Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 78 (2005) 60–72
The Journal of Systems and Software
www.elsevier.com/locate/jss

Overcoming the obfuscation of Java programs by identifier renaming

S. Cimato, A. De Santis, U. Ferraro Petrillo *

Dipartimento di Informatica ed Applicazioni, Universitá degli Studi di Salerno, Via S. Allende, 84081 Baronissi (Salerno), Italy

Received 3 August 2004; received in revised form 4 November 2004; accepted 14 November 2004
Available online 8 January 2005

Abstract

Decompilation is the process of translating object code to source code and is usually the first step towards the reverse-engineering of an application. Many obfuscation techniques and tools have been developed, with the aim of modifying a program, such that its functionalities are preserved, while its understandability is compromised for a human reader or the decompilation is made unsuccessful. Some approaches rely on malicious identifiers renaming, i.e., on the modification of the program identifiers in order to introduce confusion and possibly prevent the decompilation of the code.

In this work we introduce a new technique to overcome the obfuscation of Java programs by identifier renaming. Such a technique relies on the intelligent modification of identifiers in Java bytecode.

We present a new software tool which implements our technique and allows the processing of an obfuscated program in order to rename the identifiers as required by our technique. Moreover, we show how to use the existing tools to provide a partial implementation of the technique we propose.

Finally, we discuss the feasibility of our approach by showing how to contrast the obfuscation techniques based on malicious identifier renaming recently presented in literature.

Keywords: Java obfuscation; Program protection; Decompilation

1. Introduction

The diffusion of Java has deeply affected the way simple programs or complex applications are developed and distributed. The Java platform comes with a rich set of function libraries which ease the task of software developers and make the language well suited for the development of programs in several application areas. Furthermore, Java applications are composed of dynamically loaded pieces of code which can be downloaded across the network and linked at runtime as required, providing an extensible programming environment.

Another characteristic of the Java language which contributed to its wide diffusion is its portability: Java programs are compiled into a neutral platform format, the bytecode, which can be executed by a Java Virtual Machine (JVM) running on the targeted platform. Since much of the information needed for the execution of a bytecode is stored as symbolic references and the JVM has been designed with a very simple architecture, the bytecode format is relative simple compared to the complexity of the machine code executed by a real microprocessor. Furthermore, a large source of documentation is available for both the Java Virtual Machine and the Java language. These facts make the decompilation process very easy and constitute a threat for the interests of software developers and companies investing money in the distribution of Java based software.

Decompilation is the process of translating object code to source code and is relevant to the security of a Java application for a number of reasons. Indeed, reverse engineering an executable code favors software

* Corresponding author. Tel.: +39 089965416; fax: +39 089965272.
E-mail addresses: cimato@dia.unisa.it (S. Cimato), ads@dia.unisa.it (A. De Santis), umbfer@dia.unisa.it (U. Ferraro Petrillo).

0164-1212/$ - see front matter © 2004 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2004.11.019

piracy, making the attacker able to thief the intellectual property of the software, appropriating and reusing (and possibly reselling) a solution to a given problem. Furthermore, such a process opens a number of security breaches by allowing an attacker to discover vulnerabilities in the application or retrieve sensitive information (password, license number, credit card number) hidden in the executable code.

The Java standard software development kit comes with a basic disassembler included in the provided standard development tools: `javap` allows users to disassemble generated bytecode and retrieve some of the information contained in the class file, such as the name and the type of variables and methods included in the class. Since the early days of Java, many third-party decompilers were also available on the market providing an easy way to retrieve source code from a compiled class. At the same time, code obfuscators were developed with the aim of making decompilation process hard.

1.1. Obfuscation techniques

The obfuscation process aims to modify the compiled code such that its functionalities are preserved, while its understandability is compromised for a human reader or the decompilation is made unsuccessful. Many practical approaches have been developed, based on the application of different kinds of transformation to the original source (or machine) code. Following Collberg et al. (1997, 1998), an obfuscating transformation T changes the source program P into a program P', such that if P fails to terminate or terminates with an error condition, then P' may or may not terminate, otherwise P' must terminate and produce the same output as P.

Obfuscating transformations can be classified according to their target, and the kind of modification they operate on the code. According to the classification in Collberg et al. (1998), it is possible to distinguish four basic categories of obfuscation, namely:

- *layout obfuscation*: affecting the information which is not really necessary to the execution of the program, such as identifier names and comments;
- *data obfuscation*: affecting the data structures used in the program, such as the way variables are stored in memory, their encoding, the way data are aggregated and so on;
- *control obfuscation*: modifying the control flow of programs, such as procedure aggregation (replacing a procedure with the statements it is composed of), ordering (changing the order of statements in procedures), and so on;
- *preventive obfuscation*: stopping decompiler operation, such as adding instructions which cause the decompiler to crash.

Obfuscation techniques based on the malicious renaming of the identifiers have been recently presented in Chan and Yang (2004). Such techniques can be classified as a form of *layout obfuscation*, since they reduce the information available to a human reader which examines the target program, or of *preventive obfuscation* since they aim to prevent the decompilation or to produce an incorrect Java source code. Such techniques try to hide the structure and the behavior information embedded in the identifiers of a Java program by replacing them with meaningless or confounding identifiers to make more difficult the task of the reverse engineer. It is worth to notice that the information associated to an identifier is completely lost after the renaming. Furthermore, by replacing the identifiers of a Java bytecode with new ones that are illegal with respect to the Java language specification, such techniques try to make the decompilation process impossible or make the decompiler return unusable source code. The authors in Chan and Yang (2004) observed that such effects will not be easily countered by the existing decompilation technologies forcing the cracker to spend lots of time to understand and debug the decompiled program manually.

