P- 1807

JAVA BYTECODE PROTECTION USING OBFUSCATION TECHNIQUE

A PROJECT REPORT

Submitted by

ATHITHYA. U 71203104005

in partial fulfillment for the award of the degree

of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

KUMARAGURU COLLEGE OF TECHNOLOGY
COIMBATORE – 641 006

ANNA UNIVERSITY: CHENNAI 600 025
APRIL 2007

P-1807

ANNA UNIVERSITY: CHENNAI 600 025

BONAFIDE CERTIFICATE

Certified that this project report "**Java Bytecode Protection Using Obfuscation Technique**" is the bonafide work of **Athithya. U** who carried out the project under my supervision.

[illegible]	[illegible] 23/4
SIGNATURE	SIGNATURE
Prof. S. Thangasamy,	Ms.Rajini. S,
HEAD OF THE DEPARTMENT,	SUPERVISOR,
	Senior Lecturer,
Department of Computer Science and Engineering,	Department of Computer Science and Engineering,
Kumaraguru College of Technology,	Kumaraguru College of Technology,
Coimbatore – 641 006.	Coimbatore – 641 006.

The candidate with University Register No.: **71203104005** was examined by us in the project viva-voce examination held on 24·04·2007

[illegible] 24·4	[illegible] 24/4/07
INTERNAL EXAMINER	EXTERNAL EXAMINER

DECLARATION

We hereby declare that the project entitled "**JAVA BYTECODE PROTECTION USING OBFUSCATION TECHNIQUE**" is a record of original work done has not been submitted to Anna University or any institutions, for the fulfillment of course study.

The report is in partial fulfillment of the requirements for the award of the degree of Bachelor of Computer Science and Engineering of Anna University, Chennai.

Place : Coimbatore
Date : 23·04·2007

U. Athithya

(Athithya. U)

ABSTRACT

There exist several obfuscation tools for preventing Java bytecode from being decompiled. Most of these tools simply scramble the names of the identifiers stored in a bytecode by substituting the identifiers with meaningless names. However, the scrambling technique cannot determine cracker very long. I have proposed several advanced obfuscation techniques that make Java bytecode impossible to recompile or make the decompiled program difficult to understand and to recompile. The crux of my approach is to over use an identifier. That is, an identifier can denote several entities, such as types, fields, and methods, simultaneously.

An additional benefit is that the size of the bytecode is reduced because fewer and shorter identifier names are used. Furthermore, I propose several techniques to intentionally introduce syntactic and semantic errors into the decompiled program while preserving the original behaviours of the bytecode. Thus, the decompiled program would have to be debugged manually. Although my basic approach is to scramble the identifiers in Java bytecode, the scrambled bytecode produced with this techniques is much harder to crack than that produced with other identifier scrambling techniques. Furthermore, the run-time efficiency of the obfuscated bytecode is also improved because the size of the bytecode becomes smaller after obfuscation.

ACKNOWLEDGEMENT

I express my heartfelt gratitude to **Dr. Joseph V. Thanikal,** Principal, Kumaraguru College of Technology for making available excellent infrastructure that has enabled me for the successful completion of the project.

I also express my gratitude to **Dr. S. Thangasamy,** Professor and Head of the Department of Computer Science and Engineering, who has inspired me throughout the project.

I express my sincere and special thanks to my guide and project coordinator **Ms. Rajini. S**, Senior Lecturer, Department of Computer Science and Engineering, who has helped me with valuable suggestions at each and every stage of this project. Her inspiration and encouragement were invaluable and she stood as a pillar of strength and support.

I also express my sincere thanks and gratitude to our class advisor **Mr. S. Mohanavel**, Senior Lecturer, Department of Computer Science and Engineering for his guidance and support throughout this project.

I would like to thank all the faculty members and student friends for their support and encouragement in my endeavour. Finally I would like to thank my parents and my brothers who gave great support and inspired me throughout this project.

Above all, the grace of GOD led me to finish this project successfully.

CONTENTS

LIST OF FIGURES

INRODUCTION

1. INTRODUCTION

Traditionally, a program is compiled to native code (or machine code). Most of the symbolic information is stripped off when the program is compiled. The identifiers that denote variables and functions in the source program become addresses in the compiled program.

Decompiling such a program, though difficult, is still possible. Because no methods can absolutely protect a program from decompilation attacks by experienced crackers.

We usually consider a protection method successful, if it can make the cracking work costly in terms of time and effort. Cracking becomes valueless when the cost is more than that of rewriting a program. Therefore, one of the basic rules is to prevent the decompilation to be done automatically with tools (i.e. decompilers).

Generally all the software companies will give the class files of their project to the clients. From that the client misuses the resource, by recompiling the class file they can extract the source code. Using that they will change the copyrights or they will modify the source code without the proper acknowledgement of the author. This makes Java programmers to think about the security.

ANALYSIS OF THE PROBLEM

2. ANALYSIS OF THE PROBLEM

The proposed system has to be analyzed well for its merits and demerits. Our main aim is to make use of all its merits and to remove the demerits of it. First the proposed system is analyzed then the merits of the system are discussed.

The requirement specification of the proposed system should also be analyzed so that the hardware and software requirement of the proposed system is known in order to execute our application. This helps the user in developing his application.

The documentation of the proposed system is made, which helps us to collect the details that are describing the system. And the graphical representation of the system and its activities are presented in order to make the user understand the system. Then the records and description of the system elements are analyzed and maintained.

2.1 PROBLEM DEFINITION

When customers buy software, they are provided with the executable file of that software. In case of Java projects, when the source code is compiled corresponding class files are obtained and the customer is provided with those class files.

The program codes can obtained from these class files and executable files. When a class file is decompiled using a decompiler, the Java source code can be obtained. A decompiler produces source code from a binary executable or a class file. This is the inverse operation of a compiler, which produces an executable from source code written in a particular language.

Programs that are written in Java are particularly amenable to decompilation because Java source code is typically compiled to Java bytecode. Java bytecode is a platform independent abstraction layer, which can be executed through a Java Virtual Machine.

The core design of Java bytecode makes decompilation considerably easier than most development languages currently in use. Java bytecode contains interface and type information, which is not normally present in executables written in other source languages.

Java decompiler examines Java bytecode to produce a flow graph. From this flow graph, Java decompiler determines a collection of statements like while, for,if etc., which when compiled could produce such a graph. Therefore the resulting source code may not be identical to the source code, but will have the same semantics. The hackers can debug those code to obtain the original source code. The source obtained in such a way is sold illegally by altering the copy right details of the program. The extraction of the source code is also done by cracking using crackers.

To avoid the easy extraction of source code by the hackers from the executables Java bytecode protection is implemented using obfuscation technique.

2.2 EXISTING SYSTEM

Obfuscation tools are one of the major defenses against the decompilers. Obfuscation transforms clear bytecode into more obscure bytecode. The goal of obfuscation is to make the decompiled program much harder to understand so that a cracker has to spend more time and effort on the obfuscated bytecode. Most of the existing obfuscation tools simply scramble the symbolic information (identifiers) in the constant pool (Dr. Java, Eastridge, Hoeniche, Plumb, Retrologic). Usually, a meaningful name is substituted by a meaningless name.

2.3 DRAWBACKS OF THE EXISTING SYSTEM

Most of the existing obfuscation tools simply scramble the names of the identifiers stored in a bytecode by substituting the identifiers with meaningless names [8]. However the scrambling technique cannot deter a determined cracker very long and also other tools insert dead codes inside the original code so the size of the source code will be increased.

When we use that tool we are getting following disadvantages:

- Introduce additional computations
- Increase size of the bytecode
- Reduce run-time efficiency of the program.

2.4 PROPOSED SYSTEM

Proposed System achieves better identifier scrambling. Based on the approach, several techniques are introduced to make the bytecode much harder to understand and, sometimes, make the decompiled program not recompilable. The basic approach is to endow an identifier

with as much information as possible. An identifier can denote several types, several fields, and several methods at the same time in the obfuscated bytecode. The cracker is confused because an identifier is identified not only by its name but also by the context it exists. An additional benefit is that the size of the bytecode is reduced because long, meaningful names are replaced by shorter, meaningless names.

We also propose several techniques to purposely introduce certain hidden compilation errors into the obfuscated bytecode so that the decompiled program cannot be compiled again. Therefore, a cracker has to spend a lot of time debugging the decompiled program manually. The basic approach and these techniques make a Java bytecode file harder to crack. Furthermore, the run-time efficiency of an obfuscated program is also improved.

2.5 BENEFITS OF THIS PROJECT

Obfuscation is a process to hide a Java program's internal information so that it keeps the program's semantics but makes the program difficult to decompile. If the obfuscated program is decompiled anyway, the source code obtained through a decompiler will not be easy for a person, even a programmer, to understand that code. The cracker has to spend more time for understanding the code.

It can detect and remove unused classes, fields, methods, and attributes. It can then optimize bytecode and remove unused instructions. Finally, it can rename the remaining classes, fields, and methods using

short meaningless names. The resulting jars are smaller and harder to reverse-engineer.

The line number information will be removed from the file. All the command lines present in the class file will also be removed .so it is difficult to reverse engineering that code. We are not saying that our project stop the decompilation process. But it will protect the cracker from understanding the original source code.

2.6 FEASIBILITY STUDY

The major benefits of Java is Portability, the compiled program can run on most platforms. A Java program is compiled to platform independent bytecode. In order to achieve platform independence, instead of the traditional memory addresses, Java uses symbolic references to link entities from different libraries (including the standard and proprietary libraries). Therefore, the names of types, fields, and methods are stored in a constant pool within a bytecode file. These names and the simple Stack-machine instructions facilitate the decompilation of the bytecode file.

There are many free or commercial Java decompiler, the decompiled program is almost identical to the original source program. These decompilers become the lethal weapon of intellectual property piracy.

The performance of the java code will not be changed after the obfuscation process. Only the identifier and symbolic names used in original file will be changed.

