and RetroGuard got the lowest mark 0. Others which do not attend in the Synthetic Workload test also got 0 mark for the Layout, Data and Control Obfuscation metrics.

6.3 Logic Program

In this test, I measured obfuscators with the far more complicated program, Logic Program. Because of intellectual property rights, I am not going to list the logic codes here

6.3.1 Introduction

The "Logic Program" is provided by Philosophy department of UCLA in America. They wanted their logic code to be obfuscated so that they could distribute the Logic Program freely and maintain their intellectual property rights.

The purpose of the program is to assist students to learn the first order logic. It contains several modules such as doing derivations, parsing the expressions of the symbolic language, symbolizing sentences of English and so on.

The program is written in JAVA using AFC (Application Foundation Classes). It is an application that uses some facilities of windows. It contains on the order of more than 15,000 lines and 700KB.

Though I could measure these obfuscators' behavior on what is known as a "synthetic workload" of my Benchmarks, testing with a "real workload" of the Java Logic Program makes my result more realistic and correct. Therefore I added "Obfuscating Logic Program" as an individual metric to my Obfuscation Metrics.

In a word, there are two purposes for me to obfuscate the Logic Program. One is to help protect UCLA's intellectual property; the other is to assess the quality of the Java obfuscators currently available on the Internet.

I used the similar approach as I did in Synthetic workload to obfuscate the Logic Program.

Here is the process of this test:

For each obfuscator o {

Attempt to obfuscate Logic Program with obfuscator o, at highest available security options a

simply verify that the obfuscated program still function

if attempt fails, try again with a lower security options a ,

repeats (o,a)

If attempts fail, report "failure" or report "Success" (o,a)

}

Figure 6_3: Evaluation Process for Logic Program

6.3.2 Selection of obfuscators

There are many different obfuscators that are available on the Internet today. It is very important to select the correct obfuscators. It not only saves the time but also helps to analyze the result clearly, especially for a complicated program such as Logic Program. Therefore, we chose only 6 obfuscators that can successfully obfuscate our Benchmarks. They are *DashPro version2.0, Jcloak version3.5.3, Jproof 1st Barrier MINI v1.1, Jshrinkv1.18, SourceGuard 4.0 and Zelix KlassMaster version2.3.2.*

I selected these obfuscators because I would like to know whether they could obfuscate complicated Java programs and still keep their functions. I am sure that other obfuscators, which failed to obfuscate my simple benchmark programs, would not function well for obfuscating the Complicated Logic Programs.

6.3.3 Obfuscating Logic Program

Obfuscation environment:

JDK1.2.2

Window 98

Personal computer

Obfuscated object: Logic class files, logic20001209.zip

Tools:

1. Six Obfuscators:

DashPro version2.0, Jcloak version3.5.3,

Jproof 1st Barrier MINI v1.1, Jshrinkv1.18,

SourceGuard 4.0 and Zelix KlassMaster version2.3.2.

2. WinZip.

Because UCLA did not provide us details in which classes, methods or attributes they did not want to obfuscate. Therefore, I obfuscated the files with as high security level as I could.

6.3.3.1 DashO-Pro v2.0

System supported: Windows, Unix, and Macs.

File type support: .ZIP, .JAR, and *. Class (or file folder).

Project type: Applet, Application, Library, Servlet and EJB

I tried several times to obfuscate the Logic Program. Following are three of them I tried.

Trying 1.
Test Time: 22/12/2000

Main steps:

1. Input classpath:
   C:\jdk1.2.2\Jre\rt.jar
   C:\DashOPro\DashPro.jar
   C:\DashOPro\Jh.jar
   D:\lhy\project\logic_obfsource\logic20001209.zip

2. Selected Project type: Application.

   The main application class is edu.ucla.phil.logic.LogicProgram

3. Chose Obfuscation options:
   - Rename class names
   - Remove All.

Obfuscating Code Result: Error message

Class file com.ms.ui.UIEdit.class not found

Reason: missing “com” package.

After I received “com” package (afc102.zip) from UCLA, I tried it again.

Trying 2.

Test Time: 28/12/2000

Main steps:

1. Input classpath: The same as above and adding
   D:\lhy\afc102.zip

C:\windows\Java\Packages\Yl1vbrpb.zip

(Yl1vbrpb.zip contains part of “com”. afc102.zip cannot cover all need “com” according to my experiment).

2. Select the same options as above.

Obfuscating Code Result: Success

Function Test Result: Eerror message:

Java.long.NoClassDetFoundErrorr

Trying 3.

Test Time: 28/12/2000

Main steps:

1. Input classpath:

   C:\jdk1.2.2\Jre\rt.jar

   C:\DashOPro\DashPro.jar

   C:\DashOPro\Jh.jar

   D:\lhy\project\logic_obfsource\logic20001209.zip

   D:\lhy\afc102.zip

   C:\windows\Java\Packages\Yi1vbrpb.zip

2. Selected Project type: Application.

   The main application class is edu.ucla.phil.logic.LogicProgram

3. Chose Obfuscation options:
   - Do not rename class names
   - Remove None.

Obfuscating Code Result: Success.

Function Test Result: Success.

Summary:

- DashO-Pro v 2.0 could obfuscate Logic20001209 class files and the obfuscated files could run. Therefore, on my Obfuscation individual metric “Obfuscating Logic Program” I scored “1” for it.

- DashO-Pro v 2.0 provides several options such as renaming public, protected and private names and removing unused methods and fields. According to my test, no matter what options I chose, it could obfuscate the logic class files successfully. However, most of the cases, when I run the obfuscated class file, it crashed. Only when I chose the not “renaming class names” and not “remove none” options, the obfuscated codes could run correctly.

- Even though we could not obfuscate the Logic program with highest options, it does not mean that there are bugs in DashO-Pro. Changing class names and removing unused methods and fields are not safe, especially where class.forName is used or native code refers to a private members. It might be the reason that Dash-Pro need provide several entries to exclude or include for obfuscating class input. But, from my point if view, if it can automatically analyse and not obfuscate these unsafe classes or methods, it will be used more conveniently and widely because it is impossible (at least inconvenient) for a complicated program like Logic files to input "unsafe class" manually.

- DashO-Pro has a friendly and sophisticated graphic user interface with detail help information. However, when one starts it, it takes too much time to wait for the first window.

6.3.3.2 JCloak v 3.5.3

System supported: Windows, Solaris/Unix.
File type Supported: .ZIP, .JAR, and *. Class (or file folder).
Project type: Applet, Application, Library.

I also tried several times to obfuscate the Logic Program. Following gives two of them I tried.

Trying 1.