1.2. Our contribution

The contribution of this paper is to introduce a new technique to overcome the obfuscation of Java programs based on the malicious renaming of the identifiers. The technique we propose is based on the intelligent renaming of the identifiers in an obfuscated program in order to avoid any confusion for both the tools or the human examining and decompiling the program itself. Independently of the obfuscating renaming strategy used, we show that it is possible to contrast the obfuscation by renaming the identifiers in two phases, in order to firstly overcome the preventive obfuscation and, then, to add type information to the identifiers in the source code so as to contrast layout obfuscation.

We present a prototype implementation of a software tool which preprocesses an obfuscated program in order to rename its identifiers as required by our technique. This makes it possible to decompile the resulting deobfuscated program by using one of the existing decompilers. The tool, called ADAM, is available at `http://www.dia.unisa.it/research/adam`. Furthermore, we show how it is possible to provide a partial implementation of the technique we propose so to contrast in many cases the malicious identifier renaming by using the existing tools.

We examine examples of obfuscation techniques, downloadable together with the ADAM tool, based on malicious renaming and show how the decompilation is operated in an efficient and automatic way by applying our technique.

1.3. Organization

The paper is organized as follows. In Section 2 we review some of the available tools for decompilation and obfuscation of Java code. Obfuscation techniques based on the renaming of identifiers are discussed in Section 3. In Section 4 we present our technique for overcoming the obfuscation of Java programs by identifier renaming. In Section 5 we show the application of our technique to contrast several examples of advanced obfuscation based on malicious identifier renaming recently presented in literature (Chan and Yang, 2004). Finally, in Section 6 we outline some conclusions and discuss some future directions for our work.

2. Java decompilers and obfuscators

Decompilation is the process of turning object code, that is code expressed in a language that is executed directly by a real or virtual machine, into source code, that is high level programming language. Actually, such process is quite hard since it essentially relies on the retrieval of large-scale high-level behavior from small-scale low-level behavior. With respect to standard object code, the decompilation of the Java bytecode is a simplified process, since it includes more information and provides a higher level representation. However, while it is relatively easy to retrieve the source code from a bytecode obtained by a standard compiler with a known compilation strategy (such as `javac`), the decompilation of arbitrary bytecode, such as the one produced by unknown compilers or subjected to optimization, can be a difficult task.

Many decompilation tools construct an internal representation of a class file enriching the information retrieved from the bytecode in order to make easier the decompilation. Indeed, the basic challenges encountered during such a process are the recovering of the correct typing of the variables and of the high level control constructs. To these purposes, the bytecode is translated into intermediate representations on which type inference algorithms and flow analysis techniques are applied. Usually Control Flow Graphs (CFG) are produced, where nodes represent (block of) operations and edges the flow of control. On such structures, several transformations are thereafter applied in order to reconstruct the original source code.

The Sable group produced the SOOT framework (Miecznikowski and Hendren, 2002), aimed to the analysis and the optimization of Java bytecode. Based on such framework, they developed the DAVA decompiler (Miecznikowski and Hendren, 2001) which uses three intermediate representations: a list of typed statements corresponding to the bytecode instructions, a control flow graph built upon the previous representation, and a structure encapsulation tree which resembles an abstract syntax tree, used to reconstruct the Java language output.

The JBET tool (McAfee Research, 2003) constructs a directed acyclic graph (DAG) representation of the bytecode corresponding to a class file. In such a representation, method implementations are divided into basic blocks each corresponding to a DAG, where the edges connect "user" nodes (such as constants, or the result of calculations) to "producer" nodes (such as method calls or other calculations). Such graphs are acyclic, since a cycle would mean that a node needed its own value in its computation. Based on the DAG representation and on the dynamic analysis of the bytecode, the JBET tool is in many cases able to determine information needed to recover the original Java source program. Two of the most used freely available decompilers are Jad (Jad—the fast JAva Decompiler), whose engine is used in numerous visual software development environments, and JODE (Java Optimize and Decompile Environment) (Hoenicke, 2002), whose source code is also provided. Among commercial decompilers, SourceAgain (Ahpah Software, 2004) is able to correctly recover many Java control structures and optimizations from the bytecode. An extensive list of decompiler tools together with usage tests is provided at `http://www.program-transformation.org/Transform/DeCompilation`.

Other tools, such as Jshrink (Eastridge Technology, 2004) and Proguard (Lafortune, 2004), provide an integrated environment for the optimization and the obfuscation of bytecode, through the removal of unused code, classes and debugging information, and the renaming of the identifiers. The obtained bytecode is functionally equivalent, but is usually smaller in size, more efficient and harder to reverse engineer. KlassMaster (Zelix Pty Ltd., 2004) provides a wide range of obfuscation techniques, and a specific scripting language allowing users to automate the obfuscation process and integrate it into the software development process.

The Sandmark tool (Collberg et al., 1997, 1998; Collberg and Thomborson, 2002), provides a modular extensible framework for the decompilation, obfuscation and watermarking of Java applications. As soon as new algorithms and techniques are developed, their implementation can be easily added to the original framework, using the dynamic class loading mechanism.

3. Obfuscating a Java program by malicious identifiers renaming

Identifiers are used in the Java language to name entities such as methods, classes, packages, interfaces and variables. The Java language specification (Gosling et al., 2000) defines an identifier as an unlimited-length

sequence of Unicode letters and digits, the first of which must be a letter. An identifier must not be the same as a boolean literal, the null literal, or a keyword in the Java programming language.

One of the most used obfuscation techniques consists of renaming the identifiers of a Java bytecode so to make it harder to be decompiled or to be understood. A *renaming strategy* is a set of rules for transforming the identifiers of a Java bytecode in an arbitrary sequence of Unicode characters. The aim of a renaming strategy is to make a target Java code more difficult to be understood (*layout obfuscation*) or to be decompiled (*preventive obfuscation*). In the first case, the original identifiers are replaced with some meaningless or confounding ones. It is worth to notice that it is impossible for a cracker to recover the original identifiers: the information associated to an identifier is completely lost after the renaming. In the second case, the identifiers are renamed in order to make it difficult, or even impossible, for a decompiler to translate the program back to Java source code or to make the resulting source code uncompilable. Notice that, in both cases, the renaming of identifiers does not affect at all the behavior of the obfuscated program.