PROGRAMMING ENVIRONMENT

3. PROGRAMMING ENVIRONMENT

3.1 HARDWARE REQUIREMENTS

Processor	:	Pentium IV 2.4 GHZ
RAM	:	128 MB
Monitor	:	17" monitor
Hard Disk	:	80 GB
Keyboard	:	Standard Keyboard with 104 Keys
Mouse	:	Serial Mouse

3.2 SOFTWARE REQUIREMENTS

Language	:	Java
Platform	:	Windows 2000

DESCRIPTION OF THE LANGUAGE USED

4. DESCRIPTION OF THE LANGUAGE USED

Java was developed by the company Sun Microsystems. From 1995 onwards, Java became a very popular language for the internet. Java is widely used in both server side and client side applications. The Java programming language is a high-level language that can be characterized by all of the following buzzwords:

- Object oriented
- Distributed
- Multithreaded Dynamic
- Architecture neutral
- Portable High performance
- Robust Secure

In the Java programming language, all source code is first written in plain text files ending with the `java` extension. Those source files are then compiled into `.class` files by the Java compiler (`javac`). A `.class` file does not contain code that is native to your processor; it instead contains byte codes- the machine language of the Java Virtual Machine. The Java launcher tool (`java`) then runs your application with an instance of the Java Virtual Machine.

Because Java is simple, it is easy to read and write. Obfuscated Java isn't nearly as common as obfuscated C. There aren't a lot of special cases or tricks that will confuse beginner. About half of the bugs in C and C++ programs are related to memory allocation and deallocation. Therefore the second important addition Java makes to providing bug-free code is automatic memory allocation and deallocation.

DESIGN AND IMPLEMENTATION

5. DESIGN AND IMPLEMENTATION

5.1 FUNCTIONAL REQUIREMENTS

In Java, an application consists of one or more packages. A programmer may divide his own application into packages. He may also use the packages in the standard library and proprietary libraries. Usually only the part of an application that is developed by the programmer is distributed. The proprietary libraries are not distributed because of the copyright restrictions. The part of a program that will be obfuscated by the obfuscation techniques is called the obfuscation scope.

Generally, only the programmer-developed part of an application is protected. The packages that serve, as utilities in the standard and proprietary libraries are not obfuscated. However, the obfuscation scope is not necessary limited to the packages written by the programmer.

When an application is not big enough to confuse the cracker, the standard and proprietary libraries could be included in the obfuscation scope. However the redistribution of the obfuscated proprietary libraries may violate the copyright.

5.2 CLASS FILE STRUCTURE

The java compiler will convert the source file into Bytecode format (class file) format. The structure of the class file is described here. The Java class file contains everything a JVM needs to know about one Java class or interface. In their order of appearance in the class file, the major

5.2.1 Access Flags

The first two bytes after the constant pool, the access flags, indicate whether or not this file defines a class or an interface, whether the class or interface is public or abstract, and (if it's a class and not an interface) whether the class is final.

5.2.2 THIS Class

The next two bytes, the *this class* component, are an index into the constant pool array. The constant referred to by *this class*, constant pool [this_class], has two parts, a one-byte tag and a two-byte name index. The tag will equal CONSTANT_Class, a value that indicates this element contains information about a class or interface. Constant_pool [name_index] is a string constant containing the name of the class or interface.

The *this class* component provides a glimpse of how the constant pool is used. *This class* itself is just an index into the constant pool. When a JVM looks up constant_pool [this_class], it finds an element that identifies itself as a CONSTANT_Class with its tag.

The JVM knows CONSTANT_Class elements always have a two-byte index into the constant pool, called name index, following their one-byte tag. So it looks up constant_pool [name_index] to get the string containing the name of the class or interface.

5.2.3 Super Class

Following the *this class* component is the *super class* component, another two-byte index into the constant pool. Constant _pool [super_class] is a CONSTANT_Class element that the points to the name of the super class from which this class descends.

5.2.4 Interfaces

The interfaces component starts with a two-byte count of the number of interfaces implemented by the class (or interface) defined in the file. Immediately following is an array that contains one index into the constant pool for each interface implemented by the class. Each interface is represented by a CONSTANT_Class element in the constant pool that points to the name of the interface.

5.2.5 Fields

The fields component starts with a two-byte count of the number of fields in this class or interface. A field is an instance or class variable of the class or interface. Following the count is an array of variable-length structures, one for each field. Each structure reveals information about one field such as the field's name, type, and, if it is a final variable, its constant value. Some information is contained in the structure itself, and some is contained in constant pool locations pointed to by the structure.

The only fields that appear in the list are those that were declared by the class or interface defined in the file; no fields inherited from super classes or super interfaces appear in the list.

5.2.6 Methods

The methods component starts with a two-byte count of the number of methods in the class or interface. This count includes only those methods that are explicitly defined by this class, not any methods that may be inherited from superclasses. Following the method count are the methods themselves.

The structure for each method contains several pieces of information about the method, including the method descriptor (its return type and argument list), the number of stack words required for the method's local variables, the maximum number of stack words required for the method's operand stack, a table of exceptions caught by the method, the bytecode sequence, and a line number table.

5.2.7 Attributes

Attributes give general information about the particular class or interface defined by the file. The attributes section has a two-byte count of the number of attributes, followed by the attributes themselves. For example, one attribute is the source code attribute; it reveals the name of the source file from which this class file was compiled. JVMs will silently ignore any attributes they don't recognize.

5.3 INPUT AND OUTPUT DESIGN

Data enter the system as input and this is the data on which the processing is performed. In the Input Design, the user-oriented inputs are converted in to computer recognizable format. It is necessary to ensure that the input design is suitable. Input Design is the part of the design

phase, which requires more attention. The Input data is accepted and it can be readily used for data processing or can be stored in the database for future use. During input design it is made sure that the data to be processed by the system is collected and entered into the system efficiently, according to the specified requirements and with minimum number of errors and with minimum redundancy.

Here in this project, the input that is to be given is a jar file. The processing is done on that jar file and the output is stored in the specified jar file. The input jar file is added by choosing the Add Input option and the location of the input file is mapped by finding the source of the input file.

Once the required output is decided, we can determine what the inputs should be. The quality of the system input determines the quality of system output. Output is the information defined to the user through the information system.

The output is also a jar file, but this output jar file is obfuscated and shrinked, i.e., the size of the file is reduced. The shrinking is performed by removing the useless comments and methods in the program.

5.4 OVERVIEW OF THE PROJECT

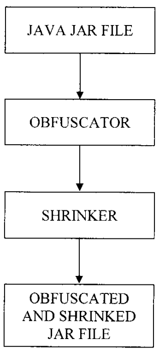


Fig 5.1 Project Overview

The objective of the project is to obfuscate and reduce the size of the input file. First the input file is obtained which is a Java source code file. After that the input file is obfuscated using the obfuscator that is implemented. Then the obfuscated file is shrinked is reduce the size of the output file to save the wastage of the memory space. Finally obfuscated and shrinked output file is obtained.

5.5 MODULES

The project is divided into three modules.

Module 1: Reading the valid jar file and parsing it.
Module 2: Implementation of the Obfuscation and shrinking techniques
Module 3: Resultant obfuscated jar file generation.

5.6 DATA FLOW DIAGRAM

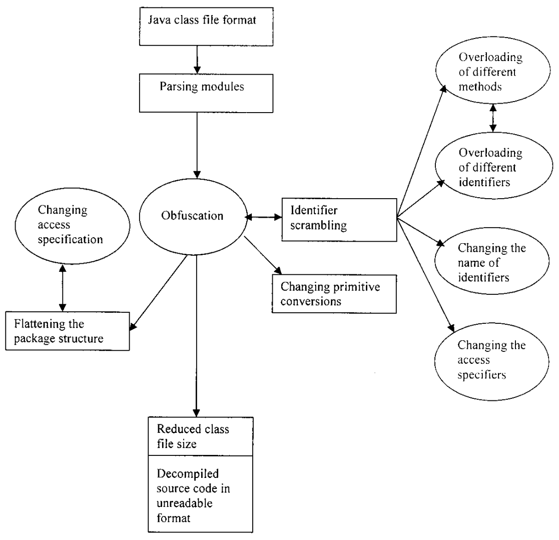

Java class file format
Parsing modules
Overloading of different methods
Overloading of different identifiers
Changing access specification
Obfuscation
Identifier scrambling
Changing the name of identifiers
Changing primitive conversions
Flattening the package structure
Changing the access specifiers
Reduced class file size
Decompiled source code in unreadable format

Fig 5.2 Data Flow Diagram

5.7 IMPLEMENTATION

5.7.1 Obfuscation

The word obfuscation literally translates to 'making something less clear and harder to understand. In computing, obfuscation is used to transform the code into a form that is functionally identical to the original code but is much more difficult to understand and reverse engineer using tools.

We are not assuming here that obfuscation will make the code impossible to reverse engineer. The aim is to increase the cost of reverse engineering the code, so that it becomes infeasible.

5.7.2 Obfuscation Techniques

The following are the some of the techniques used in project,

- Identifier scrambling
- Illegal identifier
- Overloading unrelated method
- Flattening the package structure

5.7.3 Basic Approach

Java specification, an identifier in a Java program may denote

- a package
- a top-level type (either a class or an interface)
- a nested type (either a class or an interface)
- a field, a method
- a parameter (of a method, a constructor, or an exception handler)
- a local variable

However, not all of them are kept in the Bytecode file after compilation. Only the identifiers that denote the first five items in the above list are stored in the bytecode.

By default, parameters and local variables are stripped off from the bytecode and become the memory addresses of the local variable array in the corresponding Stack frame If the debug-info option of the compiler is enabled, the names of parameters and local variables will be stored in the LocalVariableTable in the bytecode.