Test Time: 22/12/2000
Main Steps:

1. Select the classpath:
   C:\Program Files\JavaSoft\JRE\1.2\lib\i18n.jar
   C:\Program Files\JavaSoft\JRE\1.2\lib\plugprov.jar
   C:\Program Files\JavaSoft\JRE\1.2\lib\rt.jar
   D:\lhy\project\logic_obfsource\logic20001209.zip
   main() class: 'edu.ucla.phil.logic.LogicProgram'
2. Choose Obfuscate Options:
   a. obfuscate symbols
   b. obfuscate classnames
   c. strip linenumbers
   d. strip localsymbols

Obfuscating Code: Error Message:

WARN: 'com.ms.ui.UIDrawText' load failed, (referenced from 'edu.ucla.phil.logic.LogicProgram', error: Can't find class 'com.ms.ui.UIDrawText' on classpath "C:\Program Files\JavaSoft\JRE\1.2\lib\i18n.jar;C:\Program Files\JavaSoft\JRE\1.2\lib\plugprov.jar;C:\Program Files\JavaSoft\JRE\1.2\lib\rt.jar;C:\lhy\project\JCloak\logic.zip")
WARN: 'com.ms.fx.FxFontMetrics' load failed, (referenced from 'edu.ucla.phil.logic.ILPFontMap', error: ……

Reason: missing "com" package.

After I received "com" package (afc102.zip) from UCLA, I tried it again.

Test Time: 26/12/2000
Main steps:

1. Input classpath: The same as above and adding
   D:\lhy\afc102.zip
   C:\windows\Java\Packages\Yl1vbrpb.zip
2. Select the same options as above.

Obfuscating Code Result: Success

Function Test Result: Failure. The error message is as follow:

Class file Format Error

No matter what options I chose, the logic files ware able to obfuscate, but the transformed codes do not function. The problems might be from the complication of the Logic codes, the limitation of the evaluation versions, or the bugs inherited from the obfuscator. Because of time limited, I will leave it to my future work. Nevertheless, I scored it "0" because the obfuscated codes failed to run.

Summary:

- *JCloak* could obfuscate the Logic Program, but the obfuscated codes could not run. Therefore on my Obfuscation individual metric "Obfuscating Logic Program" I scored "0" to *JCloak.*
- *JCloak* can automatically analyze the classes referenced by logic application to determine what is "safe" to obfuscate. Therefore, it is very convenient.
- *JCloak* uses a sophisticated GUI wizard to handle the work of obfuscation parameter options.

6.3.3.3 *Jproof 1stBarrer MINI version1.1*

System supported: windows.
File type Supported : JAR.

Project type: applets, applications and libraries

Test Time: 26/12/2000
Main steps:

1. Archived the logic program, logic20001209.zip:
   - Unzip logic20001209.zip with WinZip to a folder, LogicFolder.
   - Archive all class files in LogicFolder into an archive, testLogic.jar

     prompt >jar cvf testLogic.jar *.class
2. Choose Obfuscation Options: Default, it is the highest security level

Obfuscating Code Result: Success

Function Test Result: Success.

Summary:

- *Jproof 1stBarrer MINI version1.1* could obfuscate the Logic Program and their obfuscated code could run simply. Therefore I gave it "1" point.
- Jproof just supports .JAR input form and Logic Program runs on .zip form. Therefore t is inconvenient for Jproof to obfuscate Logic Program.

6.3.3.4 *Jshrink v1.18*

System supported: Windows/NT command line.

File type Supported : *. Class.

Project type: Applet, Application, Library.

Test Time: 20/12/2000

Main step:

In the Logic Program class path, input :

Prompt>Jshrink .class

Obfuscating Code Result: Success. Message:

Processed 165 class files

Function Test Result: Success.

Summary:

- *Jshrink v1.18* can successfully obfuscate the codes and the obfuscated codes can run properly. Therefore I scored "1" to for it.
- It obfuscates codes without options.

6.3.3.5 SourceGuard Enterprise (Evaluation), Version 4.0

System supported: Windows /NT, Unix.

File type Supported : .ZIP, .JAR, and *. Class (or file folder).

Project type: Applet, Application, Library, Beans, Servlet

Test Time: 28/12/2000

Main Step:

1. Create Sourceguard project to protect appliction: project_0.
2. Application Input: D:\lhy\project\logic_obfsource\logic20001209.zip
3. Libraries Input:

   C:\windows\temp

   C:\Program Files\JavaSoft\JRE\1.2\lib\rt.jar

   D:\lhy\project\logic_obfsource\logic20001209.zip

   D:\lhy\afc102.zip

   C:\windows\Java\Packages\Yi1vbrpb.zip

Obfuscation options and result:

Choice 1: both rename and enable code pruning

The result was successful. However, when I run the obfuscated codes, I got an error message: "Java.long.illegalAccessError".

Choice 2: only rename.

The result SourceGuard can both obfuscate Logic Program and keep their function.

Summary:

- *SourceGuard 4.0* could successfully obfuscate the codes and the obfuscated codes could run properly. Therefore I scored "1" for it.
- Even though, *SourceGuard 4.0* could analyse class files to generate dependency information, it could not keep the obfuscated codes' function when I chose pruning code option to obfuscate the Logic Program.
- *SourceGuard 4.0* provides many obfuscation option sets. However, I could not obfuscate the Logic Program with the highest security level option set.
- *SourceGuard 4.0* has a sophisticated and friendly GUI.

6.3.3.6 Zelix KlassMaster version 2.3.2

System supported: Windows /NT, Unix.

File type Supported ：.ZIP, .JAR, and *. Class (or file folder).

Project type: Applet, Application, Library.

Test Time: 27/12/2000

Main steps:

3. Double click “ZKM.jar” to start ZKM
4. Input classpath:

   C:\KLASSM~1\ZKM.JAR

   C:\PROGRAM FILES\JAVAOSFT\JRE\1.2\LIB\rt.jar

   D:\lhy\project\logic_obfsource\logic20001209.zip

   D:\lhy\afc102.zip

   C:\windows\Java\Packages\Yi1vbrpb.zip

5. Choose Obfuscation Options:
   - Rename and remove all
   - Obfuscate Control Flow: aggressive
   - Encrypt String Literal: flow obfuscate.
   - Exclude main class names

Obfuscating Code Result: Success. Output message:

165 class loaded 19 seconds 10587k of memory used.

Function Test Result: Success.

I also used this way to obfuscate other logic program versions such as logic2001208, logic20001218, logic20010105, logic20010106b, logic20010107, and logic20010107c. According to the feedback from UCLA, these obfuscated files functioned properly.

Summary:

- *KlassMaster* could successfully obfuscate the codes and the obfuscated codes could run properly. Therefore I scored “1” for it.
- I also had successfully obfuscated all other logic program versions that were required by UCLA. These obfuscated files all function well.
- KlassMaster has a sophisticated and friend GUI. It is easy to use.