A *malicious renaming* is the operation of obfuscating a Java bytecode by applying a renaming strategy. Obfuscation techniques based on malicious renaming have been well-known and practiced for a long time. Simple renaming strategies include: replacing the original identifiers of a Java bytecode with meaningless ones (e.g., see the Klassmaster obfuscator, Zelix Pty Ltd, 2004), shuffling them (e.g., see the JAurora obfuscator, de Roo and van den Oord, 2003) or replacing them with some other ones according to a user-defined translation table (e.g., see the JObfuscator obfuscator, Helseth Digital Systems Inc. (2001)).

3.1. Which identifiers can be renamed?

When we rename the identifier of an entity in a Java program, we also need to update all the references to this entity accordingly, otherwise the obfuscated program may not run correctly. However, it is not always possible to update external references to the renamed identifiers.

Consider, as an example, the case of an abstract Java class `Y` that is defined in the Java runtime library. Suppose we define a new class `X` that extends and implements the methods of `Y`. The renaming of all the methods of the class `X` would be possible only by renaming the methods of the class `Y` too. But this would require the modification of the Java runtime library and, thus, it would break the compatibility, of the obfuscated application, with standard Java installations.

Following the definition given in Chan and Yang (2004), we introduce the concept of *obfuscation scope* as the set of identifiers in a Java program that can be safely renamed for obfuscation purposes. By default, this set does not include the identifiers of parameters and local variables because these information are usually stripped off when compiling a bytecode. The obfuscation scope of a program can be built by grouping all the identifiers existing in the program and, then, removing those ones whose renaming could introduce undesired or erroneous behavior.

We also observe that there are some cases where the set of possible candidates for the renaming is very small. For example, this is the case of programs that determine, at run-time, which classes have to be loaded or referenced, which methods have to be invoked or which fields have to be accessed. In these cases, arbitrary program identifiers could be referred to by program variables whose content will be known only at run-time; in such a situation it is, by far, more difficult for an obfuscator to determine which identifiers are referred to in a program and to update these references according to the renaming.

3.2. Advanced malicious renaming techniques

Several advanced techniques for the obfuscation of Java programs based on malicious renaming have been proposed in Chan and Yang (2004). The common approach of all these techniques is to rename the identifiers in the obfuscation scope of a Java bytecode by reusing the same identifier as often as possible. The decompilation of the resulting code will be indeed more difficult to be understood by a malicious user because of the redundancy of these identifiers. Moreover, by referring to different unrelated entities with a same identifier, decompilation tools will be fooled as well leading, in many cases, to the generation of incorrect Java source code. The expectation of the authors is that these effects will not be easily contrasted by existing decompilation technologies; some of these could also be unable to decompile the code at all, thus forcing the cracker to spend lots of time to understand and debug the decompiled program manually. Instead, we will show that these effects can be overcome by smartly renaming the identifiers of an obfuscated bytecode in an automatic way with the help of the existing bytecode manipulation and decompilation technologies. In the following we describe some of these techniques.

3.2.1. Overusing identifiers

The strategy of obfuscating by overusing identifiers operates by renaming the entities of a Java bytecode so to reuse an identifier as often as possible. Such a strategy confuses the cracker because he has to understand the behavior of a decompiled program where each

identifier has, at the same time, several different meanings depending on the context where it does appear.

This renaming strategy is described by the following simple algorithm. Let us consider a set of bytecodes that has to be obfuscated. First of all, the obfuscation scope (i.e. the set of entities that can be safely renamed) of the considered bytecodes has to be determined. Then, the package and the inheritance structures of all the classes and of all the interfaces in the obfuscation scope are built. The renaming begins by traversing the package structure of the bytecodes. During the traversal, packages, interfaces, and classes are relabeled by using sequentially generated names (e.g., `a`, `b`, `c`). The next step is to traverse the inheritance structure of all the considered classes and interfaces. During the traversal and for each type `T` (either a class or an interface), inner types, fields and methods are relabeled each using the sequentially generated names adopted in the previous step. Finally, all the references to the entities that have been renamed are updated accordingly.

3.2.2. Overloading unrelated methods

The obfuscation technique based on the overloading of unrelated methods relies on the concept of *widening conversion* and *methods overloading*. The Java language specification defines a *conversion* from type `S` to type `T` as the operation that allows an expression of type `S` to be treated, at compile time, as if it were of type `T` instead. A conversion is *widening* if the range of values supported by the target type `T` is wider than the one of the source type `S`.

If two methods of an interface or of a class have a same name `X` but different signatures, then the method name `X` is said to be *overloaded*. Whenever, in a Java source code, the overloaded method `X` is invoked, the compiler generates a bytecode containing the invocation of the method `X` whose formal parameters types perfectly match with the types of the arguments provided with the invocation. If such a match does not exists, then the method `X` whose formal parameters types are "wider" and "nearest" to the arguments provided with the invocation, is invoked (Gosling et al., 2000).

The obfuscation by overloading unrelated methods consists in renaming with the same identifer all the methods of a compiled class, wherever this is possible. This change does not affect the behavior of the bytecode, because the identity of the methods to be executed in a Java program are resolved, via a symbolic reference, at compile time before performing the obfuscation. Instead, such a change can drastically change the behavior of a program if this will be decompiled and then recompiled again.

To explain this, suppose that in the original code there were several methods having distinct names but with the same number and similar types of arguments. By renaming these methods using the same name, they will all become overloaded. The invocation of one of these methods in the original source code may now become troublesome to handle when recompiling the decompiled code. Here, the Java compiler has to choose which one of the overloaded methods best matches the types of the arguments provided with the method invocation. As a matter of fact, it is not guaranteed that the original (and thus correct) method will be chosen, because there could exist another method that better matches the types of the input arguments.