The LocalVariable- Table can be removed by disabling the option (which is, the default setting of the Java compiler). If the LocalVariableTable is not available, Java decompilers usually automatically generate names sequentially for parameters and local variables. Though it is possible to rename the variables in the LocalVariableTable to make the decompilation process more difficult, a smarter decompiler may simply ignore these modified names and generate new variable names instead. Since we cannot prevent the decompilers from generating names for parameters and local variables, names in the LocalVariableTable are not candidates for obfuscation. The candidates for obfuscation are the first five items.

On the other hand, not all of the candidates can be obfuscated. When an application runs, the Java virtual machine (JVM) dynamically loads and links the referenced types into the runtime environment. The bytecode file that stores the referenced type is located by a symbolic reference–the fully qualified name of a class or an interface. These

symbolic references cannot be changed. Hence, only the candidates that reference entities in the obfuscation scope will be obfuscated. The candidates that reference entities outside the obfuscation scope (which generally denote entities in the standard library or the proprietary libraries) should not be obfuscated. The identifiers that denote entities in the obfuscation scope need further investigation.

The following four groups of identifiers should not be obfuscated (these groups are called the Exception groups):

Exception group 1:

The instance method that implements an abstract method of a superclass (or a superinterface) that is outside the obfuscation scope.

Exception group 2:

The instance method that overrides an inherited method of a superclass that is outside the obfuscation scope.

Exception group 3:

The entities that are explicitly designated by the programmer to remain unchanged.

Exception group 4:

Java supports polymorphism. An instance method is dynamically dispatched at run time based on the signature of the method. The signature of a method consists of the name of the method and the number and the types of the formal parameters. Note that the return type and the throws clause are not part of the signature of a method in Java. Because

the name of a method M outside the obfuscation scope is retained, the name of the method that is in the obfuscation scope and overrides the method M should be retained as well.

Otherwise, JVM cannot find the overriding methods based on the signature of M. These retained methods belong to Exception groups 1 and 2.

When a package is in the obfuscation scope, sometimes, it is necessary to keep some parts of a package outside the scope. For example, the main method is the entry point of an application. Therefore, the name of the main method should be retained.

Furthermore, a proprietary library may export certain types and certain methods as the interface of the library. The names of these exported types and methods should be retained as well. These retained entities are said to belong to Exception group 3.

The callback mechanism is heavily used in the event model of the graphical-user-interface (GUI) library of Java.

When the caller of an instance method N that serves as a callback function is outside the obfuscation scope, the name of N should not be obfuscated. Otherwise, the caller cannot find method N at run time. On the other hand, if the caller is also in the obfuscation scope, the symbolic reference can be changed to the new, obfuscated name of N. In this case, the name of N can be obfuscated.

Determining whether a method is a callback function is a complex task. We first have to construct a call-graph by examining the whole application and all the referenced libraries. Through the call-graph, callback methods can be identified. However, this construction would take a lot of time. We made a safe assumption here. Generally, the class that contains callback methods implements specific interfaces or extends a specific class.

The caller of the callback method takes a parameter whose type is the superinterface or the superclass. The actual object that contains the callback method is passed as a parameter and a callback method is invoked through the polymorphism mechanism.

Based on this assumption, all the callback methods whose names should be retained will belong to Exception group 1 or 2. Fields, static methods, and nested types are statically resolved by the Java compiler. Once the bytecode is generated, the JVM will not change the resolution.

Therefore, the names of fields, static methods, and nested types that are in the obfuscation scope may be changed arbitrarily.In summary, the targets of obfuscation include the names of the entities in the obfuscation scope except those in Exception groups 1, 2, and 3.

5.7.4 Identifier Scrambling

Our basic obfuscation approach is to reuse an identifier as often as possible. This approach makes an identifier heavily overloaded and hence confusing to a cracker. An identifier can denote several types, fields, and methods at the same time after obfuscation.

When the obfuscated bytecode is decompiled, the meaning of an identifier is not determined only by its name but also by the context it exists. A cracker is confused because he has to identify the context in which an identifier exists. An additional benefit is that the size of the bytecode is reduced because fewer and shorter names are used.

There are two hierarchical structures in a Java application.The first is the package structure. An application consists of one or more packages. A package may contain zero or more subpackages and top-level types (i.e., classes and interfaces). The subpackages and the top-level types in a package cannot have the same name. However, a subpackages or a top-level type may have the same name as the enclosing package. Suppose that sequentially generated identifiers are used in an obfuscation tool. The generation of identifiers may be restarted for every package.

The second structure is the inheritance structure. Every class, except the Object class, has a direct super class. A class may implement zero or more interfaces. An interface can inherit zero or more interfaces. An interface and a class implicitly inherit the Object class if they do not inherit any other interfaces and other classes, respectively. The depth of the inheritance hierarchy is unlimited.

Through the inheritance structure, we can identify instance methods that belong to Exception groups 1 and 2; these instance methods will not be renamed. Furthermore, the instance methods that have an overriding relationship and are in the obfuscation scope must be renamed consistently, if they are ever renamed.

```
package p;
public class A extends t.M {}
public class B extents A implements t.J {}

package p.q;
public interface I {}
public class C {}
public class D extents C implements I {}

package t;
public class M {}
public interface J {}
```

Notice that the Object class is implicitly included in every Java application. Suppose that the packages p and p.q are in the obfuscation scope. The corresponding package structure and inheritance structure are shown in Figs. 4.4.2 and 4.4.3, respectively. The shadowed area (surround by a dotted curve) denotes the obfuscation scope.

The basic obfuscation approach consists of the following five steps:

a) Analyze all the bytecode files that are in the obfuscation scope and construct the package structure and the inheritance structure.
b) Traverse the package structure from root to leaf. During the traversal, use sequentially generated names to substitute the package names and top-leveltype names. The generation of names is restarted for each package node.

For example, suppose that the generating sequence of names is a, b, c...

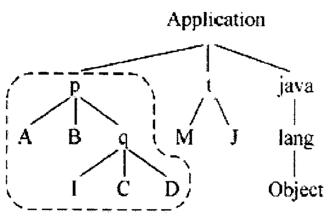


Fig.5.3(a) Package Structure

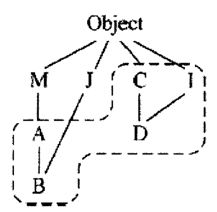


Fig : 5.3(b) Inheritance Structure

After this Step, the package structure in Fig. 5.3 (a) will become the one in Fig. 5.3 (b) and the inheritance structure in Fig.5.4 (a) will become the one in Fig.5.4 (b)

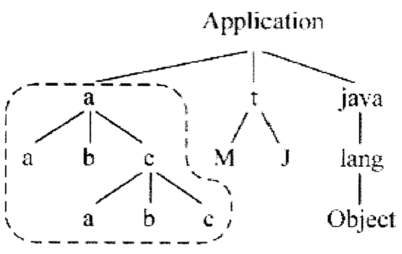


Fig 5.4 (a) Obfuscated Package Structure

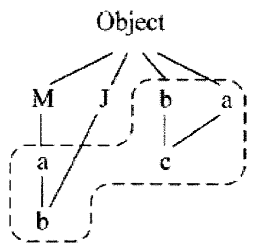


Fig 5.4 (a) Obfuscated Inheritance Structure

c) Traverse the inheritance structure from root to leaf. During the traversal, perform the following steps for each type T.

c(1) Restart the generation of names. Replace the names of all the fields with the sequentially generated new names.

c(2) Restart the generation of names. Replace the names of all the nested types with the sequentially generated new names. A type in this paper means a class or an interface. A type that is declared within another type is called a nested type (it is also called a member type).There is no limit on the depth of type nesting. According to the Java language specification, a nested type cannot have the same name as any of its enclosing types. Therefore, the name used by the enclosing types must be avoided.

c(3) Restart the generation of names. Replace the name of a method M with the sequentially generated new names. When M is an instance method, check whether M belongs to Exception group 1 or 2. If so, the original name of M remains unchanged. Otherwise, a super type S of T must also be in the obfuscation scope. When there is an instance method N in S with the same original signature as M, use the same name of N for M. Otherwise, use a newly generated name for M.Notice that the newly generated name cannot be the same as the name of the method in S. If so, it may result in a new overriding relationship. An inherited instance method cannot be overridden arbitrarily. Otherwise, the invoked method may be changed unexpectedly. Recursively repeat steps c1 to c3 for each nested type.

d) Update all the symbolic references in the obfuscation scope with the substituted names.

e) Save the obfuscated code to each bytecode file in the obfuscation scope.

Notice that in Java source programs, a field may be shadowed by the fields in the subtypes or the nested types. A nested type may be shadowed by a nested type in the subtypes or other more deeply nested types. Arbitrarily changing the name of a field, a static method, and a nested type may cause unexpected shadowing or obscurity among the names. However, these shadowing and obscurity do not change the behavior of the application because those static entities are determined statically at compile time. Even if they have the same name, JVM knows which type should be checked. Renaming will not change the entities to be used. Therefore, steps c1 and c3 are simplified because we do not have to worry about the obscurity problem that has been resolved by the compiler.

5.7.5 Illegal Identifiers

The names in the constant pool of the bytecode could be changed to use illegal characters, keywords, Boolean literals or the null literal. When the obfuscated bytecode is decompiled and then compiled, these identifiers will result in compilation errors

5.7.6 Flattening the Package Structure

Usually, types (classes and interfaces) that provide related functions are grouped in a package. This is the purpose of introducing the

package structure in Java. Packages help programmers to organize their programs.

However, packages also help a cracker to analyze the byte code. The functions of the types in a package are easier to understand after some of them have been understood. Another purpose of the package structure is to control the accessibility of the members (i.e., fields, methods, and nested types) in a type. The members declared as protected in a type T can be accessed by the types in the same package that T belongs to and by the subtypes of T.

The members that are declared as default-access (that is, no access modifier is specified) in a class T can be accessed by the types in the same package that T belongs to. Flattening the package structure is to put all the types that comprise an application into a single package. A cracker cannot make use of the package structure to crack an application.

Though flattening the package structure will extend the accessible range of the protected and the default-access members to the whole application. Extending the accessible range in the bytecode, not the source code, will not change the behavior of an application.

