Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 71 (2004) 1–10
The Journal of Systems and Software
www.elsevier.com/locate/jss

Advanced obfuscation techniques for Java bytecode

Jien-Tsai Chan *, Wuu Yang

Department of Computer and Information Science, National Chiao Tung University, Hsinchu, Taiwan 300, ROC

Received 17 May 2002; received in revised form 30 July 2002; accepted 2 August 2002

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 deter a determined cracker very long. We propose 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 our 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, we also propose several techniques to intentionally introduce syntactic and semantic errors into the decompiled program while preserving the original behaviors of the bytecode. Thus, the decompiled program would have to be debugged manually. Although our basic approach is to scramble the identifiers in Java bytecode, the scrambled bytecode produced with our 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.
© 2002 Elsevier Inc. All rights reserved.

Keywords: Program protection; Bytecode obfuscation; Java programming language

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).

The Java programming language has become more and more popular since its first release in 1994 (Gosling et al., 2000). One of 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 (Engel, 1999; Lindholm and Yellin, 1999; Meyer and Downing, 1997; Venners, 1998). These names and the simple stack-machine instructions facilitate the decompilation of the bytecode file.

There are many free or commercial Java decompilers (D & C, 2001; Hoeniche, 2001; Kouznetsov, 2001; Kumar, 2001; Mayon, 2001; PsychoticSoftware, 2001; Vliet, 1996). The decompiled program is almost identical to the original source program. These decompilers become the lethal weapon of intellectual property piracy.

* Corresponding author. Tel.: +886-9-3330-3945; fax: +886-3-572-1490.
E-mail addresses: maxchan@cis.nctu.edu.tw (J.-T. Chan), wuuyang@cis.nctu.edu.tw (W. Yang).

0164-1212/$ - see front matter © 2002 Elsevier Inc. All rights reserved.
doi:10.1016/S0164-1212(02)00066-3

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, 2001; Eastridge, 2000; Hoeniche, 2001; Plumb, 2001; Retrologic, 2000). Usually, a meaningful name is substituted by a meaningless name.

In this paper, we propose a new obfuscation approach that 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 re-compilable. 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. Obfuscation scope

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.

3. The candidates for identifier scrambling

According to the Java specification (Gosling et al., 2000), 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 (see Section 3.6 of Lindholm and Yellin (1999) and Section 3.7 of Engel (1999)). 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 LocalVariableTable 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: The instance method that serves as a callback function.

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.

4. Basic approach

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 subpackage 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 superclass. 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 does 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 extends A implements t.J {}

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

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

Fig. 1. All the types of an application.

Fig. 1 shows an example in which an application consists of several packages and types. 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. 2 and 3, respectively. The shadowed area (surround by a dotted curve) denotes the obfuscation scope.

The basic obfuscation approach consists of the following five steps:

(1) Analyze all the bytecode files that are in the obfuscation scope and construct the package structure and the inheritance structure.
(2) Traverse the package structure from root to leaf. During the traversal, use sequentially generated names to substitute the package names and top-level type names. The generation of names is restarted for each package node. For example, suppose that the generating sequence of names is a, b, c... After this step, the package structure in Fig. 2 will become the one in Fig. 4 and the inheritance structure in Fig. 3 will become the one in Fig. 5.

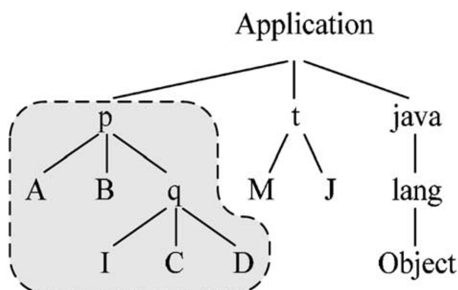

Fig. 2. Package structure.

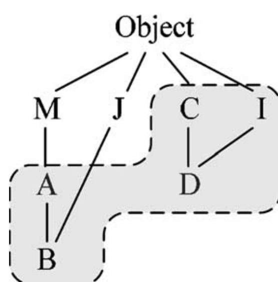

Fig. 3. Inheritance structure.

