Java Control Flow Obfuscation

Douglas Low

A thesis submitted in partial fulfilment of the
requirements for the degree of
Master of Science in Computer Science

University of Auckland
June 3, 1998

Abstract

The language Java was designed to be compiled into a platform independent bytecode format. Much of the information contained in the source code remains in the bytecode, which means that decompilation is easier than with traditional native codes. As a result, software developers are taking seriously the threat of competitors using reverse-engineering to extract proprietary algorithms from compiled Java programs.

We examine several technical protection techniques that could be used to hinder the reverse-engineering of software. We claim that *code obfuscation* is the most suitable technical protection technique that can be applied to a portable language like Java.

The technique of code obfuscation involves applying *obfuscating transformations* to a program. These transformations make the program more difficult for a reverse-engineer to understand but do not affect the functionality of the program. We focus on a particular category of obfuscating transformations — *control flow obfuscation*. Control flow obfuscations disguise the algorithms used by a program by introducing new fake control flows, creating features at the object code level which have no source code equivalent or altering the way in which statements are grouped.

There are many practical aspects to be considered when applying obfuscating transformations to Java programs. The fact that Java programs are portable and are verified before execution makes obfuscating transformations more difficult to apply. The verification stage ensures that programs do not perform illegal operations, such as corrupting a user's system.

Obfuscating transformations can be applied automatically to a program by a tool called an obfuscator. In this thesis, we present one possible method of implementing such an obfuscator. We first discuss the design decisions made to address implementation problems. Then, we use the obfuscator on examples of Java code and examine how effective the obfuscating transformations are in impeding reverse-engineering.

Acknowledgements

I would like to thank my supervisors Dr. Christian Collberg and Professor Clark Thomborson for their ideas and support. This thesis grew out of the joint work performed with my supervisors, which was presented at the conferences POPL'98 [11] and ICCL'98 [10].

Gordon and Jarno deserve mention for helping with the LaTeX typesetting system. Along with the rest of the people in the office, they provided me with a stimulating work environment.

I owe my family for their encouragement and support, without whose help it would have been a far more difficult task to complete my thesis.

Contents

Abstract **i**

Acknowledgements **iii**

Contents **v**

Terminology **ix**

1 Introduction **1**
1.1 Reverse-Engineering 1
1.2 Reverse-Engineering Software 2
1.3 The Threat to Java 2
1.4 Related Work 3
1.5 Organisation of this Thesis 3

2 Technical Protection Techniques **5**
2.1 Introduction 5
2.2 Protection by Server-Side Execution 6
2.3 Protection by Encryption 7
2.4 Protection through Signed Native Code 8
2.5 Protection through Code Obfuscation 9
2.6 Discussion 10
2.7 Summary 11

3 Code Obfuscation **13**
3.1 Introduction 13
3.2 Obfuscating Transformations 13
3.3 Classification 14
3.4 Evaluation 17
3.5 Using Obfuscation Measures 24
3.6 Discussion 26

3.7 Summary 26

4 Control Flow Obfuscation 29
4.1 Introduction 29
4.2 Opaque Constructs 29
4.3 Computation Obfuscations 41
4.4 Aggregation Obfuscations 44
4.5 Ordering Obfuscations 46
4.6 Discussion 47
4.7 Summary 48

5 Java Obfuscation in Practice 49
5.1 Introduction 49
5.2 Executing Java Code 49
5.3 Java Class File Format 53
5.4 The Java Bytecode Instruction Set 55
5.5 Java Bytecode Verifier 60
5.6 Unusual Bytecode Features 61
5.7 Discussion 66
5.8 Summary 69

6 Overview of the Java Obfuscator 71
6.1 Introduction 71
6.2 Using the Java Obfuscator 71
6.3 Implementation Language 76
6.4 Obfuscator Structure 78
6.5 Discussion 78
6.6 Summary 80

7 Implementation of the Java Obfuscator 81
7.1 Introduction 81
7.2 Source Code Objects 81
7.3 Parsing the Java Class Files 83
7.4 Building an Internal Representation 83
7.5 Constructing Mappings 85
7.6 Applying Obfuscating Transformations 91
7.7 Re-constituting the Application 94
7.8 Discussion 94
7.9 Summary 95

8 Examples 97
8.1 Introduction 97
8.2 Change to Subroutine 98

8.3 Insert Dead Code 101
8.4 Non-standard Code Patterns 106
8.5 Sample Obfuscator Output 109

9 Conclusion **117**
9.1 The Cost of Obfuscation 117
9.2 Future Work 118
9.3 Summary 119

A Obfuscation Modules **121**
A.1 The Insert Bogus Branch Transformation 122

B The Opaque Predicate Library **125**
B.1 Selecting an Opaque Predicate 125
B.2 Opaque Predicate Attributes 125
B.3 Inserting an Opaque Predicate 126
B.4 The Simple Opaque Predicates Library 127
B.5 The Graph Opaque Predicates Library 128

C The Set Abstract Data Type **131**

Terminology

Obfuscation Section 2.5

The process by which a program is made more difficult to understand, so that it is more resistant to reverse-engineering.

Obfuscator Section 2.5

A tool designed to apply obfuscating transformations to a program.