The Java compiler has already checked the accessibility of the members. Although the Java virtual machine will check the accessibility again when the application runs, program behavior will not be affected.

P- 1807

5.7.7 Overloading Unrelated Methods

Java supports dynamical loading and reflection. Through the method Class.forName (''MyClass- Name''), JVM can load the named class at run time. The name of the class could be determined at run time.

Therefore, we cannot determine whether to keep the name of the type or to change the value of the parameter through static analysis at obfuscation time. so, the names of the types that will be dynamically loaded should be retained and be indicated manually. The members of a dynamically loaded type need further investigation. Most of the methods in the class Class return the public members of the dynamically loaded type. Therefore, the public members of a dynamically loaded type should not be renamed.

There might be chances to use the protected and the default access members of a dynamically loaded type through reflection. This situation rarely happens and should be considered as problematic. Nevertheless, the protected and the default-access members of a dynamically loaded type should not be renamed. Only the private members of a dynamically loaded type can be renamed arbitrarily. In Java, methods are determined by their signatures.

This means that two methods are considered different if they have the same name but different numbers or types of the formal parameters. Such methods are called the overloaded methods.

For further obfuscating bytecode, we can use the same name for all the methods that have different names and different number or types of the formal parameters. Therefore, a program is further obscured because the methods can only be differentiated by the number and the types of their formal parameters.

There are some important issues when unrelated methods are overloaded. First, the methods in a subclass should preserve the overriding relationship among the Super classes and the subclass. Because the overriding relationship affects the method to be invoked at run time, the overriding relationship should not be modified when the methods are renamed. Notice that the overloading relationship among the methods of the super classes and the subclass need not be preserved. Furthermore, we can make use of the relationships between static methods and instance methods of the super classes and the subclass to provide another layer of protection.

```
class X ¦
  void m (float a) ¦...¦
  void p (long b) ¦...¦
  void f() ¦
    int s = 1;
    m(s);
  ¦
¦
```

Fig 5.4 Normal method structure

The method invocation m(s) in the method f invokes the method m. When the bytecode is obfuscated by overriding unrelated methods, the decompiled program (produced by the cavaj decompiler) becomes

```
class X ¦
  void g (float a) ¦...¦
  void g (long b) ¦...¦
  void g() ¦
    int i = 1;
    g(i);
  ¦
¦
```

Fig 5.5 Obfuscated method structure

When the decompiled program is compiled to bytecode again, the method invocation g(i) will invoke the method g(long) (the original p(long)) because an integer value is converted to a long value rather than a float value by a widening conversion. In contrast, in the original code, it is g (float) that will be invoked.

The behavior of the decompiled program is silently changed by the obfuscation. This kind of a silent semantic change provides better protection for bytecode. The technique of overloading unrelated methods Cleverly introduces semantic changes in the decompiled program. All the methods already exist; no bogus methods are introduced.

This semantic change is almost impossible to discover, even by an expert. However, not all the decompilers we examined are fooled by this technique. Some decompilers, such as (Hoeniche and JReverse Pro), add an explicit casting conversion to each parameter when necessary.

TESTING

6. TESTING

Testing is vital to the success of the system. System testing makes a logical assumption that if all the parts of the system are correct, the goal will be successfully achieved. Inadequate testing or non-testing leads to errors that may not appear until months later. This create two problems

- The time lag between the cause and appearance of the problem.
- The effect of system errors on files and records with in the system. A small system error can conceivably exploded into much larger problem. Effective early in the process translates directly into long term cost savings from a reduced number of errors.

This project is tested using the following testing techniques,

- Unit testing
- Integration testing
- White box testing
- Black box testing
- Output testing
- System testing.

6.1 UNIT TESTING

A unit test is a procedure used to validate that a particular module of source code is working properly. Normally programs are written as a series of individual modules, which are separately tested.

In this project unit testing is performed over all modules. Reading a jar file is the first module which is tested for different forms of jar files.

In the next module, implementation of all the obfuscation techniques is tested for their correctness.

Generation of valid obfuscated jar file is the third module. Unit testing is performed in this module to test generation of valid jar file.

The Syntax and logical error are corrected automatically whenever they occur. All these syntax errors are rectified during compilation. The output is tested with the manual input. All the data are stored correctly.

6.2 INTEGRATION TESTING

Integration testing is the phase of software testing in which individual software modules are combined and tested as a group. It follows unit testing and precedes system testing.

Integration testing takes as its input modules that have been checked out by unit testing, groups them in larger aggregates, applies tests defined in an Integration test plan to those aggregates, and delivers as its output the integrated system ready for system testing.

Here, the jar file which is read and stored in the constant pool, is changed by the obfuscation techniques. This then integrated with the third module and tested to find whether the valid form of jar produced or not

6.3 WHITE BOX TESTING

White box testing, sometimes called glass-box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Using white box testing method, the software engineer can derive test cases.

The following techniques is performed in the project,

- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decisions on their true and false sides.
- Execute all loops at their boundaries and within their operational bounds.

6.4 BLACK BOX TESTING

Black box testing, also called behavioral testing, focuses on the functional requirements of the software. That is, black testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program. Black box testing is not alternative to white box techniques. Rather it is a complementary approach that is likely to uncover a different class of errors than white box methods.

Black box testing attempts to find errors in the following categories.

- Incorrect or missing functions.
- Interface errors.
- Errors in a data structures or external data base access.

- Behavior or performance errors.
- Initialization and termination errors.

The program is tested by giving valid jar file as input. This then produces a valid jar file. It is tested by giving an invalid jar file as input to produce an error message.

6.5 OUTPUT TESTING

The next step is output testing of the proposed system, since no system could be useful if it does not produce the required output in the specific format. The proposed system is tested for the required output in the proper format that is clear for the user. This testing is done in our project, by entering a valid jar file as input. The decompiler will produce the java code of the obfuscated class file, which is compared with original source file.

6.6 SYSTEM TESTING

System testing is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements. System testing takes, as its input, all of the integrated software components that have successfully passed Integration testing and also the software system itself.

System testing is more of a limiting type of testing, where it seeks to detect both defects within the integrated software components and also the system as a whole.

The project is tested as a whole with several inputs and is found to satisfy the specified requirements efficiently.

Testing is vital to the success of the system .The system is tested at the end of all development steps. This is a cyclic process and the software will continue to circulate through all steps till it attains the required quality.

CONCLUSION AND FUTURE WORK

7. CONCLUSION AND FUTURE WORK

CONCLUSION

A Good obfuscation tool should

- Preserve the semantics of the bytecode,
- Deter the cracker as long as possible,
- Be difficult to be overcome by a cracking tool, and
- Improve the run-time efficiency and reduce the bytecode size.

The techniques proposed in this document satisfy all the above requirements. Preserving the semantics of the bytecode is the most important criterion of obfuscation. Many techniques make the decompiled program uncompilable.

The obfuscation effects cannot be easily undone by other cracking tools. A cracker has to spend lots of time to debug the decompiled buggy program manually. The shorter names reduce the size of a bytecode file. Overloaded names also contribute to the compression of the bytecode and the jar file. Consequently, the storage space and the loading time are reduced.

The main objective of the obfuscation techniques proposed in this paper is to scramble the symbolic names and the symbolic references in the bytecode. Although the technique of identifier scrambling appeared several years ago and several commercial or free products are based on similar ideas, these techniques provide stronger protection for bytecode than other existing techniques.

SCOPE OF FUTURE ENHANCEMENTS

In this project, the data obfuscation techniques only have been used. We also have layout and control obfuscation which extends the security of the Java code .It is possible to extend the proposed obfuscation techniques to other languages. The prerequisite of the obfuscation techniques is that the information of the identifiers is stored in the bytecode and the decompilers rely on the information during decompilation.

For those languages that use a similar mechanism, i.e., symbolic linking, it is possible to apply the proposed techniques on them. Example is C# which is a .NET language.