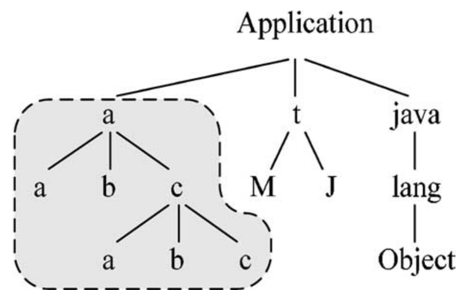

Fig. 4. Obfuscated package structure.

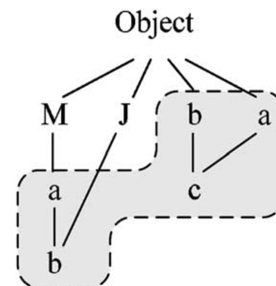

Fig. 5. Obfuscated inheritance structure.

(3) Traverse the inheritance structure from root to leaf. During the traversal, perform the following steps for each type `T`.
  (3a) Restart the generation of names. Replace the names of all the fields with the sequentially generated new names.
  (3b) 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 (Gosling et al., 2000), 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.
  (3c) 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 supertype `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 3a to 3c for each nested type.

(4) Update all the symbolic references in the obfuscation scope with the substituted names.
(5) Save the obfuscated code to each bytecode file in the obfuscation scope.

Notice that in Java source programs, a field may be *shadowed* (see Section 6.3.1 of (Gosling et al., 2000)) 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 3a and 3b are simplified because we do not have to worry about the obscurity problem that has been resolved by the compiler.

4.1. 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 bytecode. 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.

4.2. Dynamic loading problem

Java supports dynamical loading and reflection. Through the method `Class.forName ("MyClassName")`, 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. Therefore, 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.

4.3. Overloading unrelated methods

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 superclasses 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 superclasses and the subclass need not be preserved. Furthermore, we can make use of the relationships between static methods and instance methods of the superclasses and the subclass to provide another layer of protection. See Section 5.4 for details.

Second, widening conversions (i.e. coercion) can result in surprising benefits. In Java, there are two kinds of widening conversions. A *widening primitive conversion* is a conversion from a primitive type to another primitive type. During a widening primitive conversion, no information about the overall precision of the numeric value is lost. A *widening reference conversion* is a conversion between non-primitive types. A widening reference conversion (i.e. from a subclass to its superclass) must be proved correct at compile time. Method invocation allows both widening conversions. The widening conversions are performed automatically and implicitly by a compiler. Consider the following example.

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

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 Jad decompiler (Kouznetsov, 2001)) becomes

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

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 Jode (Hoeniche, 2001) and JReverse Pro (Kumar, 2001), add an explicit casting conversion to each parameter when necessary. The semantic change will not occur due to the explicit casting conversion.

(a)
```
class A {
  int x = 1;
}
class B extends A {
  int y = 2;
}
class C extends B {
  int t = x;
  int u = y;
}
```

(b)
```
class a {
  int a = 1;
}
class b extends a {
  int a = 2;
}
class c extends b {
  int a = a;
  int b = a;
}
```

Fig. 6. (a) An example program and (b) the decompiled result by the JAD decompiler.

4.4. An interesting example

The example in Fig. 6a is a common situation in many applications. Class `C` contains two fields `t` and `u` and inherits the fields `x` and `y` from the superclasses `A` and `B`, respectively. The values of `x` and `y` are accessed in class `C`. After applying the basic obfuscation approach discussed in the previous section on the bytecode of the example, the obfuscated bytecode is then decompiled with the Jad decompiler (Kouznetsov, 2001). The result is shown in Fig. 6b. The obfuscated bytecode still functions correctly. However, the decompiled result cannot be compiled successfully again.

This example shows the possibility of introducing changes to the bytecode so that the decompiled program of the obfuscated bytecode cannot be compiled successfully again. The detail will be discussed in the next section.

5. Making the decompiled program uncompilable

The techniques discussed in this section modify the bytecode so that the decompiled program of the obfuscated bytecode contains obscure compilation errors while the obfuscated bytecode still functions correctly. Therefore, a cracker has to debug the decompiled program manually and, hopefully, painfully.

In a Java program, an identifier may denote a type, a field, a method, a parameter, or a local variable. The Java compiler could be confused about which entity is intended when an identifier denotes more than one entity. Therefore, several rules are defined in the Java language specification (Gosling et al., 2000) to clarify the confusion. A Java compiler should obey these rules strictly during compilation. However, once the bytecode is produced, the Java virtual machine will ignore these rules. This gap between a Java compiler and a Java virtual machine offers opportunities to intentionally violate these rules, not in the Java source program, but in the bytecode.