A similar technique can be implemented by renaming with the same identifiers the field variables of a class and of all its sub-classes. While the bytecode will be unaffected by this change because references to field variables are local to the class, the decompilation of the obfuscated code will return source code where field variables declared in the parent are erroneously overridden in the sub-classes.

3.2.3. Introducing illegal identifiers

The rules for naming identifiers in the Java language are stricter than the ones used for the Java bytecode. For example, it is possible to use, as identifiers in a bytecode, the string literals that are reserved words in the Java language (e.g., "try", "catch"). Furthermore, there exist several characters, such as "," or ".", that can be used to name the identifiers of a Java bytecode but cannot be used in the Java language since they have some special meaning (Gosling et al., 2000). As a consequence, there may exist some *legal* identifiers in a Java bytecode that become *illegal* when the bytecode is translated back to a source code.

It is possible to obfuscate a Java bytecode by replacing its identifiers with new illegal ones. Many decompilers will either produce a Java code that is uncompilable, because it defines illegal identifiers, or they will be unable to decompile the bytecode at all.

3.2.4. Renaming nested types

A nested type is a type that is defined inside another type. In a Java program it is possible to refer to a nested type by just using its name. For this reason, the Java language specification does not allow a nested type and its enclosing type to have the same name so to avoid any potential ambiguity when referring to either one of these. Therefore, it is not possible, for example, to define a nested type named `B` inside a type with the same name `B`.

Differently from the Java language, the standard bytecode language represents a nested type (e.g., `B`) in a machine independent way by prepending to its name the name of its enclosing type (e.g., `A`) followed by the "$" character (e.g., `A$B`). This allows the bytecode interpreter to distinguish between enclosing and nesting types, even if they have the same name.

The obfuscation by renaming nested types works by redefining the names of the types in a Java bytecode so that these match with the names of their corresponding enclosing types. A straightforward decompilation of the obfuscated code produces an incorrect source code where nested types have the same name of their enclosing types.

3.2.5. Overriding static methods

The Java language specification does not allow an instance method `M` defined in a class `A` to be overridden by a static method M' defined in a subclass of `A` and having the same signature of `M`, or vice-versa.

This restriction does not hold for Java bytecode. In this case, there is no ambiguity between static methods or instance methods since the Java Virtual Machine uses different instructions for invoking either of them (i.e., `invokeVirtual` or `invokeInterface` for instance methods, `invokeStatic` for static methods).

The obfuscation technique based on the renaming of static methods processes a set of compiled Java classes and renames, wherever possible, static methods (respectively, instance methods) using the name of instance methods (respectively, static methods) defined in one of their superclasses and having the same number and types of formal parameters. A decompilation of the obfuscated code produces an incorrect Java source code where instance methods are overridden by static methods, and/or vice versa.

This technique can be applied also to bytecodes that do not have static methods and instance methods with the same signature simply by introducing fake static and instance methods.

4. Good renaming for contrasting malicious renaming

The technique we propose for overcoming the effects of malicious renaming consists of properly renaming all the entities of an obfuscated code. Our technique replaces obfuscated identifiers with new identifiers that are legal and distinct. In this way, it is possible to contrast the effects of preventive obfuscation techniques by avoiding any ambiguity when parsing the identifiers during the decompilation of a Java bytecode. At the same time, it is possible also to reduce the effects of layout obfuscation techniques since confounding identifiers are replaced with meaningful and non-confounding ones. Indeed, each identifier is generated in a way to include the information which can be extracted about the entity it identifies. This improves the readability and the understandability of the decompiled code. We refer to our technique as to a *good renaming* technique as it can be used to contrast the effects of malicious renaming techniques.

In our technique each identifier in the obfuscation scope of the obfuscated bytecode is renamed using the following information:

Entity_Number_Info

where

- Entity is a string literal denoting the category of the entity, namely `class`, `field`, `interface`, `method`, `package`;
- Number is a unique progressive number for each category of entities;
- Info is a string literal containing additional information on the entity, such as its type (for field variables), the return type (for methods), its visibility, the access modifiers, and so on.

Overriding relationships are preserved by renaming both overriding and overridden methods with a same identifier.

Some of the existing decompilers are already provided with the ability to automatically perform an informative renaming of the identifiers found during the decompilation of a Java bytecode. For instance, SourceAgain is capable to replace each identifier with a name which is unique and denotes the type of the entity it refers to. However, in many cases these tools fail to perform the renaming because this process, that typically occurs *during* the decompilation of the code, is fooled as well by the techniques used to obfuscate the target program.

Since malicious renaming techniques are targeted to contrast the decompilation process while not affecting the structure of the bytecode, a viable approach to implement the good renaming technique we propose is to operate it *before* the decompilation and, thus, at the bytecode level.

We present two possible approaches for implementing the good renaming technique we propose. The first approach relies on the usage of existing decompilation and obfuscation tools but it does provide only a partial implementation of our technique. The second approach uses an ad-hoc bytecode renaming tool, called ADAM, we developed and provides a full implementation of our renaming technique.

4.1. Implementing (partially) good renaming with existing tools

A partial implementation of the good renaming can be carried out by exploiting the same tools used to obfuscate Java programs. Namely, we observe that obfuscators generally manipulate Java programs at a bytecode level and provide, in most cases, the ability to rename entities with confounding but legal identifiers for obfuscation purposes. This implies the possibility of

using obfuscators for canceling the effects of preventive obfuscation techniques based on malicious renaming. Then, the good renaming technique we propose can be partially implemented using existing tools in two steps:

(1) The identifiers of the obfuscated Java bytecode are renamed using distinct identifiers that are legal for the Java language. For example, all the identifiers could be renamed using sequentially generated string literals in lexicographic order (e.g., `a`, `b`, `c`) or using progressive numbers (e.g., `1`, `2`, `3`).
(2) The transformed bytecode is decompiled to source code using a tool able to rename identifiers with unique names, possibly holding also additional information extracted from the bytecode.