APPENDIX

8. APPENDIX

8.1 SAMPLE SOURCE CODE

```
package proguard.gui;
import proguard.*;
import proguard.classfile.util.ClassUtil;
import proguard.gui.splash.*;
import proguard.util.ListUtil;

import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.URL;
import java.util.*;
import java.util.List;

public class OBFGUI extends JFrame
{
  private static final String NO_SPLASH_OPTION = "-nosplash";
  private static final String TITLE_IMAGE_FILE    = "vtitle.gif";
  private static final String BOILERPLATE_CONFIGURATION =
"boilerplate.pro";
  private static final String DEFAULT_CONFIGURATION          =
"default.pro";
```

```
  private static final String KEEP_ATTRIBUTE_DEFAULT =
"InnerClasses,                                    SourceFile,
LineNumberTable,Deprecated,Signature,*Annotation*,EnclosingMethod
";
  private static final String SOURCE_FILE_ATTRIBUTE_DEFAULT =
"SourceFile";
  private         static         final         Border         BORDER          =
BorderFactory.createEtchedBorder(EtchedBorder.RAISED);
  static boolean systemOutRedirected;
  private JFileChooser configurationChooser = new JFileChooser("");
  private JFileChooser fileChooser    = new JFileChooser("");
  private SplashPanel splashPanel;
  private ClassPathPanel programPanel = new ClassPathPanel(this, true);
  private ClassPathPanel libraryPanel = new ClassPathPanel(this, false);

  private ClassSpecification[] boilerplateKeep;
  private JCheckBox[]    boilerplateKeepCheckBoxes;
  private JTextField[]    boilerplateKeepTextFields;
  private    ClassSpecificationsPanel    additionalKeepPanel    =    new
ClassSpecificationsPanel(this, true);
  private ClassSpecification[] boilerplateKeepNames;
  private JCheckBox[]    boilerplateKeepNamesCheckBoxes;
  private JTextField[]    boilerplateKeepNamesTextFields;
  private  ClassSpecificationsPanel  additionalKeepNamesPanel  =  new
ClassSpecificationsPanel(this, true);
  private ClassSpecification[] boilerplateNoSideEffectMethods;
  private JCheckBox[]    boilerplateNoSideEffectMethodCheckBoxes;
```

```
  private ClassSpecificationsPanel additionalNoSideEffectsPanel = new
ClassSpecificationsPanel(this, false);
  private  ClassSpecificationsPanel  whyAreYouKeepingPanel  =  new
ClassSpecificationsPanel(this, false);
  private  JCheckBox  shrinkCheckBox                        =  new
JCheckBox(msg("shrink"));
  private        JCheckBox        printUsageCheckBox        =        new
JCheckBox(msg("printUsage"));
  private        JCheckBox        optimizeCheckBox        =        new
JCheckBox(msg("optimize"));
  private   JCheckBox   allowAccessModificationCheckBox   =   new
JCheckBox(msg("allowAccessModification"));
  private        JCheckBox        obfuscateCheckBox        =        new
JCheckBox(msg("obfuscate"));
  private        JCheckBox        printMappingCheckBox        =        new
JCheckBox(msg("printMapping"));
  private        JCheckBox        applyMappingCheckBox        =        new
JCheckBox(msg("applyMapping"));
  private     JCheckBox     obfuscationDictionaryCheckBox     =     new
JCheckBox(msg("obfuscationDictionary"));
  private     JCheckBox     overloadAggressivelyCheckBox     =     new
JCheckBox(msg("overloadAggressively"));
  private JCheckBox useUniqueClassMemberNamesCheckBox    = new
JCheckBox(msg("useUniqueClassMemberNames"));
  private  JCheckBox  defaultPackageCheckBox                  =  new
JCheckBox(msg("defaultPackage"));
```

```
  private JCheckBox useMixedCaseClassNamesCheckBox          = new
JCheckBox(msg("useMixedCaseClassNames"));
  private JCheckBox keepAttributesCheckBox                  = new
JCheckBox(msg("keepAttributes"));
  private JCheckBox newSourceFileAttributeCheckBox          = new
JCheckBox(msg("renameSourceFileAttribute"));

  private JCheckBox printSeedsCheckBox                      = new
JCheckBox(msg("printSeeds"));
  private JCheckBox verboseCheckBox                         = new
JCheckBox(msg("verbose"));
  private JCheckBox ignoreWarningsCheckBox                  = new
JCheckBox(msg("ignoreWarnings"));
  private JCheckBox warnCheckBox                            = new
JCheckBox(msg("warn"));
  private JCheckBox noteCheckBox                            = new
JCheckBox(msg("note"));
  private JCheckBox skipNonPublicLibraryClassesCheckBox     = new
JCheckBox(msg("skipNonPublicLibraryClasses"));
  private JCheckBox skipNonPublicLibraryClassMembersCheckBox =
new JCheckBox(msg("skipNonPublicLibraryClassMembers"));

  private JTextField printUsageTextField          = new JTextField(40);
  private JTextField printMappingTextField        = new JTextField(40);
  private JTextField applyMappingTextField        = new JTextField(40);
  private JTextField obfuscationDictionaryTextField         = new
JTextField(40);
```

```
  private JTextField defaultPackageTextField       = new JTextField(40);
  private JTextField keepAttributesTextField       = new JTextField(40);
  private JTextField newSourceFileAttributeTextField          = new
JTextField(40);
  private JTextField printSeedsTextField        = new JTextField(40);

  private      JTextArea      consoleTextArea            =       new
JTextArea(msg("processingInfo"), 3, 40);
  private JTextField reTraceMappingTextField                  = new
JTextField(40);
  private     JCheckBox     reTraceVerboseCheckBox         =     new
JCheckBox(msg("verbose"));
  private JTextArea stackTraceTextArea       = new JTextArea(3, 40);
  private      JTextArea      reTraceTextArea              =       new
JTextArea(msg("reTraceInfo"), 3, 40);


  /**
   * Creates a new OBFGUI.
   */
  public OBFGUI()
  {
    setTitle("OBF");
    setDefaultCloseOperation(EXIT_ON_CLOSE);

    // Create some constraints that can be reused.
    GridBagConstraints constraints = new GridBagConstraints();
```

```
    constraints.anchor = GridBagConstraints.WEST;
    constraints.insets = new Insets(0, 4, 0, 4);

    GridBagConstraints constraintsStretch = new GridBagConstraints();
    constraintsStretch.fill  = GridBagConstraints.HORIZONTAL;
    constraintsStretch.weightx = 1.0;
    constraintsStretch.anchor = GridBagConstraints.WEST;
    constraintsStretch.insets = constraints.insets;

    GridBagConstraints constraintsLast = new GridBagConstraints();
    constraintsLast.gridwidth = GridBagConstraints.REMAINDER;
    constraintsLast.anchor  = GridBagConstraints.WEST;
    constraintsLast.insets  = constraints.insets;

    GridBagConstraints         constraintsLastStretch        =        new
GridBagConstraints();
    constraintsLastStretch.gridwidth = GridBagConstraints.REMAINDER;
      constraintsLastStretch.fill    = GridBagConstraints.HORIZONTAL;
     constraintsLastStretch.weightx  = 1.0;
     constraintsLastStretch.anchor  = GridBagConstraints.WEST;
     constraintsLastStretch.insets  = constraints.insets;

      GridBagConstraints        splashPanelConstraints        =        new
GridBagConstraints();
      splashPanelConstraints.gridwidth                                =
GridBagConstraints.REMAINDER;
      splashPanelConstraints.fill     = GridBagConstraints.BOTH;
```

```
      splashPanelConstraints.weightx  = 1.0;
      splashPanelConstraints.weighty  = 0.02;
      splashPanelConstraints.anchor                                  =
GridBagConstraints.NORTHWEST;
      //splashPanelConstraints.insets   = constraints.insets;

      GridBagConstraints      welcomeTextAreaConstraints       =      new
GridBagConstraints();
      welcomeTextAreaConstraints.gridwidth                           =
GridBagConstraints.REMAINDER;
      welcomeTextAreaConstraints.fill     = GridBagConstraints.NONE;
      welcomeTextAreaConstraints.weightx  = 1.0;
      welcomeTextAreaConstraints.weighty  = 0.01;
      welcomeTextAreaConstraints.anchor                              =
GridBagConstraints.CENTER;//NORTHWEST;
      welcomeTextAreaConstraints.insets   = new Insets(20, 40, 20, 40);

      GridBagConstraints panelConstraints = new GridBagConstraints();
      panelConstraints.gridwidth = GridBagConstraints.REMAINDER;
      panelConstraints.fill       = GridBagConstraints.HORIZONTAL;
      panelConstraints.weightx   = 1.0;
      panelConstraints.anchor    = GridBagConstraints.NORTHWEST;
      panelConstraints.insets    = constraints.insets;

      GridBagConstraints      stretchPanelConstraints       =       new
GridBagConstraints();
```

```
    stretchPanelConstraints.gridwidth =
GridBagConstraints.REMAINDER;
    stretchPanelConstraints.fill    = GridBagConstraints.BOTH;
    stretchPanelConstraints.weightx  = 1.0;
    stretchPanelConstraints.weighty  = 1.0;
    stretchPanelConstraints.anchor =
GridBagConstraints.NORTHWEST;
    stretchPanelConstraints.insets   = constraints.insets;

    GridBagConstraints glueConstraints = new GridBagConstraints();
    glueConstraints.fill    = GridBagConstraints.BOTH;
    glueConstraints.weightx = 0.01;
    glueConstraints.weighty = 0.01;
    glueConstraints.anchor  = GridBagConstraints.NORTHWEST;
    glueConstraints.insets  = constraints.insets;

    GridBagConstraints bottomButtonConstraints = new
GridBagConstraints();
    bottomButtonConstraints.anchor =
GridBagConstraints.SOUTHEAST;
    bottomButtonConstraints.insets = new Insets(2, 2, 4, 6);
    bottomButtonConstraints.ipadx  = 10;
    bottomButtonConstraints.ipady  = 2;

    GridBagConstraints lastBottomButtonConstraints = new
GridBagConstraints();
```

```
    lastBottomButtonConstraints.gridwidth =
GridBagConstraints.REMAINDER;
    lastBottomButtonConstraints.anchor =
GridBagConstraints.SOUTHEAST;
    lastBottomButtonConstraints.insets =
bottomButtonConstraints.insets;
    lastBottomButtonConstraints.ipadx =
bottomButtonConstraints.ipadx;
    lastBottomButtonConstraints.ipady =
bottomButtonConstraints.ipady;

    // Leave room for a growBox on Mac OS X.
    if (System.getProperty("os.name").toLowerCase().startsWith("mac
os x"))
    {
      lastBottomButtonConstraints.insets = new Insets(2, 2, 4, 6 + 16);
    }

    GridBagLayout layout = new GridBagLayout();

    configurationChooser.addChoosableFileFilter(
      new ExtensionFileFilter(msg("proExtension"), new String[] {
".pro" }));

    // Create the opening panel.
    Font font = new Font("sansserif", Font.BOLD, 50);
    Color fontColor = Color.white;
```

```
    Sprite splash =
      new CompositeSprite(new Sprite[]
    {
      new TextSprite(new ConstantString("OBF"),
                new ConstantFont(new Font("sansserif", Font.BOLD,
90)),
                new ConstantColor(Color.gray),
                new ConstantInt(160),
                new LinearInt(-10, 120, new SmoothTiming(500,
1000))),

      new ShadowedSprite(new ConstantInt(3),
                  new ConstantInt(3),
                  new ConstantDouble(0.4),
                  new ConstantInt(2),
                  new CompositeSprite(new Sprite[]
      {
        new TextSprite(new ConstantString(msg("shrinking")),
                  new ConstantFont(font),
                  new ConstantColor(fontColor),
                  new LinearInt(1000, 60, new SmoothTiming(1000,
2000)),
                  new ConstantInt(70)),
        new TextSprite(new ConstantString(msg("optimization")),
                  new ConstantFont(font),
                  new ConstantColor(fontColor),
```

```
                  new LinearInt(1000, 400, new SmoothTiming(1500,
2500)),
                  new ConstantInt(60)),
          new TextSprite(new ConstantString(msg("obfuscation")),
                  new ConstantFont(font),
                  new ConstantColor(fontColor),
                  new LinearInt(1000, 350, new SmoothTiming(2000,
3000)),
                  new ConstantInt(140)),
          new TextSprite(new TypeWriterString(msg("developed"), new
LinearTiming(3000, 5000)),
                  new ConstantFont(new Font("monospaced",
Font.BOLD, 20)),
                  new ConstantColor(fontColor),