Therefore, the crux of the techniques presented in this section is to intentionally violate some of these rules in

the bytecode in order to prevent the bytecode from being decompiled. Note that these obfuscations cannot be easily undone by automatic tools.

The rules and the techniques of violating them are discussed in the following subsections. Several techniques may be applied together to enforce even better protection. We will also discuss the limitations of each technique.

5.1. Illegal identifiers

The Java language specification (Gosling et al., 2000) states that an identifier should be a letter followed by letters or digits. An identifier cannot be identical to a keyword, a boolean literal or the null literal. These rules help the lexical analyzer and the parser to analyze a program. However, these rules need not be obeyed in the bytecode. Note that when JVM loads the bytecode, JVM does not verify whether the names in the constant pool comply with the definition of an identifier. Therefore, 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 illegal identifiers will result in compilation errors.

For example, we can rename an identifier in the bytecode as the boolean literal "`false`" or the symbol "`<>?!#`". The modified bytecode still runs as before. However, decompilers will face troubles for this change. We tested this technique with several available decompilers. Many decompilers use Jad as the decompilation engine and add their own user interfaces. Only the decompilers based on different decompilation engines are chosen for the experiment (D & C, 2001; Hoeniche, 2001; Kouznetsov, 2001; Kumar, 2001; Mayon, 2001; PsychoticSoftware, 2001; Vliet, 1996). The result is shown in Table 1. Jad and jAscii are smarter than others when handling keywords as identifiers. They change the keyword identifier to an ordinary identifier automatically. Other decompilers use the keyword as the name of an identifier directly in the decompiled program. On the other hand, all the tested decompilers are fooled by the illegal symbol. JReverse Pro and Jode even fail to decompile the modified bytecode.

Table 1
The decompiled results of using illegal characters in the name of an identifier

Decompiler	Use keyword as identifiers	Use illegal characters in identifiers
Jad	"_fldfalse"	"<_3E_3F_21_23_"
jAscii	"_fldfalse"	"< >?!#"
Mocha	"false"	"< >?!#"
deClassify	"false"	"< >?!#"
JReverse Pro	"false"	ERROR
ClassSpy	"false"	"< >?!#"
Jode	"false"	ERROR

Table 2
The symbols and characters that are unfit as or in an identifier in JVM

Symbol and character	Note
<init>	The constructor name of a class
<clinit>	The static initializer of a class
/	Path separator in UNIX system
\	Path separator in Windows system
:	Path separator in Mac system or drive character in Windows system
$	Nested type separator
.	The separator in fully qualified names should not be used in the name of a type

Note that not all the symbols and characters can be used as or in an identifier in the constant pool. Some symbols and characters have special meanings to the JVM and to the host system. The symbols and characters that are unfit as or in an identifier are listed in Table 2.

All the constructors of a type are named "`<init>`" in the bytecode. Therefore, the name "`<init>`" should be avoided in obfuscation. Otherwise, JVM might be confused when calling a constructor. The symbol "`<clinit>`" is the name of the static initializer of a type. JVM will invoke it to initialize the static members of a type.

The characters "/", "\", and ":" should not to be used in the name of a type. The three characters are used as path separators in the host file systems on different platforms. Currently, most implementations of the Java runtime system use the file system of the host to store the bytecode files. The only exception we know is IBM's VisualAge for Java (IBM, 2001), which uses a database system (called ENVY) to manage the bytecode files. If the three characters are used in the names of types and the types are stored in the host file system, these separators will cause the JVM to misinterpret the types.

The character "$" is used as the separator of a type and its nested types. Arbitrarily using "$" in the name of an identifier may cause some unexpected results. However, it can be used cleverly to introduce another kind of protection. See Section 5.3 for details.

5.2. Some interesting examples

We could use a few characters that have specific meanings in a Java source program to rename the identifiers in the bytecode. The characters include "`. `", "`(`", "`)`", "`; `", and the space character. The following code is the original code of the example in this subsection.

```
class A {
  int foo = 1;
}
```

The name `foo` in the bytecode is changed to a name made up of the above characters. The decompilation results by different decompilers are shown in Table 3.