Notice that the first step can be performed by using one of the obfuscators with renaming capabilities (e.g., KlassMaster), while the second step can be fulfilled by using a decompiler with smart renaming capabilities (e.g., SourceAgain).

On the one side, such an approach has the advantage that it can cancel the effects of preventive obfuscation since non-related entities are renamed using distinct legal identifiers. Moreover, it can be carried out in automatic way using only existing tools. On the other side, it has the disadvantage that the new names of the identifiers do not carry all the information required by our renaming technique and, thus, the code resulting from the decompilation may be more difficult to be understood.

4.2. ADAM: a tool for good renaming

The good renaming technique can be fully implemented by developing an application which operates at bytecode level and redefines the identifiers exactly as required by our technique. To this purpose, it is possible to use a large variety of bytecode transformation libraries such as Javassist (Chiba and Nishizawa, 2003), JikesBT (Laffra et al., 2000), JOIE (Cohen et al., 1998), and BCEL (Dahm et al., 2003).

The tool we developed, ADAM (Another Decompilation Assistant Methodology), can be used as a decompiler pre-processor so as to redefine the identifiers of a set of Java classes before these are decompiled. The amount of information that the new identifiers will carry can be parameterized. Thus, for example, the tool can be configured so as to not report the access level of field variables when renaming them or so as to use abbreviations when describing the category of each renamed entity (e.g., `f` in place of `field`). The transformation implemented by our tool takes place in several steps:

(1) The set of classes to be renamed are loaded in memory using the JikesBT library.
(2) An internal graph-based representation of the entities of the loaded classes and of their relationships is built.
(3) The set of loaded classes is traversed and the entities defined in each class are redefined according to the strategy defined in Section 4.
(4) The original set of obfuscated Java classes is replaced with the set of transformed Java classes.

Notice that the transformation of an obfuscated bytecode performed by ADAM is completely automatic and does not requires any human intervention.

From a technical point of view, identifiers are renamed using the functions provided with the JikesBT library. These functions allow to browse, per category, all the entities defined in a Java class and to rename their identifiers. References to renamed entities will be automatically updated. Moreover, we explicitly preserved overriding relationships by renaming at the same time both overriding and overridden methods using the same identifier.

5. Experiments with good renaming

In this section, we present some experiments concerning the application of our approach in the decompilation of Java bytecodes obfuscated using the advanced malicious renaming techniques presented in Section 3.2. For each of the malicious renaming techniques, we considered a simple Java program that has been compiled and then obfuscated using that technique. Then, we successfully applied our renaming technique by using our tool to pre-process obfuscated bytecode and Jad to decompile it. We remark here that in all the cases we considered, the proper decompilation of the obfuscated codes has been possible in an automatic way.

Furthermore, we experimented also the partial implementation of our technique by properly using existing tools to decompile obfuscated code. This has been done by pre-processing the obfuscated classes using the renaming capabilities of several obfuscators (i.e., JShrink, KlassMaster, ProGuard) and then decompiling the outcoming classes using the SourceAgain decompiler. The results of these experimentation show that this approach is able to contrast all the preventive obfuscation techniques we considered as the decompilation was always successful. On the other hand, the amount of information encoded in the outcoming source code was lesser than the one required by our technique thus affecting the overall understandability of the code.

We now show the results of our experiments with the application of our technique on Java programs obfuscated using the techniques presented in Section 3.2. The code of all these experiments, including the original

source codes, the obfuscated bytecodes and the source codes decompiled by using ADAM and Jad, is included in the ADAM software distribution.

5.1. Overcoming obfuscation by overusing identifiers

The rationale of the obfuscation technique by overusing identifiers is to use the same identifier to label entities of different types existing in a program. The expected effect is that the resulting bytecode, once decompiled, will be difficult to understand because of the redundancy of the identifiers.

An example of this technique is shown in Fig. 1. We have chosen, to this end, a simple Java class used to determine if two integers are relatively prime. The obfuscated code, whose corresponding source code is shown in the same figure, uses only the identifiers `a`, `b`, and `c` to label the class, the methods of the class, and the fields of the class.

In Fig. 2, we show the resulting code produced by the application of the ADAM tool to the obfuscated class and the subsequent decompilation using Jad. The result is compared to the code obtained from the partial implementation of good renaming relying on KlassMaster as obfuscator tool and SourceAgain as decompiler tool. Even if in both cases the resulting deobfuscated code does not have any ambiguity in the naming of identifiers, the information provided by the code resulting from the application of our technique allow a better comprehension of the class behaviour.

5.2. Overcoming obfuscation by overloading unrelated methods

The technique of overloading several unrelated methods works by renaming all the methods of a class using a single identifier. The expectation is that whatever method was executed in the original source code by means of a widening conversion, the source obtained by decompiling the obfuscated code could erroneously execute a different method having the same name.

An example of the application of this technique is reported in Fig. 3. The class `A` defines the methods `f`, `X` and `Y`, with the method `f` executing the method `X`. In the obfuscated code, both the methods `X` and `Y` are renamed to `a`.