                  new ConstantInt(250),
                  new ConstantInt(170)),
      })),
    });
    splashPanel = new SplashPanel(splash, 0.5, 5000L);
    splashPanel.setPreferredSize(new Dimension(0, 200));

    JTextArea welcomeTextArea = new
JTextArea(msg("proGuardInfo"), 18, 50);
    welcomeTextArea.setOpaque(false);
    welcomeTextArea.setEditable(false);
    welcomeTextArea.setLineWrap(true);
    welcomeTextArea.setWrapStyleWord(true);
    welcomeTextArea.setPreferredSize(new Dimension(0, 0));
    welcomeTextArea.setBorder(new EmptyBorder(20, 20, 20, 20));
    addBorder(welcomeTextArea, "welcome");

    JPanel proGuardPanel = new JPanel(layout);
    proGuardPanel.add(splashPanel,    splashPanelConstraints);
    proGuardPanel.add(welcomeTextArea,
welcomeTextAreaConstraints);

    // Create the input panel.
    // TODO: properly clone the ClassPath objects.
    //    This    is    awkward    to    implement    in    the    generic
ListPanel.addElements(...)
    // method, since the Object.clone() method is not public.
    programPanel.addCopyToPanelButton(msg("moveToLibraries"),
libraryPanel);
    libraryPanel.addCopyToPanelButton(msg("moveToProgram"),
programPanel);

    // Collect all buttons of these panels and make sure they are equally
    // sized.
    List panelButtons = new ArrayList();
    panelButtons.addAll(programPanel.getButtons());
    panelButtons.addAll(libraryPanel.getButtons());
    setCommonPreferredSize(panelButtons);
    panelButtons = null;
```

```
    shrinkingOptionsPanel.add(printUsageBrowseButton,
constraintsLast);

    JPanel shrinkingPanel = new JPanel(layout);

    shrinkingPanel.add(shrinkingOptionsPanel, panelConstraints);
    addClassSpecifications(boilerplateKeep, shrinkingPanel,
boilerplateKeepCheckBoxes, boilerplateKeepTextFields);
    addBorder(additionalKeepPanel, "keepAdditional");
    shrinkingPanel.add(additionalKeepPanel, stretchPanelConstraints);

    // Create the boiler plate keep names panels.
    BoilerplateKeepNamesCheckBoxes              =              new
JCheckBox[boilerplateKeepNames.length];
    boilerplateKeepNamesTextFields              =              new
JTextField[boilerplateKeepNames.length];

    JButton             printMappingBrowseButton             =
createBrowseButton(printMappingTextField,
                                  msg("selectPrintMappingFile"));
    JButton             applyMappingBrowseButton             =
createBrowseButton(applyMappingTextField,
msg("selectApplyMappingFile"));
    JButton obfucationDictionaryBrowseButton =
createBrowseButton(obfuscationDictionaryTextField,
msg("selectObfuscationDictionaryFile"));
```

```
    obfuscationOptionsPanel.add(defaultPackageCheckBox,
constraints);
    obfuscationOptionsPanel.add(defaultPackageTextField,
constraintsLastStretch);
    obfuscationOptionsPanel.add(useMixedCaseClassNamesCheckBox,
constraintsLastStretch);
    obfuscationOptionsPanel.add(keepAttributesCheckBox,
constraints);
    obfuscationOptionsPanel.add(keepAttributesTextField,
constraintsLastStretch);
    obfuscationOptionsPanel.add(newSourceFileAttributeCheckBox,
constraints);
    obfuscationOptionsPanel.add(newSourceFileAttributeTextField,
constraintsLastStretch);

    JPanel obfuscationPanel = new JPanel(layout);
    obfuscationPanel.add(obfuscationOptionsPanel, panelConstraints);
    addClassSpecifications(boilerplateKeepNames, obfuscationPanel,
boilerplateKeepNamesCheckBoxes, boilerplateKeepNamesTextFields);

    addBorder(additionalKeepNamesPanel, "keepNamesAdditional");
    obfuscationPanel.add(additionalKeepNamesPanel,
stretchPanelConstraints);

    // Create the boiler plate "no side effect methods" panels.
    boilerplateNoSideEffectMethodCheckBoxes         =         new
JCheckBox[boilerplateNoSideEffectMethods.length];
```

```
    JPanel optimizationOptionsPanel = new JPanel(layout);
    addBorder(optimizationOptionsPanel, "options");

    optimizationOptionsPanel.add(optimizeCheckBox,
constraintsLastStretch);
    optimizationOptionsPanel.add(allowAccessModificationCheckBox,
constraintsLastStretch);

    JPanel optimizationPanel = new JPanel(layout);

    optimizationPanel.add(optimizationOptionsPanel, panelConstraints);
    addClassSpecifications(boilerplateNoSideEffectMethods,
optimizationPanel,
                  boilerplateNoSideEffectMethodCheckBoxes, null);

    addBorder(additionalNoSideEffectsPanel,
"assumeNoSideEffectsAdditional");
    optimizationPanel.add(additionalNoSideEffectsPanel,
stretchPanelConstraints);

    // Create the options panel.
    JButton             printSeedsBrowseButton             =
createBrowseButton(printSeedsTextField,
                                  msg("selectSeedsFile"));

    JPanel consistencyPanel = new JPanel(layout);
```

```
        addBorder(consistencyPanel, "consistencyAndCorrectness");

        consistencyPanel.add(printSeedsCheckBox, constraints);
        consistencyPanel.add(printSeedsTextField, constraintsStretch);
        consistencyPanel.add(printSeedsBrowseButton, constraintsLast);
        consistencyPanel.add(verboseCheckBox, constraintsLastStretch);
        consistencyPanel.add(noteCheckBox, constraintsLastStretch);
        consistencyPanel.add(warnCheckBox, constraintsLastStretch);
        consistencyPanel.add(ignoreWarningsCheckBox,
constraintsLastStretch);
        consistencyPanel.add(skipNonPublicLibraryClassesCheckBox,
constraintsLastStretch);

consistencyPanel.add(skipNonPublicLibraryClassMembersCheckBox,
constraintsLastStretch);

        // Collect all components that are followed by text fields and make
        // sure they are equally sized. That way the text fields start at the
        // same horizontal position.
        setCommonPreferredSize(Arrays.asList(new JComponent[] {
            printMappingCheckBox,
            applyMappingCheckBox,
            defaultPackageCheckBox,
            newSourceFileAttributeCheckBox,
        }));

        JPanel optionsPanel = new JPanel(layout);
```

```
        optionsPanel.add(consistencyPanel,  panelConstraints);
        addBorder(whyAreYouKeepingPanel, "whyAreYouKeeping");
        optionsPanel.add(whyAreYouKeepingPanel,
stretchPanelConstraints);

        // Create the process panel.
        consoleTextArea.setOpaque(false);
        consoleTextArea.setEditable(false);
        consoleTextArea.setLineWrap(false);
        consoleTextArea.setWrapStyleWord(false);
        JScrollPane            consoleScrollPane            =            new
JScrollPane(consoleTextArea);
        consoleScrollPane.setBorder(new EmptyBorder(1, 1, 1, 1));
        addBorder(consoleScrollPane, "processingConsole");
        JPanel processPanel = new JPanel(layout);
        processPanel.add(consoleScrollPane, stretchPanelConstraints);

        // Create the load, save, and process buttons.
        JButton loadButton = new JButton(msg("loadConfiguration"));
        loadButton.addActionListener(new
MyLoadConfigurationActionListener());
        JButton viewButton = new JButton(msg("viewConfiguration"));
        viewButton.addActionListener(new
MyViewConfigurationActionListener());
        JButton saveButton = new JButton(msg("saveConfiguration"));
        saveButton.addActionListener(new
MySaveConfigurationActionListener());
```

```
        JButton processButton = new JButton(msg("process"));
        processButton.addActionListener(new MyProcessActionListener());

        // Create the ReTrace panel.
        JPanel reTraceSettingsPanel = new JPanel(layout);
        addBorder(reTraceSettingsPanel, "reTraceSettings");
        Jbutton reTraceMappingBrowseButton =
createBrowseButton(reTraceMappingTextField,
msg("selectApplyMappingFile"));

        JLabel reTraceMappingLabel = new JLabel(msg("mappingFile"));

reTraceMappingLabel.setForeground(reTraceVerboseCheckBox.getFore
ground());

        reTraceSettingsPanel.add(reTraceMappingLabel,       constraints);
        reTraceSettingsPanel.add(reTraceMappingTextField,
constraintsStretch);
        reTraceSettingsPanel.add(reTraceMappingBrowseButton,
constraintsLast);
        reTraceSettingsPanel.add(reTraceVerboseCheckBox,
constraintsLastStretch);

        stackTraceTextArea.setOpaque(true);
        stackTraceTextArea.setEditable(true);
        stackTraceTextArea.setLineWrap(false);
        stackTraceTextArea.setWrapStyleWord(true);
```

```
        JScrollPane            stackTraceScrollPane            =            new
JScrollPane(stackTraceTextArea);
        addBorder(stackTraceScrollPane, "obfuscatedStackTrace");

        reTraceTextArea.setOpaque(false);
        reTraceTextArea.setEditable(false);
        reTraceTextArea.setLineWrap(true);
        reTraceTextArea.setWrapStyleWord(true);
        JScrollPane            reTraceScrollPane            =            new
JScrollPane(reTraceTextArea);
        reTraceScrollPane.setBorder(new EmptyBorder(1, 1, 1, 1));
        addBorder(reTraceScrollPane, "deobfuscatedStackTrace");

        JPanel reTracePanel = new JPanel(layout);
        reTracePanel.add(reTraceSettingsPanel, panelConstraints);
        reTracePanel.add(stackTraceScrollPane, panelConstraints);
        reTracePanel.add(reTraceScrollPane,    stretchPanelConstraints);

        // Create the load button.
        JButton            loadStackTraceButton            =            new
JButton(msg("loadStackTrace"));
        loadStackTraceButton.addActionListener(new
MyLoadStackTraceActionListener());

        JButton reTraceButton = new JButton(msg("reTrace"));
        reTraceButton.addActionListener(new
MyReTraceActionListener());
```

```
        // Create the main tabbed pane.
        TabbedPane tabs = new TabbedPane();
        tabs.add(msg("proGuardTab"),     proGuardPanel);
        tabs.add(msg("inputOutputTab"),  inputOutputPanel);
        tabs.add(msg("shrinkingTab"),    shrinkingPanel);
        tabs.add(msg("obfuscationTab"),  obfuscationPanel);
        tabs.add(msg("optimizationTab"), optimizationPanel);
        tabs.add(msg("informationTab"),  optionsPanel);
        tabs.add(msg("processTab"),      processPanel);
        tabs.add(msg("reTraceTab"),      reTracePanel);

tabs.addImage(Toolkit.getDefaultToolkit().getImage(this.getClass().getR
esource(TITLE_IMAGE_FILE)));

        // Add the bottom buttons to each panel.
        proGuardPanel.add(Box.createGlue(), glueConstraints);
        proGuardPanel.add(loadButton, bottomButtonConstraints);
        proGuardPanel.add(createNextButton(tabs),
lastBottomButtonConstraints);

        inputOutputPanel.add(Box.createGlue(), glueConstraints);
        inputOutputPanel.add(createPreviousButton(tabs),
bottomButtonConstraints);
        inputOutputPanel.add(createNextButton(tabs),
lastBottomButtonConstraints);

        shrinkingPanel .add(Box.createGlue(),       glueConstraints);
```

```
        shrinkingPanel                  .add(createPreviousButton(tabs),
bottomButtonConstraints);
        shrinkingPanel                  .add(createNextButton(tabs),
lastBottomButtonConstraints);

        obfuscationPanel.add(Box.createGlue(),       glueConstraints);
        obfuscationPanel.add(createPreviousButton(tabs),
bottomButtonConstraints);
        obfuscationPanel.add(createNextButton(tabs),
lastBottomButtonConstraints);

        optimizationPanel.add(Box.createGlue(),       glueConstraints);
        optimizationPanel.add(createPreviousButton(tabs),
bottomButtonConstraints);
        optimizationPanel.add(createNextButton(tabs),
lastBottomButtonConstraints);

        optionsPanel.add(Box.createGlue(), glueConstraints);
        optionsPanel.add(createPreviousButton(tabs),
bottomButtonConstraints);
        optionsPanel.add(createNextButton(tabs),
lastBottomButtonConstraints);

        processPanel.add(Box.createGlue(),glueConstraints);
        processPanel.add(createPreviousButton(tabs),
bottomButtonConstraints);
        processPanel.add(viewButton, bottomButtonConstraints);
```