6.3.3.7 Summary

Six obfuscators, which have successfully obfuscated my Benchmarks, have been tested on this stage. Five of them successfully obfuscate the Logic Program and their obfuscated codes could run properly. See Following table.

Table 6_17: The Summary of Logic Program Test

Obfuscators	Logic Program (logic2001209)		Score
	Obfuscated	Run	
DashO-Pro v2.0	Success	Success	1
JCloak v3.5.3 From Force5	Success	Failure	0
Jproof 1stBarrier MINI	Success	Success	1
Jshrink v1.18	Success	Success	1
SourceGuard 4.0	Success	Success	1
Zelix KlassMaster2.3	Success	Success	1

From above Table 6_17, we can see that all six obfuscators could obfuscate the Logic Program, logic20001209.zip. Five out of them, their obfuscated codes could run properly.
JCloak v3.5.3 could obfuscate the Logic program. However, its obfuscated codes failed to run. So it got "0" point on this stage. Other five got "1" score each. Those obfuscators that did not qualify to be tested by the Logic Program would be scored "0" because of their failure on prior steps.

Many obfuscators provide various options for different security levels. Some options are easy to choose correctly, some are not. For example, SourceGuard and DashO-Pro, their options are not easy to choose correctly because they needs too many manually analyses and input. The classes obfuscated with these options do not function well even though the classes can be obfuscated successfully. But some obfuscators like JCloak can automatically analyses which classes are "safe" to be obfuscated. However, when they obfuscate a complicated program like Logic Program, they might get some trouble. From my point of view, both kinds of obfuscators are usually difficult to obfuscate the complicated programs with full options because of the limited knowledge about the obfuscated programs or the quality of the obfuscators.

Most obfuscators support multiple file types such as .JAR, .ZIP and .CLASS. Some obfuscators such as Jproof just support only JAR file. It is good for the program whose executable file is JAR file. But it is not convenient for the ZIP file runner to change its form.

6.4 Decompilation by SourceAgain

In this test, I used SourceAgain decompiler to decompile all the obfuscated Benchmarks and obfuscated Logic Programs. The purpose of this test is to assess the obfuscators and assign two of my individual metrics. They are "Decompiling Obfuscated Benchmarks" and "Decompiling Obfuscated Logic Program".

6.4.1 Decompiling Obfuscated Benchmarks

The Obfuscated Benchmarks were all from the results of Synthetic Workload test. From Table 6_14, we can see that there are 6 obfuscators that could successfully obfuscate my Benchmarks. They are *DashPro v2.0, Jcloak v3.5.3, Jproof 1st Barrier MINI v1.1, Jshrinkv1.18, SourceGuard 4.0 and Zelix KlassMaster version2.3.2*. The class files obfuscated by these six obfuscators would be decompiled in this step.

Here is the process of this test:

```
For each set of obfuscated class files f {
    For each obfuscated benchmark I {
        Attempt to decompile obfuscated benchmark  i  by SourceAgain,
        verify that the decompiled program still function
        If attempts fail, report "Success" or report "Failure" (f,i)
    }
}
```

Figure 6_4: Decompilation Process for

6.4.1.1 DashO-Pro v2.0

Dasho-pro obfuscated only Benchmark1. Therefore its output of the obfuscated files consist of TestOBF.class and b.class.

Following shows the steps of decompiling these files:

1. Decompile them by running:

   Prompt >srcagain *.class

   Get the result: _b.java and _TestOBF.java

   Therefore SourceAgain could decompile this obfuscated Benchmark1 (See detail code in appendix)

2. Recompiling the decompiled files:

   - Rename _TestOBF.java and _b.java to TestOBF.java and b.java
   - Recompile them by running:

     Prompt >javac TestOBF.java

     and

     Prompt >javac b.java

     Get class files:

     TestOBF.class and b.class

3. Run them to test their functions

   Prompt >java TestOBF

   Get result:

   ```
   HELLO
   GOOD
   number=1
   ```

   Therefore these decompiled class files still function properly.

Result report: **Failure.**

This Obfuscated Benchmarks could be successfully decompiled by SourceAgain. Therefore I assigned “0" to the metric “Decompiling Obfuscated Benchmarks” of Dasho-Pro v2.0.

6.4.1.2 Jcloak v3.5.3

Jcloak v3.5.3 obfuscated only Benchmark1. Therefore its output of the obfuscated files consist of TestOBF.class and C0.class.

Following shows the steps of decompiling these files:

1. Decompile them by running:

   Prompt >srcagain *.class

   Get the result _TestOBF.java and _C0.java

   Therefore SourceAgain could decompile this obfuscated Benchmark1 (See detail code in appendix)

2. Recompiling the decompiled files:

- Rename _TestOBF.java and _C0.java to TestOBF.java and C0.java
- Recompile them by running:

  Prompt >javac C0.java

  Prompt >javac TestOBF.java

  Get class files:

  C0.class and TestOBF.class

3. Run them to test their functions

   Prompt >java TestOBF

   Get result:

```
HELLO
GOOD
number=1
```

   Therefore these decompiled class files still function properly.

Result report : **Failure.**

This Obfuscated Benchmarks could be successfully decompiled by SourceAgain. Therefore I assigned “0" to the metric “Decompiling Obfuscated Benchmarks” of *Jcloak v3.53.*

6.4.1.3 Jproof 1st Barrier MINI v1.1

Jproof 1st Barrier MINI v1.1 obfuscated only Benchmark1. Therefore its output of the obfuscated files consist of TestOBF.class and dl.class.

Following shows the steps of decompiling these files:

1. Decompile them by running:

   Prompt >srcagain *.class

   Get the result _TestOBF.java and _dl.java

   Therefore SourceAgain could decompile this obfuscated Benchmark1 (See detail code in appendix)

2. Recompiling the decompiled files:

   - Rename _TestOBF.java and _dl.java to TestOBF.java and dl.java
   - Recompile them by running:

     Prompt >javac dl.java

     Prompt >javac TestOBF.java

     Get class files:

     dl.class and TestOBF.class

3. Run them to test their functions

   Prompt >java TestOBF

   Get result:

```
HELLO
GOOD
number=1
```

Therefore these decompiled class files still function properly.

Result report : **Failure**

This Obfuscated Benchmarks could be successfully decompiled by SourceAgain. Therefore I assigned “0" to the metric “Decompiling Obfuscated Benchmarks” of *Jproof 1st Barrier MINI v1.1*.

6.4.1.4 Jshrinkv1.18

Jshrinkv1.18 obfuscated only Benchmark1. Therefore its output of the obfuscated files consist of TestOBF.class and Hello.class.