Table 3
The decompiled results of some interesting variable names

Decompiler	foo → a.b	foo → 1.2	foo → a()	foo → □□□ (3 spaces)	foo → a;b
JAD	int a.b; b = 1;	int_cls1_fld2; _fld2 = 1;	int f_28_29_ = 1;	int _20_20_20_ = 1;	int a_3B_b = 1;
jAscii	int a.b = 1;	int_fld1.2 = 1;	int a() = 1;	int □□□ = 1;	int a;b = 1;
Mocha	int a.b = 1;	int 1.2 = 1;	int a() = 1;	int □□□ = 1;	int a;b = 1;
ClassCracker	int a.b = 1;	int 1.2 = 1;	int a() = 1;	int □□□ = 1;	int a;b = 1;
JReverse Pro	int a.b = 1;	int 1.2 = 1;	int a() = 1;	int □□□ = 1;	int a;b = 1;
ClassSpy	int a.b = 1;	int spy_1.2 = 1;	int a() = 1;	int □□□ = 1;	int a;b = 1;
Jode	int a.b = 1;	int 1.2 = 1;	int a() = 1;	int □□□ = 1;	int a;b = 1;

The character "." is the separator in a fully qualified name and the decimal point in a floating-point number. It is also the separator between a reference and its members. After we change `foo` to "a.b", the Java compiler will consider "a" as an object or a type and "b" as a member of "a". This name results in a compilation error. All the decompilers are fooled by this renaming. The Jad decompiler even uses a wrong variable name in the constructor.

Changing `foo` to a floating-point number "1.2" also fools all the tested decompilers. This time, Jad, jAscii and ClassSpy try to correct the illegal identifier. But all fail.

However, the character "." should not be used in the name of a type in the bytecode. The substring before "." will be treated as the name of a package or a type by the JVM. Consequently, a runtime error happens.

The characters "(" and ")" are used as a pair and are appended to the end of the name of a field. In this case, the field name will be treated as a method name by the Java compiler. Jad can correct this illegal name automatically while other decompilers cannot.

Notice that the two characters "(" and ")" can also be used in the name of a method. In the bytecode, the return type and the types of the parameters of a method are encoded as a string, called a *descriptor*, which is separated from the method name. Therefore, using "(" and ")" in the name of a method does not affect the JVM to determine the signature of the method.

The space character and the tab character are the separators of tokens in a source program. After `foo` is changed to be three spaces, the variable becomes invisible! Among the tested decompilers, only Jad can correct this illegal name; other decompilers cannot.

A semicolon ";" is the end mark of a statement or a declaration in a Java program. In the decompiled program, the name "a;b" is divided into two names. Consequently, a runtime error happens. Among the tested decompilers, only Jad can correct the illegal name; other decompilers cannot.

5.3. Nested type names

According to the Java language specification (Gosling et al., 2000), a nested type cannot have the same name as any of its enclosing types. If a nested type could have the same name as one of its enclosing types, the Java compiler would be confused in certain situations. For example, consider the following program.

```
class M {
  class M {}
  void f() {
    M n;          // which M ??
    n = new M(); // which M() ??
  }
}
```

The above example cannot pass the compilation because the Java compiler cannot determine which `M` is intended in the declaration of the local variable `n`.

After compilation, the name of a nested type `N` enclosed in type `M` becomes `M$N`. The Java compiler uses "$" as the separator between an enclosing type and a nested type while using "." as the separator among a package, its subpackages, and its top-level types. Suppose that `M` is a top-level type in the package `p.q`. The fully qualified name of `N` is `p.q.M$N`. Furthermore, `N` is compiled to an independent bytecode file named "`M$N.class`".

In the bytecode, the simple name of a nested type can be changed to be the same as that of its enclosing type. For example, a nested type named `M$N` can be renamed as `M$M`.

After a nested type is renamed, the bytecode and the corresponding symbolic references also have to be modified accordingly. Otherwise, JVM cannot find the bytecode file when trying to load the nested type. The decompiled program of the obfuscated bytecode cannot be successfully compiled again.

5.4. Static methods vs. instance methods

According to the specification of the Java language (Gosling et al., 2000), an inherited static method cannot be overridden by an instance method with the same signature in a subclass. Similarly, an inherited instance method cannot be overridden by a static method with the same signature in a subclass.

