

Approaches to Make the Decompiled Java Programs Uncompilable

JIEN-TSAI CHAN and WUU YANG

Department of Computer and Information Science, National Chiao Tung University
Hsinchu, Taiwan 300, R.O.C.

(email: {maxchan, wuuyang}@cis.nctu.edu.tw

ABSTRACT

There are several available obfuscation tools for protecting compiled Java programs—bytecode files. Most of the tools just scramble the identifier information stored in a bytecode file. This job can easily be done by textually substituting the identifiers with sequentially or randomly generated meaningless names. However, these new unique identifiers cannot deter an experienced reverse engineer too long. In this paper, we proposed several techniques that make the decompiled program difficult to understand and to recompile. Several techniques are proposed to introduce syntax and semantic errors into the decompiled program while keeping the original behaviors of the bytecode. The reverse engineer has to debug the

decompiled program manually. Although most of the methods we proposed are only scrambling the identifiers in a Java bytecode file, the bytecode file become much harder to be reverse engineered than other identifier scrambling techniques.

Keywords

Program protection, bytecode obfuscation, Java programming language.

1 INTRODUCTION

All the names of types, fields, and methods of a Java program are stored in the constant pool within the compiled program—the bytecode file [1-4]. These names and the simple stack-machine instructions facilitate the decompilation of the bytecode file.

There are many decompilers of Java available freely or commercially [5-11]. The decompiled program is almost identical to the original source program. The decompilers of Java become the lethal weapons of intellectual property piracy.

Obfuscation tools are one of the major defenses against the decompilers. Most of the obfuscation tools just scramble the symbolic information (identifiers) in the constant pool of a bytecode file [6, 12-15]. A meaningful name is substituted by a sequentially or randomly generated meaningless name.

The techniques discussed in this paper introduce some compilation errors in the decompiled program while keeping the modified bytecode function correctly. These techniques transform a bytecode file to a superset of the original compilable program. The extra parts look like general programs but will result in some syntax and semantic errors for a Java compiler. Therefore, the reverse

engineer has to debug the decompiled program manually.

In a Java program, an identifier may represent a type, a field, a method, a parameter, or a local variable. The Java compiler may be confused about which entity should be used when an identifier represents many entities at the same time. Therefore, many rules defined in the Java language specification are used to clarify the confusion.

Once the entities are determined after the compilation, those rules have not to be obeyed. Therefore, the main principle of these techniques is to violate some rules of the Java compiler to protect the bytecode file. The decompiled program cannot be re-compiled successfully. These transformations cannot be easily undone by an automatic tool.

2 THE APPROACHES

2.1 Overloading Unrelated Methods

In Java, methods are determined on a signature-by-signature basis. The signature of

a method includes the name of the method and the number and the types of the formal parameters. This means that two methods will be treated as two different methods if they have the same name but different numbers or types of the formal parameters. Such methods are called overloaded methods.

We can use the same name for the methods that have different names and different number or types of the formal parameters. Therefore, a method is further obscured because the methods can only be differentiated by the number or the types of the formal parameters.

With the widening conversions, this technique can result in surprising benefits. In Java, there are two kinds of widening conversions (i.e. coercion)—the widening primitive conversion and the widening reference conversion. Method invocation allows the use of both widening conversions

and performs them automatically and implicitly. 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` invoke the method `m`. When the bytecode is transformed with this technique, the decompiled program by the Jad decompiler [7] becomes

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

The method invocation `g(i)` will invoke the method `g(long)` (the original `p(long)`) because an integer prefers being converted to the *long* type to the *float* type. The behavior of the decompiled program is silently changed by the transformation. This

Table 1. The decompiled results of using illegal characters.

Decompiler	Use keyword as identifiers	Use illegal characters in indentifiers
Jad	“ fldfalse”	“< 3E 3F 21 23 ”
jAscii	“ fldfalse”	“<?#!#”
Mocha	“false”	“<?#!#”
deClassify	“false”	“<?#!#”
JReverse Pro	“false”	ERROR
ClassSpy	“false”	“<?#!#”
Jode	“false”	ERROR

kind of semantic errors provide better protections on bytecode.