Following shows the steps of decompiling these files:

1. Decompile them by running:

   Prompt >srcagain *.class

   Get the result _TestOBF.java and _Hello.java

   Therefore SourceAgain could decompile this obfuscated Benchmark1 (See detail code in appendix_B)

2. Decompiling the decompiled files:

   - Rename _TestOBF.java and _Hello.java to TestOBF.java and Hello.java
   - Recompile them by running:

     Prompt >javac Hello.java

     Prompt >javac TestOBF.java

     Get class files:

     Hello.class and TestOBF.class

3. Run them to test their functions

   Prompt >java TestOBF

   Get result:

```
HELLO
GOOD
number=1
```

Therefore these decompilated class files still function properly.

Result report : **Failure**

This Obfuscated Benchmarks could be successfully decompiled by SourceAgain. Therefore I assigned “0" to the metric “Decompiling Obfuscated Benchmarks” of *Jshrinkv1.18*.

6.4.1.5 SourceGuard 4.0

SourceGuard 4.0 obfuscated just Benchmark1. Therefore its output of the obfuscated files consist of TestOBF.class and a.class.

Following shows the steps of decompiling these files:

1. Decompile them by running:

   Prompt >srcagain *.class

   Get the result _TestOBF.java and _a.java

   Therefore SourceAgain could decompile this obfuscated Benchmark1 (See detail code in appendix_B)

2. Recompiling the decompiled files:

   - Rename _TestOBF.java and _a.java to TestOBF.java and a.java
   - Recompile them by running:

     Prompt >javac a.java

     Prompt >javac TestOBF.java

   Get class files:

   a.class and TestOBF.class

3. Run them to test their functions

   Prompt >java TestOBF

   Get result:

   ```
   HELLO
   GOOD
   number=1
   ```

   Therefore these decompiled class files still function properly.

Result report : **Failure**

This Obfuscated Benchmarks could be successfully decompiled by SourceAgain. Therefore I assigned “0" to the metric “Decompiling Obfuscated Benchmarks” of *SourceGuard 4.0*.

6.4.1.6 KlassMaster v2.3

KlassMaster v2.3 obfuscated both Benchmark1 and Benchmark2. Their output of the obfuscated files consist of b.class, a.class and 0.class.

When I decompiled them by running:

Prompt >srcagain *.class

SourceAgain run into infinity.

SourceAgain can not decompile the obfuscated classes properly. See the incomplete codes on the Appendix.

Result report : **Failure**

SourceAgain could not successfully decompile this Obfuscated Benchmarks. Therefore I assigned “1" to the metric “Decompiling Obfuscated Benchmarks” of *KlassMaster v2.3*

6.4.1.7 Summary:

Summarising all the decompiling result from above. I got following Table 6_18

Table 6_18 The Summary of Decompilation Obfuscated Benchmarks

Obfuscators	Benchmark1		Benchmark2		Score
	Decompiled	Run	Decompiled	Run	
DashO-Pro v2.0	Failure	Failure	N/A	N/A	0
JCloak v3.5.3 From Force5	Failure	Failure	N/A	N/A	0
Jproof 1stBarrier MINI	Failure	Failure	N/A	N/A	0
Jshrink v1.18	Failure	Failure	N/A	N/A	0
SourceGuard 4.0	Failure	Failure	N/A	N/A	0
Zelix KlassMaster v2.3	Success		Success		1

The “blank” cell means that this step is not attempted because the code could not be decompiled.

From the Table 6_18, we can see that six obfuscators were tested. Five out of them got “0”. SourceAgain successfully decompiled all the Benchmark class files obfuscated by these five obfuscators. Only Zelix KlassMaster v2.3 could got 1 score. It is the only one that could prevent SourceAgain from decompiling its obfuscated Benchmarks

6.4.2 Decompiling Obfuscated Logic Program

The Obfuscated Logic Programs were all from the results of Logic Program test. From Table 6_17, we can see that there are 5 obfuscators that could successfully obfuscate my Benchmarks. They are *DashPro v2.0, Jproof 1st Barrier MINI v1.1, Jshrinkv1.18, SourceGuard 4.0 and Zelix KlassMaster version2.3.2*. The class files obfuscated by these five obfuscators would be decompiled in this step.

In this test, I would not verify the decompiled programs’ function. I just decompiled the obfuscated files and checked the results.

Here is the process of this test:

For each set of obfuscated class files f {
Attempt to decompile obfuscated Logic Program i by SourceAgain,
If attempts fail, report "Success" or report "Failure" (f,i)
}

Figure 6_5 Decompilation Process for Logic

I followed this process to test the all the Logic files obfuscated by the five obfuscators.
The way I used to decompile these obfuscated files is the same as I did in Decompiling Obfuscated Benchmarks. Here is the result for Decompiling Obfuscated Logic Program.

Table 6_19 The Summary of Decompiling Obfuscated Logic Program

Obfuscators	Logic Program (logic20001209) Decompiled	Score
DashO-Pro v2.0	Failure	0
Jproof 1stBarrier MINI	Failure	0
Jshrink v1.18	Failure	0
SourceGuard 4.0	Failure	0
Zelix KlassMaster2.3	Success	1

From the Table 6_19, we can see that five obfuscators were tested. Four out of them got "0". SourceAgain could successfully decompile the class files obfuscated by these four obfuscators. Only *Zelix KlassMaster v2.3* could got 1 score. It is the only one that could prevent SourceAgain from decompiling its obfuscated Logic Programs.

6.4.3 Summary

Combining and summarizing above results, Table 6_18 and Table 6_19, I got the following Table 6_20.

Table 6_20 The Summary of Decompilation

Obfuscators	Benchmarks		Logic Program		Total Scores
	Decompiled	Score	Decompiled	Score	
DashO-Pro v2.0	Failure	0	Failure	0	0
JCloak v3.5.3 From Force5	Failure	0			0
Jproof 1stBarrier MINI	Failure	0	Failure	0	0
Jshrink v1.18	Failure	0	Failure	0	0
SourceGuard 4.0	Failure	0	Failure	0	0
Zelix KlassMaster v2.3	Success	1	Success	1	2

The "blank" cell means that this step is not attempted because prior steps had failed.

From the table, I concluded that:

- Total six obfuscators were tested on this step. All six were tested by Decompiling Obfuscated Benchmarks. Five out of the them were tested by Decompiling Obfuscated Logic Program.
- *Zelix KlassMaster v2.3* is the only one that could prevent SourceAgain from decompiling both its obfuscated Benchmarks and obfuscated Logic Programs. Others' obfuscated files all could be decompiled by SourceAgain.
- On this step, *Zelix KlassMaster v2.3* got the highest marks, 2 scores. One is from Decompiling Obfuscated Benchmarks. Other one is from Decompiling Obfuscating Logic Program. Other obfuscators got 0 mark because they fail to protect the code obfuscated by them from decompilation.

