Protecting Java Code via Code Obfuscation

by *Douglas Low*

Abstract

The Java language is compiled into a platform independent bytecode format. Much of the information contained in the original source code remains in the bytecode, thus decompilation is easy. We will examine how code obfuscation can help protect Java bytecodes.

Introduction

Traditionally it has been difficult to reverse engineer [13] applications because they are large, monolithic and distributed as "stripped" object code. Stripping object code of its symbol table removes information like variable names and obscures references to library routines. For example, a call to the C language library routine `printf` in the source code might appear in the stripped object code as a procedure call to the memory address 35720.

Since the advent of Java [7], the threat of reverse engineering has become more serious. The language is compiled into a platform independent bytecode format. Being portable, there is little control over the distribution of the bytecodes. Also, much of the information contained in the source code remains in the bytecode, facilitating decompilation [17 , 19]. The threat of reverse engineering is thus intensified.

Protection Techniques

One possible way to prevent reverse engineering of source code is not to allow physical access to the program. Instead, users communicate with the program via an interface with a limited number of services. This is the **client-server** model [14]. Unfortunately, this imposes performance penalties because of limitations on network bandwidth and latency. A partial solution is to keep the parts of the program that need to be hidden on the server and have the user's machine run the rest locally.

Encryption of code is another possibility. However, unless the entire encryption/decryption process takes place in hardware, it is possible for the user to intercept and decrypt the code [8, 18]. Unfortunately, specialized hardware tends to limit the portability of programs.

Transmitting programs in a form less vulnerable to decompilation might seem like a good idea. Native object codes can be supplied instead of Java bytecodes. The task of decompilation is made more difficult,

although not impossible [4]. However native object codes are not subject to bytecode verification, which gives Java a measure of protection against malicious programs such as viruses [3]. Digital signatures [15], verifying that the native code is actually from a trusted source and has not been tampered with, can help to alleviate this problem. The downside is that different versions of the program are required for different architectures. This means that the software maintenance effort is increased. With Java, only one version is required, since it is executed by a virtual machine.

If we cannot make reverse engineering impossible, we can at least make the task costly in terms of time and effort. Code obfuscation transforms a program so that it is more difficult to understand, yet is functionally identical to the original [5, 6]. The program must still produce the same results, although it may execute slower or have additional side effects because of the added code. There is a trade-off between the security provided by code obfuscation and the execution time-space penalty imposed on the transformed program.

Code Obfuscation

The current work on Java obfuscation has been in the form of freeware, shareware and commercial programs, rather than academic publications. Some of the code obfuscation techniques discussed below are based on traditional compiler optimizations. Examples include array and loop reordering, and procedure inlining [2].

We classify code obfuscation according to *what* kind of information they target and *how* they affect their target.

Layout Obfuscation

Information that is unnecessary to the execution of the program, such as identifier names and comments, is altered. There are many utilities (such as [10, 16]) that will change the identifiers in a program to less meaningful ones. Identifier scrambling is a common obfuscation that has been applied to other languages. The C Shroud system [9], a source code obfuscator available for the C language, is an example of such a tool.

Data Obfuscation

These methods affect the data structures used by a program.

Data storage obfuscation affects how data is stored in memory. For example a local variable can be converted into a global one. **Data encoding** obfuscation affect how the stored data is interpreted. For example, replacing an integer variable `i` by `8 * i + 3` . We can see the effect below:

Before	After

int i = 1;
while (i < 1000) {
... A[I] ...;
i ++;
}
	int i=11;
	while (i < 8003) {
	... A[(i-3)/8] ...;
	i += 8;
	}

Data aggregation obfuscation alters how data is grouped together. For example, a two-dimensional array can be converted into a one-dimensional array and vice-versa.

Data ordering obfuscation changes how data is ordered. For example, an array used to store a list of integers usually has the `i`th element in the list at position `i` in the array. Instead, we could use a function `f(i)` to determine the position of the `i`th element in the list.

Control Obfuscation

The idea here is to disguise the real control flow in a program.

Control aggregation obfuscation changes the way in which program statements are grouped together. For example, it is possible to *inline* procedures. That is, replacing a procedure call with the statements from the called procedure itself.

Control ordering obfuscation alters the order in which statements are executed. For example, loops can be made to iterate backwards instead of forwards.

Control computation obfuscation affects the control flow in a program. These can be divided up further:

- **"Smoke and mirrors"** obfuscation hides the real control flow behind irrelevant statements. For example, code that will never be executed can be inserted (dead code).
- **High-level language breaking** obfuscation introduces features at the object code level that have no direct source code equivalent. Java has no `goto` statement, and thus cannot represent non-reducible control flow graphs [1]. However there is a Java bytecode `goto` instruction, so Java bytecodes can represent non-reducible control flow graphs. For example, inserting a jump into the middle of a `while-loop` creates a non-reducible control flow graph that cannot be transformed back into a standard `while-loop` without difficulty.
- **Control flow abstraction** is essentially the inverse of the task that a compiler performs. An example of altering control flow abstraction would be to find a sequence of low-level instructions that are equivalent to a `while-loop` and then add redundant termination conditions to the loop. The loops both terminate when `i` equal 1000. `i` is not divisible by 1000 for all values that `i` takes

on in the loop, namely for 1 <= `i` < 1000.