2.2 Illegal Identifiers

The Java language specification [16] defines that an identifier should be a Java letter followed by letters or digits. An identifier cannot be the same as a keyword, boolean literals or the null literal. These definitions help the lexer and the parser of Java to analyze the program correctly. However, these rules need not be obeyed in the bytecode file. Although the bytecode will be verified when JVM loads the files, JVM does not verify whether the names in the constant pool comply with the definition of an identifier in the Java language specification. Therefore, the names in the constant pool of a

bytecode file could be changed to use those illegal symbols, boolean literals, and the null literal. When a transformed bytecode file is decompiled, these illegal identifiers will result in compilation errors.

For example, we can change an identifier in a bytecode file to be the boolean literal “false” or “<?#!#”. The bytecode still runs well. However, decompilers will face troubles for this technique. We tested this technique with several available decompilers. Many decompilers use Jad as the decompiling engine and add their own user interface. Only the decompilers based on different decompiling engines are chosen for the experiment [5-11]. The result is shown in Table 1.

Table 2. The symbols that are unfit for an identifier in JVM.

Symbol	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 symbol in Windows system.
\$	nested type separator
.	the separator in fully qualified names. should not be used in the name of a type

It is noticeable that not all the symbols can be used for an identifier in the constant pool. Some symbols have special meaning for JVM and the host system. The symbols that are unfit for an identifier are listed in Table 2.

All the constructors of a type use the name “<init>” in the bytecode file. The name of constructors should be avoided to be used for obfuscation. Otherwise, JVM will be confused when calling constructors.

The name “<clinit>” is the name of the method that serves as the static initializer of the type. JVM will call it to initialize the static members of a type.

The symbols “/”, “\”, and “:” should not to be used as the name of a type. The three symbols are used as separators in the host file system for 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 [17], which uses a database system (called ENVY) to manage the bytecode files. If the three symbols are used as the name of a type and the type is stored in the host file system, these separators will cause the JVM to misinterpret the type.

The symbol “\$” is used as the separator of a type and its nested types. Arbitrarily using “\$” in an identifier may cause some

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;

unexpected results. However, it can be used carefully to introduce another kind of protections. See Section 2.4 for details.

2.3 Some Interesting Examples

We use a few characters that have specific meanings in a Java source program to rename the identifier in the bytecode file. 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 file is changed to a name made up the above characters. The decompilation results by different decompilers are shown in Table 3.

2.4 Nested Type Name

According to the Java language specification [16], a nested type cannot have the same name as its enclosing types. Consider the following program.

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

The above example cannot pass the compilation because the Java compiler cannot determine which `M` should be used in the declaration of the local variable `m`.

After the compilation, the name of a nested type `N` in the enclosing type `M` becomes `M$N`. Furthermore, `N` is compiled to be an

independent bytecode file named “M\$N.class”.

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

After the name of the nested type is renamed, the name of the bytecode file and the corresponding symbolic reference also has to be renamed to comply with the new name. Otherwise, JVM cannot find the file when trying to load the nested type.

2.5 Static Methods vs. Instance Methods

According to the specification of the Java language [16], an inherited static method of the superclasses cannot be overridden by an instance method with the same signature in a subclass. Similarly, an inherited instance method of the superclasses cannot be overridden by a static method with the same signature in a subclass.

There are two methods to violate this rule. They all require that both the superclasses and the subclasses are in the transformation scope. The first method is that we can add a fake static method in the supertype (or subtype) for an instance method and add an instance method in the supertype (or subtype) for a static method. Suppose that there is an instance method m in class Y and Y inherits class X . We can make a method m' in X . Method m' has the same signature as method m . To make the fake one looks like a real method, the body of m' can be the same as that of m . Furthermore, we can make m' to be a buggy version of m . It is hard to detect which method is the correct one when semantic errors are introduced into the buggy version.