As a consequence of the decompilation of this code, the original method execution found in `f` becomes ambiguous since it could refer to either one of the two methods named `a`. The Java compiler solves this

Original Code

```
public class test1 {
  private int term1;
  private int term2;
  private boolean areRelativelyPrime;

  public test1(int term1, int term2){
    this.term1 = term1;
    this.term2 = term2;
    areRelativelyPrime =
      areRelativelyPrime();
  }

  private static int
        gcd(int term1, int term2) {
    int remainder;

    remainder = term1 % term2;
    if (remainder == 0) {
      return term2;
    }
    else {
      return gcd(term2, remainder);
    }
  }


  private boolean
    areRelativelyPrime()
    {if (gcd(term1, term2) == 1) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

Obfuscated Code

```
public class a {
  private int a;
  private int b;
  private boolean c;

  public a(int a, int b) {
    this.a = a;
    this.b = b;
    c = c();
  }


  private static int b(int a, int b) {

    int c;

    c = a % b;
    if (c == 0) {
      return b;
    }
    else {
      return b(b, c);
    }
  }


  private boolean c() {

    if (b(a, b) == 1) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

Fig. 1. A Java class that determines if two numbers are relatively prime. The first version presents the original source code. The second version has been obtained by obfuscating the original code with the obfuscation technique of overusing identifiers.

Code Deobfuscated using ADAM

```
public class class_0_pub {

  private int field_int_0_priv;
  private int field_int_1_priv;
  private boolean field_boolean_2_priv;


  public class_0_pub(int i, int j) {
    field_int_0_priv = i;
    field_int_1_priv = j;
    field_boolean_2_priv =
      method_1_priv_ret_boolean();
  }

  private static int
     method_0_priv_ret_int(int i, int j){
     int k = i % j;

    if (k == 0)
        return j;
    return method_0_priv_ret_int(j, k);
  }


  public boolean
     method_1_priv_ret_boolean(){
   if (method_0_priv_ret_int(field_int_0_priv,
                         field_int_1_priv) == 1)
       return true;
    return false;
  }


}
```

Code Deobfuscated using existing tools

```
public class a {

  private int m_int0;
  private int m_int1;
  private boolean m_boolean2;


  public a(int i, int j) {
    m_int0 = i;
    m_int1 = j;
    m_boolean2 = m_function2();
  }


  private static int
        m_function1(int i, int j){
    int k = i % j;

    if (k == 0)
      return j;
    else
      return m_function1(j, k);
  }

  private boolean
        m_function2(){
    if (m_function1(m_int0, m_int1) == 1) {
      return true;
    }
    else {
      return false;
    }
  }
}
```

Fig. 2. The results of the decompilation of the obfuscated Java class that determines if two numbers are relatively prime. The first version has been obtained by pre-processing the obfuscated bytecode with our tool and then decompiling it with the Jad decompiler. The second version has been obtained by renaming the identifiers of the class using the KlassMaster obfuscator and then decompiling the output bytecode with the SourceAgain decompiler.

Original Code	Obfuscated Code	Deobfuscated Code

`public class A {`
 	
`  void X(float a){`
`  }`
 	
`  void Y(long b){`
`  }`
 	
`  void f(){`
`    int s = 1;`
`    X(s);`
`  }`
`}`
	`public class A {`
	
	`  void a(float a){`
	`  }`
	
	`  void a(long  b){`
	`  }`
	
	`  void a(){`
	`    int a = 1;`
	`    a(a);`
	`  }`
	`}`
		`public class class_0_pub {`
		
		`    void method_0_ret_void(float f) {`
		`    }`
		
		`    void method_1_ret_void(long l) {`
		`    }`
		
		`    void method_2_ret_void() {`
		`      int i = 1;`
		`      method_0_ret_void(i);`
		`    }`
		
		`    public class_0_pub(){}`
		`}`

Fig. 3. An example of obfuscation by overloading unrelated methods. The original source code defines a class `A` with three unrelated methods. In the obfuscated code, all the methods of `A` are renamed to `a`. This implies that, in the decompiled code, the method formerly named `f` will erroneously invoke the method formerly named `Y`. The deobfuscated code, obtained by applying our technique by means of the ADAM tool, solves this ambiguity by renaming with distinct identifiers all the methods of `A`.

ambiguity by choosing the method whose arguments' types are nearest to the ones provided as input to the method invocation. In our case, this rule implies the execution of the method formerly named as `Y` thus changing the behavior of the original program.

In Fig. 3 we show the result of the application of the good renaming to the obfuscated code. In this case, all the methods in the bytecode are renamed, by means of our tool, using distinct identifiers before being decompiled. The original method execution will refer, now,

to a method that has a unique identifier (`method_2_ret_void`) and thus no ambiguity may arise when the code is decompiled. We obtained a similar result by using an obfuscator tool, instead of our renaming tool, to partially implement our renaming technique.

A similar approach can be used to contrast obfuscation based on the renaming of unrelated field variables. This technique renames with the same identifier some of the field variables defined in classes that are related by some inheritance relationship. The decompilation of the obfuscated program will return a source code where field variables are erroneously overridden in their subclasses.

By using the ADAM tool, all the field variables that do not have an explicit overriding relationship in the obfuscated bytecode are renamed using distinct identifiers and, thus, no ambiguity may arise when the code is decompiled.

5.3. Overcoming obfuscation by illegal identifiers

The technique of obfuscation by illegal identifiers works by replacing the identifiers of a Java bytecode with new ones that are illegal with respect to the Java language specification.

In order to experiment with this technique, we assembled the bytecode of a simple Java class (see Fig. 4) containing several illegal identifiers: the identifiers we have included to this end are the same tested in Chan and Yang (2004). Then, we successfully renamed illegal identifiers by using our tool. We obtained a similar result by using an obfuscator tool to replace the illegal identifiers with legal ones, as required by the partial implementation of our technique.

5.4. Overcoming obfuscation by nested types renaming

The technique of obfuscation by nested type renaming requires to rename any nested type using the name of its enclosing type. Since such a condition is not allowed by the Java language, the decompilation of the obfuscated code will return an incorrect source code.