Before

```
int i = 1;
while (i < 1000) {
...
i ++;
}
```

After

```
int i = 1;
while ((i < 1000) || (i % 1000 ==
0)) {
...
i ++;
}
```

Preventative Transformations

These attempt to stop decompilers from operating, by exploiting their weaknesses. *HoseMocha* [11] is a utility which appends extra instructions after a `return` instruction. The execution of the program is unaffected but the obfuscation causes the Java decompiler *Mocha* [17] to crash.

Conclusion

The task of making reverse engineering costly is difficult. Client-server models of protection, while providing the best security, suffer from limitations on network capacity. Encryption requires the use of specialized hardware, in turn limiting the portability of programs. Using native object codes makes reverse engineering harder but increases the software support effort. Also, digital signatures are required to prevent tampering. Code obfuscation, while not providing absolute security, is portable, does not require specialized hardware and is transparent to the Java bytecode verifier. However, it does impose an execution time-space penalty on the program being protected.

Code obfuscation is a fruitful area for further research. There are many issues and implications, both theoretical and practical [5], that remain to be resolved.

Acknowledgements

My Masters Thesis research into Java obfuscation has been performed jointly with my supervisors Dr. Christian Collberg and Professor Clark Thomborson.

Glossary

Reverse engineering

A process to obtain information about a program. This includes obtaining source code from compiled programs.

Native object codes

Binary files designed to execute directly on a specific computer platform.

References

1

Aho, Alfred V., Ravi Sethi and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*, Code Optimization chapter, pp. 606-608. Addison-Wesley, 1986.

2

Bacon, David F., Susan L. Graham and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, vol. 26, no. 4, pp. 345 - 420, December, 1994. http://www.acm.org/pubs/toc/Abstracts/0360-0300/197406.html

3

Bank, Joseph A. Java Security. http://swissnet.ai.mit.edu/~jbank/javapaper/javapaper.html , December, 1995.

4

Cifuentes, Cristina and K. John Gough. Decompilation of binary programs. *Software - Practice and Experience*, vol. 25, no. 7, pp. 811-829, July, 1995.

5

Collberg, Christian, Clark Thomborson and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, University of Auckland, New Zealand, July, 1997. http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html

6

Collberg, Christian, Clark Thomborson and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. Department of Computer Science, University of Auckland, New Zealand, July, 1997. http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborsonLow98a/index.html . To appear in ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98).

7

Gosling, James, Bill Joy and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

8

Herzberg, Amir and Shlomit S. Pinter. Public protection of software. *ACM Transactions on Computer Systems*, vol. 5, no. 4, pp. 371-393, November, 1987.

9

Jaeschke, Rex. Encrypting C source for distribution. *Journal of C Language Translation*, vol. 2, no. 1, 1990. http://jclt.iecc.com

10

Jokipii, Eron. Jobe - The Java obfuscator. http://www.primenet.com/~ej/index.html, 1996.

11

LaDue, Mark D. HoseMocha. http://www.oasis.leo.org/java/development/bytecode/obfuscators/HoseMocha.dsc.html

12

Lindholm, Tim and Frank Yellin. The Java Virtual Machine Specification. http://java.sun.com:80/docs/books/vmspec/html/VMSpecTOC.doc.html, September, 1996.

13

Marciniak, John J., editor. *Encyclopedia of Software Engineering*, Reverse Engineering chapter, pp. 1077-1084. John Wiley & Sons, Inc, 1994.

14

Sape Mullender, editor. *Distributed Systems* . Addison-Wesley, 2nd edition, 1993.

15

Rivest, R., A. Shamir and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*. pp. 120-126. February, 1978.

16

van Vliet, Hans Peter. Crema - The Java obfuscator. http://java.cern.ch/Java/java/CremaE1/DOC/Index.html, January, 1996.

17

van Vliet, Hans Peter. Mocha - The Java decompiler. http://java.cern.ch/Java/java/MochaB1/Readme.txt, January, 1996.

18

Wilhelm, Uwe G. Cryptographically protected objects. http://lsewww.epfl.ch/~wilhelm/CryPO.html, May, 1997. A french version appeared in the Proceedings of RenPar'9, Lausanne.

19

WingSoft Company. JavaDis - The Java Decompiler. http://www.wingsoft.com/wingdis.shtml, March 1997.

Douglas Low is currently a graduate student at the University of Auckland, New Zealand, working on the application of code obfuscation to Java. His research interests include compiler implementation, computational combinatorics and constraint satisfaction (AI). Electronic versions of papers that he has co-authored can be found at http://www.cs.auckland.ac.nz/~collberg/Research/Students/DouglasLow .