There are two processes to intentionally violate the above rule. They both require that the superclass and the subclass in which the methods exist are in the obfuscation scope. The first process is that we can add a bogus static method in the superclass (or subclass) for an instance method and add a bogus instance method in the superclass (or subclass) for a static method. Suppose that there is an instance method `m` in class `Y` and that `Y` inherits class `X`. We can make a bogus static method `m′` in `X`. Method `m′` has the same signature as method `m`. To make the bogus one looks like a real method, the body of `m′` could be identical to that of `m`. To make things even more complicated, we can make `m′` to be slightly different from `m`. It would be difficult to determine which method is the correct one when semantic change are introduced into the bogus version.

The second process is to override inherited methods that have different names but have the same number and types of parameters. Consider the following example.

```
class X {
  static int m(int a, String b)
    throws EOFException {...}
}
class Y extends X {
  boolean n(int c, String d)
    throws FileNotFoundException {...}
}
```

The names `m` and `n` can be changed to be the same, such as `p`. The new program becomes

```
class X {
  static int p(int a, String b)
    throws EOFException {...}
}
class Y extends X {
  boolean p(int c, String d)
    throws FileNotFoundException {...}
}
```

The new program violates the compiler rule in the Java specification (Gosling et al., 2000) which says that an inherited instance method cannot be overridden by a static method.

Note that the return type and the *throws* clause of a method is not part of the signature. If both `m` and `n` were instance methods, renaming them to `p` makes `n` override `m`. This overriding relationship violates yet another rule in the Java specification (Gosling et al., 2000) which says that an instance method and the overridden inherited instance method must have the same return types and *compatible* `throws` clauses. This technique provides a layer of protection. But overriding may change the method that is invoked when JVM dynamically dispatches an instance method. Therefore, arbitrarily overriding instance methods is not allowed in our obfuscation method. This technique requires that one of the two methods is a static method and the other is an instance method.

JVM uses four instructions—*invokestatic*, *invokeinterface*, *invokevirtual*, *invokespecial*—for invoking methods. A static method is invoked with the instruction *invokestatic*. An instance method is invoked with the instruction *invokevirtual* (or *invokeinterface*) if the declared type of the reference is a class (or an interface, respectively). An instance method can also be invoked with the instruction *invokespecial*, which invokes an instance method of a superclass, a private method, or the instance initialization method. The differences between the four instructions lie in the lookup procedure for resolving the method to be invoked. Because static methods and instance methods are invoked with different instructions, renaming `m` and `n` to be `p` in the bytecode in the above example will not change the behavior of the obfuscated bytecode in the above example. However, the decompiled program contains a subtle bug that is difficult to discover.

6. Related works

Obfuscation is a very useful tool for protecting bytecode. Although there are many commercial or free products available (Dr. Java, 2001; Eastridge, 2000; Hoeniche, 2001; Plumb, 2001; Retrologic, 2000), few researches focus on this topic. LaDue provides a tool named HoseMocha. The tool adds several extra instructions at illegal positions int the bytecode (e.g. after the *return* instruction of a method) to foul the decompilers (LaDue, 1997). Low discusses the concept of obfuscation (Low, 1998b). Low's master thesis concentrates on the obfuscation of control flow (Low, 1998a). Collberg et al. study obfuscation extensively including a survey of obfuscation (Collberg and Thomborson, 2000; Collberg et al., 1997), data obfuscation (Collberg et al., 1998a), and control obfuscation (Collberg et al., 1998b).

Data obfuscation (Collberg et al., 1998a) uses techniques such as splitting variables and merging scalar variables to transform a simple expression into an equivalent, but complex one. It also alters the inheritance structure of types, restructures arrays, and changes procedural abstraction.

Control obfuscation (Collberg et al., 1998b) adds extra predicates in programs to confuse the cracker. These extra predicates, called *opaque predicates*, have a fixed value even though they are computed with complex expressions. The extra predicates obscure the program.

Both data obfuscation and control obfuscation change the bodies of types and methods. To make things complicated, the two obfuscation techniques introduce

additional computations into the bytecode. These additional computations will increase the size of the bytecode. Furthermore, the additional computations will reduce the run-time efficiency of the program.