6.6 Assess Metrics

Until now, I have finish all the tests and got the test results from each test. In this step, I will use the results to assess all the metrics of the tested objects, 13 obfuscators.

6.6.1 2Lkit Obfuscator v1.1

From Table 6_16: The Summary of Synthetic Workload Test, the result shows that *2Lkit Obfuscator v1.1* is "Failure" and gets 0 score. It did not qualify to attend other after tests such as "Logic Program" test and "Decompilation by SourceAgain" Test and got 0 point on each test. Therefore, I give it all 0 point for its metrics.

6.6.2 Cloakware

From Table 6_3: The Summary of Installation Test, the result shows that *Cloakware* is "Failure" and gets 0 score. Cloakware is an obfuscation service, not an obfuscator program for public release, so I gave it all 0 point for its metrics.

6.6.3 DashO-Pro v2.0

DashO-Pro v2.0 was tested by all test steps: Installation, Synthetic Workload, Logic Program and Decompilation by SourceAgain. Here is the table I sum it up from these test results:

Table 6_21: Obfuscation Metrics of *DashO-Pro v2.0*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	1
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		0
	Data unstructured		0
Control Obfuscation	Change "if…else", "for" or "while" loop		0
Obfuscating Logic Program			1
Decompiling Obfuscated Benchmarks			0
Decompiling Obfuscated Logic Program			0
Total Scores			5

This table shows that:

- *DashO-Pro v2.0* could successfully scramble public names, protected names and private names and remove Line numbers. So it got 1 mark for each Layout Obfuscation individual metrics. But it got 0 mark for each individual metrics of Data Obfuscations and Control Obfuscations because it did not have any behaviors on these metrics. The results are from Table 6_5: The result of obfuscating Benchmarks by *DashO-Pro v2.0*.
- *DashO-Pro v2.0* could successfully obfuscate the complicated Logic Program. So got 1 mark for this metric. The result is from Table 6_17: The Summary of Logic Program Test.

- It could not prevent SourceAgain from decompiling its obfuscated codes. So I score 0 for its decompiling obfuscated code metrics. The results are from Table 6_20: The Summary of Decompilation Test.
- The total number of scores it got is 5.

6.6.4 Hashjava

From Table 6_3, the result shows that *Hashjava* is "Failure" and gets 0 score because it failed to download. It did not qualify to attend other later tests such as "Synthetic Workload" test, "Logic Program" test and "Decompilation by SourceAgain" test and got 0 point on each test. Therefore, I gave it all 0 point for its metrics.

6.6.5 JCloak v3.5.3 from Force5

JCloak v3.5.3 was tested by all test steps: Installation, Synthetic Workload, Logic Program and Decompilation by SourceAgain. Here is the table I summed it up from these test results:

Table 6_22 Obfuscation Metrics of JCloak v3.5.3

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	1
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		0
	Data unstructured		0
Control Obfuscation	Change "if…else", "for" or "while" loop		0
Obfuscating Logic Program			0
Decompiling Obfuscated Benchmarks			0
Decompiling Obfuscated Logic Program			0
Total Scores			4

This table shows that:

- *JCloak v3.5.3* could successfully scramble public names, protected names and private names and remove Line numbers. So it got 1 mark for each Layout Obfuscation individual metrics. But it got 0 mark for each individual metrics of Data Obfuscations nd Control Obfuscations because it did not have any behaviors on these metrics. The results are from Table 6_7: The result of obfuscating Benchmarks by *JCloak v3.5.3*.

- *JCloak v3.5.3* could obfuscate the complicated Logic Program. But the codes obfuscated by it could not run. So it got 0 mark for this metric. The result is from Table 6_17
- It could not prevent SourceAgain from decompiling its obfuscated Benchmarks. So I score 0 for this metric. *JCloak v3.5.3* did not attend the Decompiling Obfuscated Logic Program test. So it got 0 on this metric. The results are from Table 6_20.
- The total number of scores it got is 4.

6.6.6 Jproof 1stBarrier MINI v1.1

Jproof 1stBarrier MINI v1.1 was tested by all test steps: Installation, Synthetic Workload, Logic Program and Decompilation by SourceAgain. Here is the table I summed it up from these test results:

Table 6_23 Obfuscation Metrics of *Jproof 1stBarrier MINI v1.1*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	0
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		0
	Data unstructured		0
Control Obfuscation	Change “if…else”, “for” or “while” loop		0
Obfuscating Logic Program			1
Decompiling Obfuscated Benchmarks			0
Decompiling Obfuscated Logic Program			0
Total Scores			4

This table shows that:

- *Jproof 1stBarrier MINI v1.1* could scramble public class names but not public method names and public field names. However, according to my metric design, I had to score 1 mark for it. It could successfully obfuscate private names and remove Line numbers. So it got 1 mark for each. But it lost mark for scrambling protected names. I scored 0 for each individual metrics of Data Obfuscation and Control Obfuscation because it had not any behaviors on these metrics. The results are from Table 6_9: The result of obfuscating Benchmarks by *Jproof 1stBarrier MINI v1.1*.

- *Jproof 1stBarrier MINI v1.1* could successfully obfuscate the complicated Logic Program. So got 1 mark for this metric. The result is from Table 6_17.
- It could not prevent SourceAgain from decompiling its obfuscated codes. So I score 0 for its decompiling obfuscated code metrics. The results are from Table 6_20.
- The total number of scores it got is 4.

6.6.7 **JOBE – the obfuscator**

From Table 6_3, the result shows that *JOBE*– the obfuscator is "Failure" and gets 0 score because it failed to download. It did not qualify to attend other after tests such as "Synthetic Workload" test, "Logic Program" test and "Decompilation by SourceAgain" test and got 0 point on each test. Therefore, I gave it all 0 point for its metrics.

6.6.8 Jshrinkv1.18

Jshrinkv1.18 was tested by all test steps: Installation, Synthetic Workload, Logic Program and Decompilation by SourceAgain. Here is the table I summed it up from these test results:

Table 6_24 Obfuscation Metrics of *Jshrinkv1.18*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	0
		Protected	0
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		0
	Data unstructured		0
Control Obfuscation	Change "if…else", "for" or "while" loop		0
Obfuscating Logic Program			1
Decompiling Obfuscated Benchmarks			0
Decompiling Obfuscated Logic Program			0
Total Scores			3

This table shows that:

- *Jshrinkv1.18* could successfully obfuscate private names and remove Line numbers. So it got 1 mark for each relative metric. But it lost mark for scrambling public names and protected names. It also got 0 score for each individual metrics of Data Obfuscation and

Control Obfuscation because it had not any behaviors on these metrics. The results are from Table 6_11: The result of obfuscating Benchmarks by *Jshrinkv1.18.*

- *Jshrinkv1.18* could successfully obfuscate the complicated Logic Program. So got 1 mark for this metric. The result is from Table 6_17.
- It could not prevent SourceAgain from decompiling its obfuscated codes. So I score 0 for its decompiling obfuscated code metrics. The results are from Table 6_20.