The second method 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 {...}
}

```

Although the return types of `m` and `n` are different and their *throws* clauses are not compatible, there is no problem for this situation. The return type and the *throws* clause of a method is not a part of the signature. However, if both `m` and `n` were instance methods, renaming them to `p` makes `n` to override `m`. The overriding violates another compiler rule that an instance method and the overridden inherited instance method must have the same return types and

compatible *throws* clauses. It may provide some kind of protection. But the overriding may change the method to be invoked when JVM dynamically dispatches an instance method. Therefore, arbitrarily overriding instance methods is not allowed in the algorithm. This technique requires that one of the methods is a static method and the other is an instance method.

JVM uses four instructions—*invokestatic*, *invokeinterface*, *invokevirtual*, *invokespecial*—to invoke methods. A static method is invoked by the instruction *invokestatic*. An instance method is invoked by the instructions *invokevirtual* or *invokeinterface* according to the declared type of the reference is a class or an interface. An instance method can also be invoked by the instruction *invokespecial*, which specially handles the method invocations of the instance methods of the superclass, the private methods, and the instance initialization. The

differences between the four instructions are the lookup procedure of resolving the method to be invoked. Because static methods and instance methods are invoked by different instructions, renaming m and n to be p will not change the behavior of the above example. However, the decompiled program cannot be re-compiled.

3 SUMMARY

A good obfuscation algorithm should

- preserve the semantics of a program.
- delay the reverse engineering as long as possible.
- be difficult to be undone by an automatic tool.
- preserve or improve the efficiency and not increase the file size too much.

The techniques proposed in this paper comply with all the above requirements. Preserving the semantics of a program is the most important rule when designing these

techniques. Many techniques make the decompiled program to be not re-compilable. The transformed effects cannot be easily undone by automatic tools. The reverse engineers have to debug the decompiled program manually. Therefore, a successful reverse engineering will cost a lot of time and efforts.

The main purpose of the techniques proposed in this chapter is scrambling the symbolic names and the symbolic references in the bytecode file. Although the technique of identifier scrambling has appeared for a long time and many commercial or free products are available, the proposed algorithm and techniques provide more advanced protections for bytecode files than other similar techniques.

ACKNOWLEDGEMENT

This work was supported in part by National Science Council, Taiwan,

R.O.C. under grant NSC 89-2213-E-009-146 and NSC 90-2213-E-009-142.

REFERENCE

1. Engel J. *Programming for the Java virtual machine*. Addison-Wesley: Reading, Mass., 1999.
2. Lindholm T, Yellin F. *The Java virtual machine specification (2nded)*. Addison-Wesley: Reading, MA, 1999.
3. Meyer J, Downing T. *Java virtual machine*. O'Reilly: Cambridge, Mass., 1997.
4. Venners B. *Inside the Java virtual machine*. McGraw-Hill: New York, 1998.
5. D&C. jAscii program. <http://www.jascii.com/>[2001].
6. Hoenicke J. Java Optimize and Decompile Environment (JODE) program. <http://jode.sourceforge.net/>[2001].
7. Kouznetsov P. Jad - the fast JAVa Decompiler program. <http://www.geocities.com/SiliconValley/Bridge/8617/jad.html>[2001].
8. Kumar K. JReversePro program. <http://www.geocities.com/akarthikkumar/JReversePro/>[2001].
9. Mayon. ClassCracker program. <http://www.pcug.org.au/~mayon/>[2001].
10. PsychoticSoftware. ClassSpy program. <http://www.psychoticsoftware.com/Products/ClassSpy/index.jsp>[2001].
11. Vliet Hv. Mocha program. <http://www.brouhaha.com/~eric/computer/mocha.html>[1996].
12. Dr.Java. Marvin Obfuscator program. <http://www.drjava.de/obfuscator/>[2001].
13. Eastridge. Jshrink program. <http://www.e-t.com/jshrink.html>[2000].
14. Retrologic. RetroGuard Bytecode Obfuscator program. <http://www.retrologic.com/>[2000].
15. Plumb. Condensity Professional Edition program. <http://www.condensity.com/index.html>[2001].
16. Gosling J, Joy B, Steele G, Bracha G. *The Java Language Specification (2nded)*. Addison-Wesley: MA, 2000.
17. IBM. VisualAge for Java program. <http://www.ibm.com/software/ad/vajava/>[2001].