The effects of this technique can be invalidated by renaming all the unrelated types in the obfuscated code using distinct identifiers: this guarantees that no nested type can have the same name of its enclosing type. In Fig. 5 we present the example of a simple Java class `M` containing a nested type `N`. In the obfuscated bytecode both the enclosing type and the inner type have the name `M`; hence, their decompilation returns an incorrect source code. We were able to obtain the correct source code by using our tool and, then, using an ordinary decompiler for the source code translation. We obtained a similar result by using, as required by the partial implementation of our technique, an obfuscator in place of our renaming tool to rename the identifiers of the obfuscated program.

5.5. Overcoming obfuscation by overriding static methods

The obfuscation by overriding static methods works by renaming, wherever possible, the static methods of a class (respectively, instance methods) using the name of instance methods (respectively, static methods) defined in one of their superclasses and having the same number and types of formal parameters.

A naive decompilation of the outcoming code produces Java classes where either static methods are overridden by instance methods or vice-versa; both these

Obfuscated Code	Deobfuscated Code

`public class X {`
`  int a.b = 0; // An identifier containing a dot`
`  int a;b = 1; // An identifier containing a semi-column`
`  int a() = 2; // An identifier containing pair of parantheses`
`  int 1   = 3; // An identifier starting with a digit`
`  int     = 4; // An identifier made of three blank spaces`
`  int try = 5; // An identifier matching a Java reserved word`
`}`
	`public class class_0_pub {`
	
	`    public class_0_pub()`
	`    {`
	`        field_int_0 = 0;`
	`        field_int_1 = 1;`
	`        field_int_2 = 2;`
	`        field_int_3 = 3;`
	`        field_int_4 = 4;`
	`        field_int_5 = 5;`
	`    }`
	
	`    int field_int_0;`
	`    int field_int_1;`
	`    int field_int_2;`
	`    int field_int_3;`
	`    int field_int_4;`
	`    int field_int_5;`
	`}`

Fig. 4. An example of obfuscation by using illegal identifiers. The obfuscated Java class defines several field variables using illegal identifiers. The deobfuscated code, obtained by using our renaming tool, replaces illegal identifiers with legal ones.

Original Code	Obfuscated Code	Deobfuscated Code

class M {
 	
  private class N{}
 	
  void f(){
  N n;
  n = new N();
  }
}
	class M {
	
	  private class M{}
	
	  void f(){
	  M n;
	  n = new M();
	  }
	}
		class class_0_pub{
		
		  private class class_1_inner{
		  class_1_inner(){}
		  }
		
		  void method_0_ret_void(){
		  class_1_inner class_1_inner
		  = new class_1_inner();
		  }
		
		  class_0_pub(){}
		}

Fig. 5. An example of obfuscation by renaming nested types. The original source code defines two classes, with the second being an inner class of the first. In the obfuscated code, the inner class is renamed so as to have the same name of the container class. The deobfuscated code, obtained by using the ADAM renaming tool, solves this ambiguity by renaming with different identifiers both classes.

Original Code	Obfuscated Code	Deobfuscated Code

class X {
  static
  int m(int a, char b){
  return 1;
  }
}
 	
class Y extends X {
  boolean n(int c, char d){
  return false;
  }
}
	class X {
	  static
	  int m(int a, char b){
	  return 1;
	  }
	}
	
	class Y extends X {
	  boolean m(int c, char d){
	  return false;
	  }
	}
		class class_0_pub{
		
		class_0_pub(){}
		
		  static
		  int method_0_ret_int(int i, char c){
		  return 1;
		  }
		}
		
		class class_1_pub extends class_0_pub{
		
		  boolean
		  method_1_ret_boolean(int i, char c){
		  return false;
		  }
		
		  class_1_pub(){}
		}

Fig. 6. An example of obfuscation by overriding static methods. The original source code defines two classes, with the first being the parent of the second. In the obfuscated code, the instance method of the child class is renamed so as to match the signature of the static method defined in the parent class. The deobfuscated code, obtained by means of our renaming tool, solves this ambiguity by renaming with different identifiers both methods.

two cases are not allowed by the Java language specification.

In Fig. 6 we present the example of two Java classes, `X` and `Y`, with `X` being the parent of `Y`. These classes are obfuscated by renaming with the same identifier `m` both the instance method defined in `Y` and the static method defined in `X`. The decompilation of the obfuscated code returns an illegal source code where the instance method `m`, formerly named `n`, overrides the static method `m`. The application of our technique solves this ambiguity by renaming the methods of `X` and of `Y` with different identifiers. Thus, the resulting code could be safely decompiled using a standard decompiler. The same result can be achieved by renaming the identifiers of the obfuscated bytecode using an obfuscator tool.

5.6. Overcoming obfuscation by combined identifier renaming techniques

The advanced malicious renaming techniques described in Section 3.2 can be combined together and applied to a set of Java classes in order to improve the obscurity degree of the outcoming code and strengthen its resilience to decompilation. Such a combination is straightforward since the renaming strategy implemented by each of these techniques applies to a distinct kind of identifiers. In Fig. 7 we present as an example the obfuscation of two Java classes, `A` and `C` using all these techniques at the same time. The class `A` defines an inner class `B` and three instance methods. The class `C` derives from the class `A` and defines three member fields and a static method.