The total number of scores it got is 3.

6.6.9 Jzipper v1.08

From Table 6_16, the result shows that *Jzipper v1.08* is "Failure" and gets 0 score. Though it could obfuscate the Benchmark, the obfuscated code could not run properly. So it failed on this test. It did not qualify to attend other after tests such as "Logic Program" test and "Decompilation by SourceAgain" Test and got 0 point on each test. Therefore, I gave it all 0 point for its metrics.

6.6.10 Obfuscate™ v1.1

From Table 6_3, the result shows that *Obfuscate™ v1.1* is "Failure" and gets 0 score because it failed to run. It did not qualify to attend other after tests such as "Synthetic Workload" test, "Logic Program" test and "Decompilation by SourceAgain" test and got 0 point on each test. Therefore, I gave it all 0 point for its metrics.

6.6.11 RetroGuard v1.1

From Table 6_16, the result shows that *RetroGuard v1.1* is "Failure" and gets 0 score. Though it could obfuscate the Benchmark, the obfuscated code could not run properly. So it failed on this test. It did not qualify to attend other after tests such as "Logic Program" test and "Decompilation by SourceAgain" Test and got 0 point on each test. Therefore, I gave it all 0 point for its metrics.

6.6.12 SourceGuard 4.0

SourceGuard 4.0 was tested by all test steps: Installation, Synthetic Workload, Logic Program and Decompilation by SourceAgain. Here is the table I summed it up from these test results:

Table 6_25 Obfuscation Metrics of *SourceGuard 4.0*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	1
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		0
	Data unstructured		0
Control Obfuscation	Change "if…else", "for" or "while" loop		0
Obfuscating Logic Program			1
Decompiling Obfuscated Benchmarks			0
Decompiling Obfuscated Logic Program			0
Total Scores			5

This table shows that:

- *SourceGuard 4.0* could successfully scramble public names, protected names and private names and remove Line numbers. So it got 1 mark for each Layout Obfuscation individual metrics. But it got 0 mark for each individual metrics of Data Obfuscations and Control Obfuscations because it did not any have behaviors on these metrics. The results are from Table 6_13: The result of obfuscating Benchmarks by *SourceGuard 4.0*.
- *SourceGuard 4.0* could successfully obfuscate the complicated Logic Program. So got 1 mark for this metric. The result is from Table 6_17.
- It could not prevent SourceAgain from decompiling its obfuscated codes. So I score 0 for its decompiling obfuscated code metrics. The results are from Table 6_20.
- The total number of scores it got is 5.

6.6.13 Zelix KlassMaster2.3

Zelix KlassMaster2.3 was tested by all test steps: Installation, Synthetic Workload, Logic Program and Decompilation by SourceAgain. Table 6_26 is the result that I summed it up from these test results.

Table 6_26: Obfuscation Metrics of *Zelix KlassMaster2.3*

ITEMS	CONTENTS		SCORES
Layout Obfuscation	Scramble Names	Public	1
		Protected	1
		Private	1
	Remove Line numbers		1
Data Obfuscation	Encrypt String		1
	Data Unstructured		0
Control Obfuscation	Change "if…else", "for" or "while" loop		1
Obfuscating Logic Program			1
Decompiling Obfuscated Benchmarks			1
Decompiling Obfuscated Logic Program			1
Total Scores			9

This table shows that:

- *Zelix KlassMaster2.3* could successfully scramble names and Line numbers. So it got 1 mark for each Layout Obfuscation individual metrics. It also could encrypt String and gets 1 mark for this metric. But it lost mark for Data Unstructured metric. On Control Obfuscation metric, it also got 1 mark for changing "if…else", "for" or "while" loop. The results are from Table 6_15: The result of obfuscating Benchmarks by *Zelix KlassMaster2.3*.
- *Zelix KlassMaster2.3* could successfully obfuscate the complicated Logic Program. So got 1 mark for this metric. The result is from Table 6_17.
- It could prevent SourceAgain from decompiling its obfuscated codes. So I score 1 for its decompiling obfuscated code metrics. The results are from Table 6_20.
- The total number of scores it got is 9.

Comparative results

In this section, I compare the 13 obfuscators based on the Obfuscation Metrics criteria I designed on the section 4 and the results from my experiment in section 6. Here is the table I concluded from these two sections.

Table 7_1 **Comparative results**

No.	Obfuscators	Installation	Synthetic Workload	Logic Program	Decompilation by SourceAgain	Score
1	Zelix KlassMaster2.3	Success	Success	Success	Success	9
2	DashO-Pro v2.0	SUCCESS	Success	Success	Failure	5
3	SourceGuard 4.0	Success	Success	Success	Failure	5
4	JCloak v3.5.3 From Force5	Success	Success	Failure		4
5	Jproof 1stBarrierMINI v1.1	Success	Success	Success	Failure	4
6	Jshrinkv1.18	Success	Success	Success	Failure	3
8	2Lkit Obfuscator v1.1	Success	Failure			0
9	Cloakware	Failure				0
10	JOBE – the obfuscator	Failure				0
11	Jzipper v1.08	Success	Failure			0
12	Obfuscate™ v1.1	Failure				0
13	RetroGuard v1.1	Success	Failure			0
Total scores						30

The “blank” cell means that this step is not attempted because a prior step had failed.

The table shows that:

- There are 13 obfuscators that had been tested in section 6. The tests divided into 4 steps: Installation, Synthetic Workload, Logic Program, Decompilation by SourceAgain. The scores are from the final work, Assess Metric, in the section 6.
- In the first step: Installation, nine out of thirteen obfuscators could successfully this step.
- In the second step: Synthetic Workload, six obfuscators could successfully obfuscate my Benchmarks and the obfuscated codes still function.
- In the third step: Logic Program, five obfuscators could successfully obfuscate the complicated Logic Programs.
- In the last step: Decompilation by SourceAgain, only one, KlassMaster v2.3 could prevent SourceAgain from decompiling the codes obfuscated by it.
- *KlassMaster v2.3* is also the only one that passed all four steps. Four obfuscators: *DashO-Pro v2.0, SourceGuard 4.0, Jproof 1stBarrierMINI v1.1*and Jshrinkv1.18 passed three steps. JCloak v3.5.3 from Force5 could pass two steps. Two obfuscators passed one step. Others failed just on the first step.
- The total number of the scores is 30. Six out of thirteen got at least more than 1 makrs marks. Others got all got 0. Here is the figure that represents the comparative result of the non-zero point obfuscators .