In contrast, our proposed methods concentrate on eliminating the symbolic information in the constant pool of the bytecode. Several techniques discussed in this paper even introduce syntactic or semantic errors in the decompiled program while preserving the behavior of the bytecode. Most of the proposed techniques do not produce additional code. Furthermore, they usually reduce the size of the bytecode.

The obfuscation methods proposed by Low and Collberg (Collberg and Thomborson, 2000; Collberg et al., 1998a; Collberg et al., 1998b; Collberg et al., 1997; Low, 1998a; Low, 1998b) can cooperate with our proposed approach to construct an even more versatile and stronger obfuscation tool.

7. 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 paper 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, our techniques provide stronger protection for bytecode than other existing techniques.

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. For example, NET common language runtime (CLR) (Gough, 2001), the runtime model of the C# language (Liberty, 2001) is similar to Java's. Therefore, the C# language could be the candidate to apply these obfuscation techniques.

References

Collberg, C., Thomborson, C., 2000. Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection, Tech Report, Department of Computer Science, The University of Auckland, New Zealand.

Collberg, C., Thomborson, C., Low, D., 1998a. Breaking Abstractions and Unstructuring Data Structures. Proceedings of the IEEE International Conference on Computer Languages. pp. 28–38.

Collberg, C., Thomborson, C., Low, D., 1998b. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. Conference Record of the Annual ACM Symposium on Principles of Programming Languages. pp. 184–196.

Collberg, C., Thomborson, C., Low, D., 1997. A Taxonomy of Obfuscating Transformations, Tech Report, Department of Computer Science, University of Auckland, New Zealand.

D&C Software Solutions Inc., 2001. jAscii. ver. 1.0.17. http://www.jascii.com/.

Dr. Java, 2001. Marvin Obfuscator. ver. 1.2. http://www.drjava.de/obfuscator/.

Eastridge Technology, 2000. Jshrink. ver. 1.19. http://www.e-t.com/jshrink.html.

Engel, J., 1999. Programming for the Java Virtual Machine. Addison-Wesley, Reading, Mass.

Gosling, J., Joy, B., Steele, G., Bracha, G., 2000. The Java Language Specification, second ed. Addison-Wesley, MA.

Gough, J., 2001. Compiling for the Net Common Language Runtime. Prentice Hall.

Hoeniche, J., 2001. Java Optimize and Decompile Environment (Jode). ver. 1.1.1. http://jode.sourceforge.net/.

IBM, 2001. Visualage for Java. ver. 4.0. http://www.ibm.com/software/ad/vajava/.

Kouznetsov, P., 2001. Jad—the Fast Java Decompiler. ver. 1.58e. http://www.geocities.com/SiliconValley/Bridge/8617/jad.html.

Kumar, K., 2001. JReverse Pro. ver. 1.2.http://www.geocities.com/akarthikkumar/JReverse Pro/.

LaDue, M.D., 1997. HoseMocha. ver. 1.0.http://www.cigital.com/hostile-pplets/HoseMocha.java.

Liberty, J., 2001. Programming C#. O'Reilly.

Lindholm, T., Yellin, F., 1999. The Java Virtual Machine Specification, second ed. Addison-Wesley, Reading, MA.

Low, D., 1998a. Java Control Flow Obfuscation, Master Thesis, University of Auckland, New Zealand.

Low, D., 1998b. Protecting Java Code Via Code Obfuscation. ACM Crossroads 4 (3), 21–23.

Mayon Software Research, 2001. Classcracker. ver. 2.02. http://www.pcug.org.au/~mayon/.

Meyer, J., Downing, T., 1997. Java Virtual Machine. O'Reilly, Cambridge, Mass.

Plumb Design, Inc., 2001. Condensity Professional Edition. ver. 2.0. http://www.condensity.com/index.html.

PsychoticSoftware Inc., 2001. Classspy. ver. 2.0.3. http://www.psychoticsoftware.com/Products/ClassSpy/index.jsp.

Retrologic Inc., 2000. Retroguard Bytecode Obfuscator. ver. 1.1. http://www.retrologic.com/.

Venners, B., 1998. Inside the Java Virtual Machine. McGraw-Hill, New York.

Vliet, H.v., 1996. Mocha. ver. beta 1. http://www.brouhaha.com/~eric/computers/mocha.html.