Original Code

```
public class A {


    private class B {}


    void Y(long b) {
    }


    void X(float a) {
        B b = new B();
    }


    void f() {
        int s=1;
        X(s);
    }
}


public class C extends A{


    int i = 1;
    int j = 2;
    int k = 3;

    static void
        g(float s){}

}
```

Obfuscated Code

```
public class a {


    private class a{}


    void a(long a) {
    }


    void a(float a) {
        a a = new a();
    }


    void a() {
        int a=1;
        a(a);
    }
}


public class b extends a{


    int 1.2 = 1;
    int 1;2 = 2;
    int 1() = 3;

    static void
        a(float a){}

}
```

Deobfuscated Code

```
public class class_0_pub {

  public class_0_pub(){}

  private class class_1_inner{
    class_1_inner(){}
  }

    void method_1_ret_void(long l){
    }


    void method_2_ret_void(float f){
     class_1_inner class_1_inner1
              = new class_1_inner();
    }

  void method_0_ret_void(){
    int i = 1;
    method_2_ret_void(i);
  }
}


public class class_2_pub
        extends class_0_pub{

    int field_int_0;
    int field_int_1;
    int field_int_2;

    static void
        method_3_ret_void(float a){}

    public class_2_pub(){
        field_int_0 = 1;
        field_int_1 = 2;
        field_int_2 = 3;
    }
}
```

Fig. 7. An example of obfuscation by combining all the advanced malicious renaming techniques described in Section 3.2. The original source code defines three classes: `A`, `B`, and `C`. In the obfuscated code, the entities of these three classes have been renamed according to all the different malicious renaming strategies we considered. The processing performed by ADAM solves the potential ambiguities existing in this code by renaming with different identifiers all the unrelated entities.

The resulting obfuscated code uses two legal identifiers, `a` and `b`, to rename all the identifiers of classes and methods existing in the obfuscation scope. In details, the obfuscation by overloading unrelated methods (see Section 3.2.2) is done by renaming both the original methods `X` and `Y` of `A` to `a`. Obfuscation by nested type renaming (see Section 3.2.4) is obtained by renaming both the original class `A` and its inner class `B` to `a`. Obfuscation by overriding static methods (see Section 3.2.5) is achieved by renaming the static method `g`, defined in the original class `C`, to `a` so as to make it override the non-static method `a` defined in its parent class. Finally, obfuscation by using illegal identifiers (see Section 3.2.3) is obtained by renaming the fields members of the original `c` class using illegal identifiers.

The application of our technique solves all the potential ambiguities existing in the obfuscated code by renaming all unrelated entities using distinct legal identifiers.

6. Conclusions

Obfuscation techniques based on the renaming of identifiers are among the most used ones. They are all based on the notion that, by properly renaming the identifiers of a target program, a decompiler could find it difficult to decompile the obfuscated program or it could generate a source program that is either difficult to understand or it is incorrect.

Indeed, all these techniques share one significant advantage: the original names of the identifiers of the obfuscated code are irremediably lost. This implies that crackers will have more difficulties in understanding the behavior of the program, as they will lack of this important information.

Nevertheless, the other effects claimed by these techniques are debatable. We showed that they can be countered by properly renaming again the identifiers of an obfuscated program before its decompilation is issued. This has been confirmed by the simple renaming

technique we developed to this end. This technique overcomes the effects of several advanced obfuscation techniques based on malicious renaming. We developed a tool for implementing our technique to rename the identifiers of a set of obfuscated classes. The experimentations we conducted have shown that, by using such a tool, it is possible to contrast the advanced obfuscation techniques based on the malicious renaming of the identifiers.

Acknowledgments

We would like to acknowledge the anonymous reviewers for their valuable suggestions and comments.

The work described in this paper has been supported in part by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT. The information in this document reflects only the author's views, is provided as is and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

Ahpah Software, 2004. Sourceagain. Available from <http://www.ahpah.com/product.html>.

Chan, J.-T., Yang, W., 2004. Advanced obfuscation techniques for java byte-code. Journal of Systems and Software 71 (1–2), 1–10.

Chiba, S., Nishizawa, M., 2003. An easy-to-use toolkit for efficient java byte-code translators. In: Proceedings of the Second International Conference on Generative Programming and Component Engineering. Springer-Verlag, New York, Inc., pp. 364–376.

Cohen, G., Chase, J., Kaminsky, D., 1998. Automatic program transformation with JOIE. In: 1998 USENIX Annual Technical Symposium. pp. 167–178, Available from <citeseer.ist.psu.edu/cohen98automatic.html>.

Collberg, C., Thomborson, C., Low, D., July 1997. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland. Available from <citeseer.ist.psu.edu/collberg97taxonomy.html>.

Collberg, C., Thomborson, C., Low, D., 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In: Proceedings of 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98). Springer-Verlag, New York, Inc., pp. 184–196.

Collberg, C.S., Thomborson, C., 2002. Watermarking, tamper-proofing, and obfuscation—tools for software protection. IEEE Transactions on Software Engineering 28 (8), 735–746.

Dahm, M., van Zyl, J., Haase, E., 2003. Byte code engineering library. Available from <http://jakarta.apache.org/bcel/>.

de Roo, A., van den Oord, L., 2003. Stealthy obfuscation techniques: mislead-ing the pirates. Tech. Rep., Department of Computer Science, University of Twente.

Eastridge Technology, 2004. Jshrink. Available from <http://www.e-t.com/jshrink.html>.

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

Helseth Digital Systems Inc., 2001. Jobfuscator. Available from <http://www.helseth.com/HJO.htm>.

Hoenicke, J., 2002. Java optimize and decompile environment. Available from <http://jode.sourceforge.net/>.

Laffra, C., Lorch, D., Streeter, D., Tip, F., Field, J., 2000. Jikes byte-code toolkit. Available from <http://www.alphaworks.ibm.com/tech/jikesbt>.

Lafortune, E., 2004. Proguard. Available from <http://proguard.sourceforge.net/>.

McAfee Research, 2003. Java binary enhancement tool. Available from <http://opensource.nailabs.com/jbet/>.

Miecznikowski, J., Hendren, L., 2001. Decompiling java using staged encapsu-lation. In: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01). IEEE Computer Society, pp. 368–374.

Miecznikowski, J., Hendren, L., 2002. Decompiling java bytecode: problems, traps and pitfalls. In: Compiler Construction, 11th International Conference, vol. 2304 of Lecture Notes in Computer Science. Springer-Verlag, New York, Inc., pp. 111–127.

Zelix Pty Ltd, 2004. Klassmaster. Available from <http://www.zelix.com/klassmaster/>.