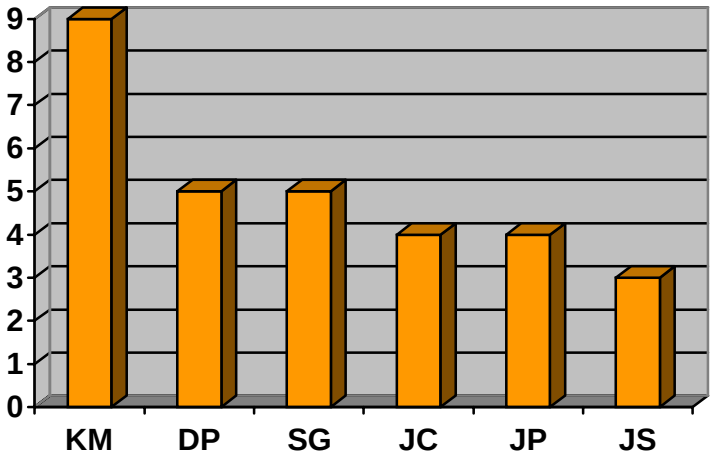


Figure 7_1 A Comparative Result Chart

Note:

KM: *Zelix KlassMaster v2.3*

DP: *DashO-Pro™ v2.0*

SG: *SourceGuard v4.0*

JC: *JCoak v3.5.3*

JP: *Jproof 1stBarrier MINI v1.1*

JS*: Jshrink v1.1*

The graph shows:

- KlassMaster wins the highest mark in my obfuscation metrics, it is 9. Jshrink get the lowest marks, 2 points. There are two obfuscators got 5 marks and other two got 4 marks.

Conclusion

Obfuscators are the major tools for protecting Java program from reverse engineering. Given so many obfuscators available on the Internet today, how can one know which is better?

In order to answer this question, I have designed my Obfuscation Metrics and my Benchmarks as a criterion. I have also developed an "evaluation flowchart" to test all available obfuscators on my synthetic workload and the "logic Program" from Philosophy department of UCLA in America

The 13 available obfuscators scored between 0 and 9 points on my 10 point metric. Many obfuscators got 0 mark. None got full marks on my metrics.

Most of obfuscators got marks only on their layout obfuscation metrics. However, they could not prevent the decompiler, SourceAgain, from decompiling the codes obfuscated by them. Only one obfuscator, which uses some techniques of data obfuscations and control obfuscations, could make SouceAgain fail to decompile the obfuscated codes. From my point of view, future obfuscators would be improved if they apply more techniques of data obfuscations and control obfuscations. This might be a good way for an obfuscator to challenge decompilers. It is also a challenge technical field.

Various obfuscators provide different protection levels for Java programs. Some can obfuscate code with high level. They obfuscate code not only with layout obfuscations and data obfuscation, but also with control obfuscations. Some obfuscators scramble private names, which I consider to be a “low level” obfuscation.

Many obfuscators are safe when obfuscating codes. However some are not safe. Even though they can obfuscate codes, the obfuscated codes function improperly. Some can successfully obfuscate the simple codes, my Benchmarks, but they cannot properly obfuscate the complicated program, Logic Program. From my point if view, “safe” is most important. If an “obfuscator” could not keep obfuscated codes’ original functions, it would not be called an obfuscator even

though it has a very powerful technique. Some of obfuscators tested in my experiment lost their marks because they were not “safe”.

Most of the obfuscators provide run-time options that may increase obfuscation quality. However, for some option settings, they produced non-functional obfuscated code but were not penalized for this bug in my study.

In future evaluations of obfuscators, error messages and non-functional output should affect the comparison.

References

[1] Christian Collberg and Clark Thomborson. Watermarking, Tamper-Proofing, and Obfuscation

- Tools for Software Protection, Computer Science Department Technical Report 170, University of Auckland, February 2000, 15 pp.

www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborson2000a/index.html

[2] Christian Collberg, Clark Thomborson and Douglas Low. Breaking abstractions and unstructuring data structures, In IEEE *International Conference on Computer Languages*, ICCL'98, Chicago, IL, May 1998.

www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97/index.html

[3] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and

stealthy opaque constructs. In *Principles of Programming Languages 1998*, POPL'98, San Diego, CA, January 1998.

www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThobersonLow98a/index.html

[4] Douglas Low. Java Control Flow Obfuscation (full document in PDF, 1 MB)

www.cs.auckland.ac.nz/~cthombor/Pbubs/dlowthesis.pdf

[5] Qusay H. Mahmoud. Java Tip22: Protect your bytecodes from reverse engineering/decompilation, 1997.

http://www.javaworld.com/javatips/jw-javatip22/html.

[6] Benoit Marchal and Meurrens. Java Decompilation and Reverse Engineering Part 1. Digital Cat's Java™ Resource Center.

http://www.javacats.com/US/articles/decompiler1.html

[7] Benoit Marchal and Meurrens. Java Decompilation and Reverse Engineering Part 2. Digital Cat's Java™ Resource Center.

http://www.javacats.com/US/articles/decompiler2.html

[8] Robert Macgregor, Dave Durbin, John Owlett, Andrew Yeomans. Java Network Security. 1998. ISBN 0-13-76529-9.

[9] Mary Campione, Kathy Walrath, Alison Huml and the Tutorial Team. The Java™ Tutorial Continued. The Rest of the JDK™. 1999. ISBN 0201485583.

Ofuscation tools

2LKit obfuscator v1.1 www.2Lkit.com download date: December 29, 2000

Cloakware www.cloakware.com

DashO-Pro™ www.preemptive.com

HashJava: www.meurrens.org

JCloak from Force5 Software: www.force5.com.

JProof 1stBarrier MINI www.jproof.com

JOBE-The java Obfuscator: www.priment.com.

JShrink: www.e-t.com/jshrinkdoc.html.

JZipper: www.vegatech.net/jzipper

Nei Aggarwal's Obfuscate and ObfuscatePro: www.jammconsulting.com

RetroGuard: www.retrologic.com.

SourceGuard: www.4thpass.com/sourceGuard
Zelix KlassMaster: www.zelix.com/klassmaster

APPENDIX

Decompiled Code Obfuscated by Dash-Pro v20.