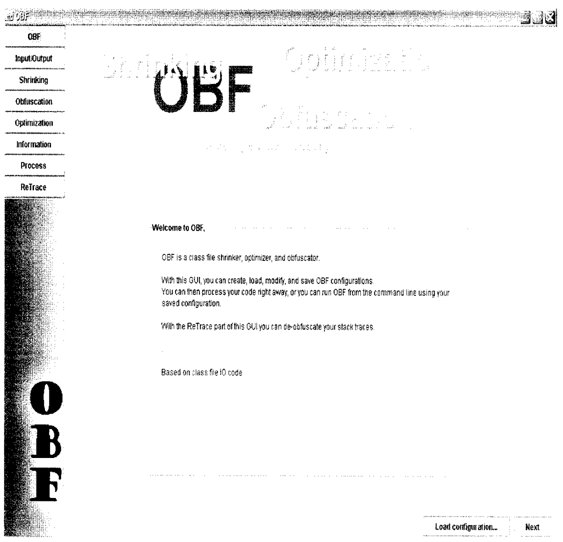
```
        processPanel.add(saveButton, bottomButtonConstraints);
        processPanel.add(processButton, lastBottomButtonConstraints);

        reTracePanel.add(Box.createGlue(),glueConstraints);
        reTracePanel.add(loadStackTraceButton, bottomButtonConstraints);
        reTracePanel.add(reTraceButton, lastBottomButtonConstraints);

        // Initialize the GUI settings to reasonable defaults.

loadConfiguration(this.getClass().getResource(DEFAULT_CONFIGUR
ATION));

        // Add the main tabs to the frame and pack it.
        getContentPane().add(tabs);
    }


    public void startSplash()
    {
        splashPanel.start();
    }
  public static void main(String[] args)
    {
        OBFGUI gui = new OBFGUI();
        gui.pack();
```

```
        Dimension                   screenSize                         =
Toolkit.getDefaultToolkit().getScreenSize();
        Dimension guiSize    = gui.getSize();
        gui.setLocation((screenSize.width - guiSize.width)  / 2,
                   (screenSize.height - guiSize.height) / 2);
        gui.show();

        // Start the splash animation, unless specified otherwise.
        int argIndex = 0;
        if (argIndex < args.length &&
            NO_SPLASH_OPTION.startsWith(args[argIndex]))
        {
            gui.skipSplash();
            argIndex++;
        }
        else
        {
            gui.startSplash();
        }

        // Load an initial configuration, if specified.
        if (argIndex < args.length)
        {
            gui.loadConfiguration(new File(args[argIndex]));
            argIndex++;
        }
        if (argIndex < args.length)
```

```
    {
        System.out.println(gui.getClass().getName() + ": ignoring extra
arguments [" + args[argIndex] + "...]");
    }
  }
}
```

8.2 SCREEN SHOTS

8.2.1 FRONT PAGE



Welcome to OBF,

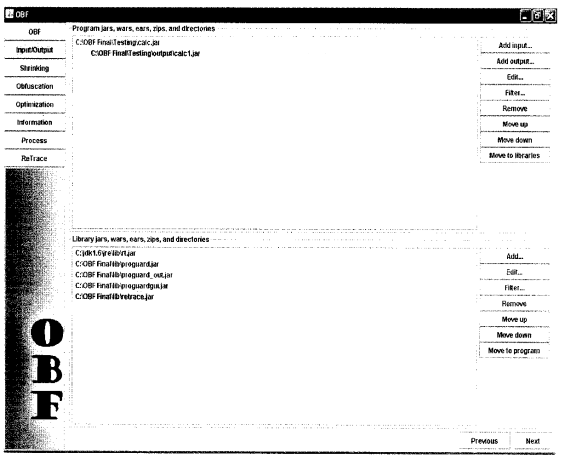
OBF is a class file shrinker, optimizer, and obfuscator.

With this GUI, you can create, load, modify, and save OBF configurations.
You can then process your code right away, or you can run OBF from the command line using your saved configuration.

With the ReTrace part of this GUI you can de-obfuscate your stack traces.

Based on class file IO code

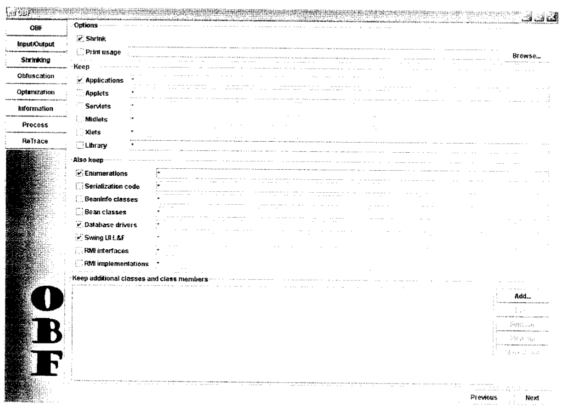
8.2.2 ADDING INPUT AND OUTPUT JAR FILES



Program jars, wars, ears, zips, and directories
C:\OBF Final\Testing\calc.jar
C:\OBF Final\Testing\output\calc1.jar

Library jars, wars, ears, zips, and directories
C:\jdk1.6\jre\lib\rt.jar
C:\OBF Final\lib\proguard.jar
C:\OBF Final\lib\proguard_out.jar
C:\OBF Final\lib\proguardgui.jar
C:\OBF Final\lib\retrace.jar

8.2.3 SHRINKING



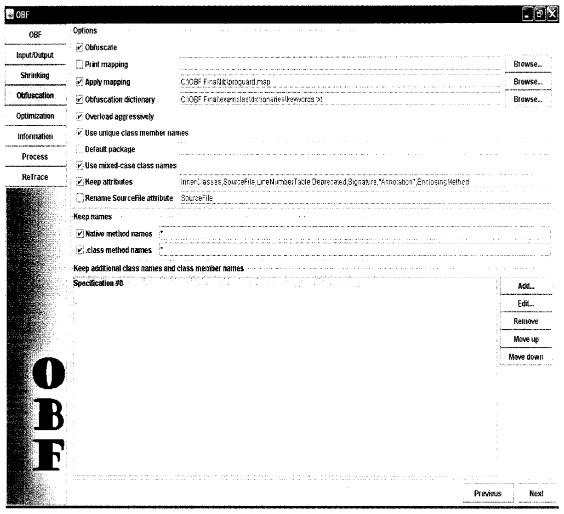
Options
- [✓] Shrink
- [ ] Print usage

Keep
- [✓] Applications *
- [ ] Applets *
- [ ] Servlets *
- [ ] Midlets *
- [ ] Xlets *
- [ ] Library *

Also keep
- [✓] Enumerations *
- [ ] Serialization code *
- [ ] BeanInfo classes *
- [ ] Bean classes *
- [✓] Database drivers *
- [✓] Swing UI L&F *
- [ ] RMI interfaces *
- [ ] RMI implementations *

Keep additional classes and class members

8.2.4 OBFUSCATION



Options

[✓] Obfuscate
[ ] Print mapping
[✓] Apply mapping C:\OBF Final\lib\proguard.map
[✓] Obfuscation dictionary C:\OBF Final\examples\dictionaries\keywords.txt
[✓] Overload aggressively
[✓] Use unique class member names
[ ] Default package
[✓] Use mixed-case class names
[✓] Keep attributes InnerClasses,SourceFile,LineNumberTable,Deprecated,Signature,*Annotation*,EnclosingMethod
[ ] Rename SourceFile attribute SourceFile

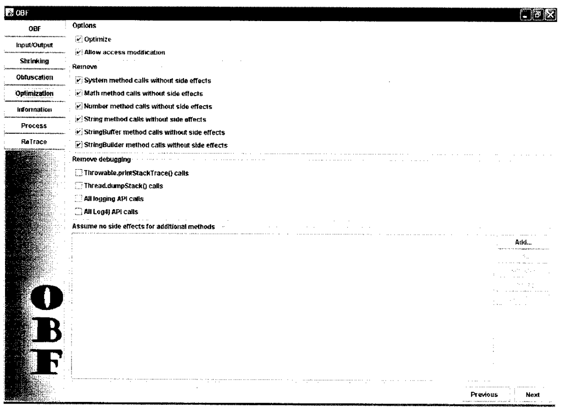
Keep names

[✓] Native method names *
[✓] .class method names *

Keep additional class names and class member names

Specification #0

8.2.5 OPTIMIZATION



Options

[✓] Optimize
[✓] Allow access modification

Remove

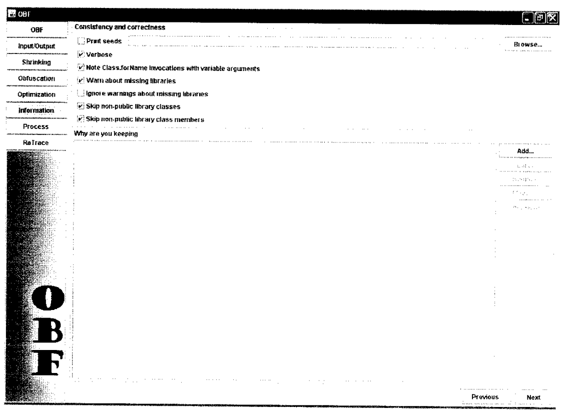
[✓] System method calls without side effects
[✓] Math method calls without side effects
[✓] Number method calls without side effects
[✓] String method calls without side effects
[✓] StringBuffer method calls without side effects
[✓] StringBuilder method calls without side effects

Remove debugging

[ ] Throwable.printStackTrace() calls
[ ] Thread.dumpStack() calls
[ ] All logging API calls
[ ] All Log4j API calls

Assume no side effects for additional methods

8.2.6 INFORMATION REQUIRED IN THE OUTPUT

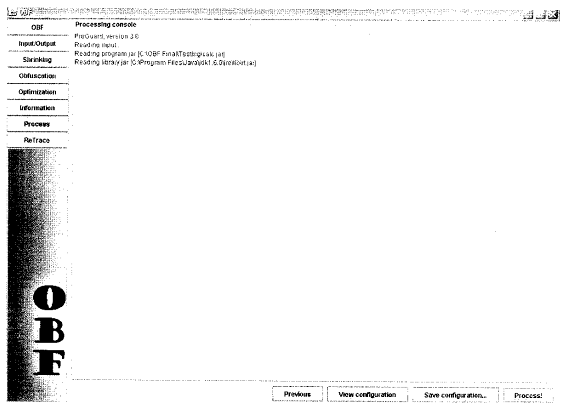


Consistency and correctness

[ ] Print seeds
[✓] Verbose
[✓] Note Class.forName invocations with variable arguments
[✓] Warn about missing libraries
[ ] Ignore warnings about missing libraries
[✓] Skip non-public library classes
[✓] Skip non-public library class members

Why are you keeping

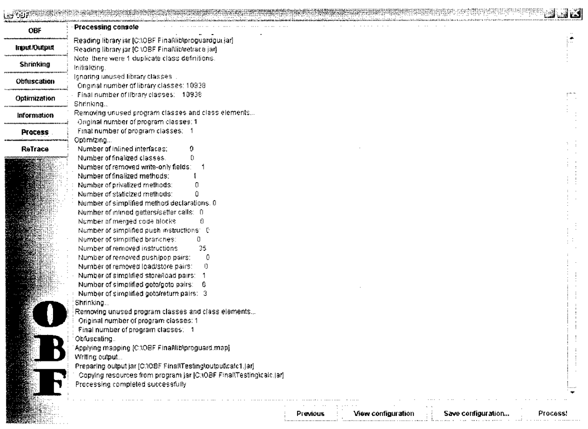
8.2.7 PROCESS



Processing console

```
ProGuard, version 3.6
Reading input...
Reading program jar [C:\OBF Final\Testing\calc.jar]
Reading library jar [C:\Program Files\Java\jdk1.6.0\jre\lib\rt.jar]
```

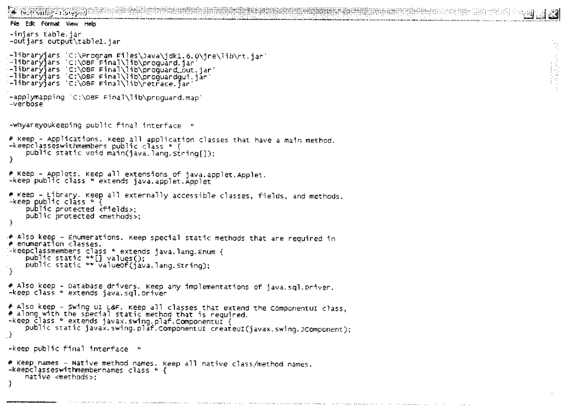
8.2.8 OUTPUT SCREEN



Processing console

```
Reading library jar [C:\OBF Final\lib\proguardgui.jar]
Reading library jar [C:\OBF Final\lib\retrace.jar]
Note: there were 1 duplicate class definitions.
Initializing.
Ignoring unused library classes...
  Original number of library classes: 10938
  Final number of library classes:    10938
Shrinking...
Removing unused program classes and class elements...
  Original number of program classes: 1
  Final number of program classes:    1
Optimizing...
  Number of inlined interfaces:          0
  Number of finalized classes:           0
  Number of removed write-only fields:   1
  Number of finalized methods:           1
  Number of privatized methods:          0
  Number of staticized methods:          0
  Number of simplified method declarations: 0
  Number of inlined getters/setter calls:  0
  Number of merged code blocks:          0
  Number of simplified push instructions:  0
  Number of simplified branches:         0
  Number of removed instructions:        35
  Number of removed push/pop pairs:      0
  Number of removed load/store pairs:    0
  Number of simplified store/load pairs: 1
  Number of simplified goto/goto pairs:  0
  Number of simplified goto/return pairs: 3
Shrinking...
Removing unused program classes and class elements...
  Original number of program classes: 1
  Final number of program classes:    1
Obfuscating...
Applying mapping [C:\OBF Final\lib\proguard.map]
Writing output...
Preparing output jar [C:\OBF Final\Testing\output\calc1.jar]
  Copying resources from program jar [C:\OBF Final\Testing\calc.jar]
Processing completed successfully
```

8.2.9 NOTEPAD



```
-injars table.jar
-outjars output\table1.jar

-libraryjars 'C:\Program Files\Java\jdk1.6.0\jre\lib\rt.jar'
-libraryjars 'C:\OBF Final\lib\proguard.jar'
-libraryjars 'C:\OBF Final\lib\proguard_out.jar'
-libraryjars 'C:\OBF Final\lib\proguardgui.jar'
-libraryjars 'C:\OBF Final\lib\retrace.jar'

-applymapping 'C:\OBF Final\lib\proguard.map'
-verbose


-whyareyoukeeping public final interface  *

# Keep - Applications. Keep all application classes that have a main method.
-keepclasseswithmembers public class * {
    public static void main(java.lang.String[]);
}

# Keep - Applets. Keep all extensions of java.applet.Applet.
-keep public class * extends java.applet.Applet

# Keep - Library. Keep all externally accessible classes, fields, and methods.
-keep public class * {
    public protected <fields>;
    public protected <methods>;
}

# Also keep - Enumerations. Keep special static methods that are required in
# enumeration classes.
-keepclassmembers class * extends java.lang.Enum {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}

# Also keep - Database drivers. Keep any implementations of java.sql.Driver.
-keep class * extends java.sql.Driver

# Also keep - Swing UI L&F. Keep all classes that extend the ComponentUI class,
# along with the special static method that is required.
-keep class * extends javax.swing.plaf.ComponentUI {
    public static javax.swing.plaf.ComponentUI createUI(javax.swing.JComponent);
}

-keep public final interface  *

# Keep names - Native method names. Keep all native class/method names.
-keepclasseswithmembernames class * {
    native <methods>;
}
```

REFERENCES

REFERENCES

1. Deitel and Deitel (1999), "JAVA How To Program", Pearson Education Inc.
2. Pautric Naughton, Herbert Schildt (1999), "The Complete Reference JAVA2 Third Edition", Tata McGraw-Hill Publishing Company Limited.
3. Java obfuscator, " www.preemptive.com "
4. Class file format, www.resources.ej-technologies.com/jclasslib
5. Class file reader, " www.kimbly.com/code/classfile "
6. Protect java application against code theft, www.excelsior-usa.com/jetprotection.html
7. Obfuscation, http://mindprod.com/jgloss
8. www.hotjava.com