_TestOBF.java:

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//
import java.io.PrintStream;
public class TestOBF {
    public TestOBF()
    {
        System.out.println( a.a( "HELLO" ) );
        System.out.println( a() );
        System.out.println( "number=" + b() );
    }
    public b a = new b();
    public String b = "";
    public int[] c = new int[10];
    private int b()
    {
        c[0] = 1;
        return c[0];
    }
    public String a()
    {
        b = "GOOD";
        return b;
    }
    public static void main(String[] String_1darray1)
    {
        new TestOBF();
    }
}
```

_b.java:

```
 //
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//
public class _b {
    public b()
    {
        int int1 = 1;
    }
    public String a(String String1)
```

```
    {
        return String1;
    }
}
```

Decompiled Code Obfuscated by JCloak v3.5.3

_TestOBF.java:

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//

import java.io.PrintStream;

public class TestOBF {

    public TestOBF()
    {
        System.out.println( c.a( "HELLO" ) );
        System.out.println( b() );
        System.out.println( "number=" + a() );
    }

    public C0 c = new C0();
    protected String d = "";
    private int[] e = new int[10];

    private int a()
    {
        e[0] = 1;
        return e[0];
    }
    protected String b()
    {
        d = "GOOD";
        return d;
    }
    public static void main(String[] String_1darray1)
    {
        new TestOBF();
    }
}
```

_C0.java:

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//

class C0 {
    public C0()
    {
        int int1 = 1;
    }
```

```
    public String a(String String1)
    {
        return String1;
    }
}
```

Decompiled Code Obfuscated by Jproof 1stBarrier MINI v1.1

_TestOBF.java:

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//

import java.io.PrintStream;

public class TestOBF {

    public TestOBF()
    {
        System.out.println( hello.getHello( "HELLO" ) );
        System.out.println( getGood() );
        System.out.println( "number=" + lt() );
    }
    public dl hello = new dl();
    protected String good = "";
    private int[] Lt = new int[10];

    private int lt()
    {
        Lt[0] = 1;
        return Lt[0];
    }

    protected String getGood()
    {
        good = "GOOD";
        return good;
    }

    public static void main(String[] String_1darray1)
    {
        new TestOBF();
    }
}
```

_dl.java.java:

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//
class dl {

    public dl()
    {
        int int1 = 1;
    }
```

```
    public String getHello(String String1)
    {
        return String1;
    }
}
```

Decompiled Code Obfuscated by Jshrink v1.18

_TestOBF.java:

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//

import java.io.PrintStream;

public class TestOBF {

    public TestOBF()
    {
        System.out.println( hello.getHello( "HELLO" ) );
        System.out.println( getGood() );
        System.out.println( "number=" + m_function1() );
    }
    public Hello hello = new Hello();
    protected String good = "";
    private int[] m_function1 = new int[10];

    private int m_function1()
    {
        m_function1[0] = 1;
        return m_function1[0];
    }

    protected String getGood()
    {
        good = "GOOD";
        return good;
    }

    public static void main(String[] String_1darray1)
    {
        new TestOBF();
    }
}
```

_Hello.java

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//

class Hello {
    public Hello()
    {
        int int1 = 1;
    }
```

```
    public String getHello(String String1)
    {
        return String1;
    }
}
```

Decompiled Code Obfuscated SourceGuard 4.0

_TestOBF.java

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//

import java.io.PrintStream;

public class TestOBF {

    public TestOBF()
    {
        System.out.println( b.a( "HELLO" ) );
        System.out.println( b() );
        System.out.println( "number=" + a() );
    }

    public a b = new a();
    protected String c = "";
    private int[] a = new int[10];

    public static void main(String[] String_1darray1)
    {
        new TestOBF();
    }
    protected String b()
    {
        c = "GOOD";
        return c;
    }

    private int a()
    {
        a[0] = 1;
        return a[0];
    }
}
```

_a.java:

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//
class a {

    public a()
    {
        int int1 = 1;
    }
```

```
    public String a(String String1)
    {
        return String1;
    }
}
```

Decompiled Code Obfuscated by Zelix KlassMaster2.3

_b.java:

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//
import java.io.PrintStream;
public class b {
    public b()
    {
        int int1 = d;

        a = new a();
        b = "";
        c = new int[10];
        System.out.println( a.a( a( "]\t\u0003\u0006X" ) ) );
        System.out.println( b() );
        System.out.println( a( "{9\"(rgq" ) + a() );
        if( int1 != 0 )
            a.a = (a.a) ? false : true;
    }

    public a a;
    protected String b;
    private int[] c;
    static int d;

    private int a()
    {
        c[0] = 1;
        return c[0];
    }

    protected String b()
    {
        b = a( "R\u0003\u0000\u000E" );
        return b;
    }

    public static void main(String[] String_1darray1)
    {
        new b();
    }
```

_a.java:

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//
class a {
    public a()
```

```
    {
        int int1 = 1;
    }
    public static boolean a;
    public String a(String String1)
    {
        return String1;
    }
}
```

_0.java:

```
//
// SourceAgain (TM) v1.10h (C) 2000 Ahpah Software Inc
//

import java.io.PrintStream;
public class type0 {
    public static boolean m_function2;
    public static boolean m_function3;
    public static void main(String[] String_1darray1)
    {
        boolean boolean5 = m_function3;
        boolean boolean4 = m_function2;
        int int2 = 0;
        int int3 = 0;
        boolean temp_boolean6 = boolean4;
        if( !boolean5 )
        {
            if( temp_boolean6 )
            {
                ++int2;
                ++int3;
                temp_boolean6 = int3;
            }
            else
                temp_boolean6 = int3;
        }
        for( ;; )
        {
            if( temp_boolean6 >= true )
            {
                if( !boolean5 )
                {
                    boolean temp_boolean7 = boolean4;
                    boolean temp_boolean8;
label_51:
                    {
label_43:
                        {
                            if( !boolean5 )
                            {
                                if( !temp_boolean7 )
                                {
                                    temp_boolean8 = int2;
                                    if( boolean5 )
                                        break label_43;
                                }
```

```
                                else
                                {
                                    ++int3;
                                    temp_boolean6 = int3;
                                    continue;
                                }
                            }
                            if( temp_boolean6 > true )
                            {
                                System.out.print( m_function2(
                                m_function3(
                                "V\u001B\bC(8\u0002\n\u0012k" ) ) );
                                temp_boolean8 = boolean4;
                                if( boolean5 )
                                    break label_51;
                                if( !temp_boolean8 )
                                {
                                    temp_boolean8 = int2;
                                    break label_51;
                                }
                            }
                            System.out.println( m_function2(
                            m_function3(
                            "V\u001B\bC(8\u0002\n\u0010eE" ) ) );
                            temp_boolean8 = boolean4;
                        }
                        if( !boolean5 )
                        {
                            if( temp_boolean8 )
                            {
                                --int2;
                                temp_boolean8 = int2;
                            }
                            else
                                temp_boolean8 = int2;
                        }
                    }
                    while( temp_boolean8 > true )
                    {
                        --int2;
                        temp_boolean8 = int2;
                    }
                    System.out.println( m_function2( m_function3(
                    "V\u001B\bC(8\u0002\n\u0011" ) ) + int2 );
                    return;
                }
                else
                {
                    ++int3;
                    temp_boolean6 = int3;
                }
            }
            else
            {
                ++int2;
                ++int3;
                temp_boolean6 = int3;
            }
        }
    }
```