Deobfuscator Section 2.5

A tool designed to reverse obfuscating transformations that are applied to a program.

Obfuscating transformation Section 3.2

A transformation which preserves the functionality of a program but makes the program more difficult to understand.

Layout Obfuscation Section 3.3.1

An obfuscation which affects the information in a program that is unnecessary for its execution.

Data Obfuscation Section 3.3.2

An obfuscation which operates on the data structures used in a program.

Control Flow Obfuscation Section 3.3.3

An obfuscation which affects the flow of execution in a program.

Preventive Transformation Section 3.3.4

A transformation that is designed to prevent decompilers and deobfuscators from operating correctly.

Potency Section 3.4.1

The degree to which an obfuscating transformation confuses a human who is trying to understand an obfuscated program.

$\mathcal{T}_{\mathbf{pot}}(P)$ Section 3.4.1
The potency of an obfuscating transformation $\mathcal{T}$ when it is applied to a program P.

Resilience Section 3.4.2
The effort required to undo the effects of an obfuscating transformation.

$\mathcal{T}_{\mathbf{res}}(P)$ Section 3.4.2
The resilience of an obfuscating transformation $\mathcal{T}$ when it is applied to a source code object S.

Cost Section 3.4.3
The additional run-time resources required to execute a program after an obfuscating transformation has been applied to the program.

$\mathcal{T}_{\mathbf{cost}}(P)$ Section 3.4.3
The cost of an obfuscating transformation $\mathcal{T}$ when it is applied to a program P.

Stealth Section 3.4.4
The degree to which the code added by an obfuscating transformation differs from the code in the original program.

$\mathcal{T}_{\mathbf{ste}}(P)$ Section 3.4.4
The stealth of an obfuscating transformation $\mathcal{T}$ when it is applied to a program P.

Quality Section 3.4.5
The combination of the potency, resilience, cost and stealth of an obfuscating transformation.

$\mathcal{T}_{\mathbf{qual}}(P)$ Section 3.4.5
The quality of an obfuscating transformation $\mathcal{T}$ when it is applied to a program P.

Appropriateness Section 3.6
A variant of transformation quality which is used to select an obfuscating transformation to apply to a program.

$\mathcal{T}_{\mathbf{app}}(P)$ Section 3.6
The appropriateness of an obfuscating transformation $\mathcal{T}$ when it is applied to a program P.

Opaque predicate Section 4.2
A boolean valued expression whose value is known to an obfuscator but is difficult for a deobfuscator to deduce.

Opaque variable Section 4.2

A variable whose value is known to an obfuscator but is difficult for a deobfuscator to deduce.

Source code object Section 7.1

A part of a program to which obfuscations can be applied. For an object-oriented language source code objects include classes, methods and fields.

Language features Section 7.5.1

Examples of language features include operators, variables, procedure calls to particular routines and whether a program has single or multiple threads of execution.

$P_s(S)$ Section 7.5.1

The language features of the source code object S.

$A(S)$ Section 7.5.2

The transformation stealth mapping for a source code object S. For each transformation $\mathcal{T}$, this mapping provides the stealth of $\mathcal{T}$ when it is applied to S.

Obfuscation priority Section 7.5.3

How important it is to obfuscate a particular source code object.

$I(S)$ Section 7.5.3

The obfuscation priority of the source code object S.

CHAPTER 1

Introduction

"From small beginnings come great things."

– Proverb

1.1 Reverse-Engineering

An extreme case of reverse-engineering is taking someone else's product apart to determine how it functions (see Marciniak [35], pp. 1077–1084). More generally, reverse-engineering involves analysing an existing system, regardless of ownership. By itself, reverse-engineering is not illegal, and in fact it is commonly used to improve one's own products. Reverse-engineering can reveal design flaws that are not otherwise obvious and detect redundancies in a system, which can then be removed.

In the field of engineering, reverse-engineering takes a finished product and recreates the knowledge needed to reproduce it. Manufacturing processes and tools are non-trivial to recreate, which contrasts with reproducing software — all that is required is to copy the bytes that make up the program. Instead, the difficulty with reverse-engineering software is in reconstructing enough knowledge about a program to be able to modify it, reuse parts of it or interface to it.

The fact that it is so easy to reproduce software as compared to manufactured goods means that a greater reliance must be placed on legal protection of copyrights.

1.2 Reverse-Engineering Software

To reverse-engineer a software application it is necessary to first gain physical access to it. Using disassemblers or decompilers [8], the reverse-engineer can decompile it to source code, then analyse its control flow and data structures. Additional tools such as program slicers may be used to perform this analysis (see Marciniak [35], pp. 873–877).

One of the most important legitimate uses of software reverse-engineering is recovering lost information about a system. Businesses are still using original COBOL language programs for the sake of being able to access old records. The source code for these programs may no longer be available in a readable form, perhaps because they are stored in obsolete systems such as punch cards. Hence a major reverse-engineering effort is needed to recover source code from the compiled COBOL programs, such as warehouse inventory control systems. These programs need to be modified in order to eliminate the so-called *Year 2000 problem*.

Wholesale reverse-engineering of applications violates copyright law [44] but this is not what worries developers. By extracting proprietary algorithms and data structures from a rival's application, and incorporating these items into their own, developers can dramatically reduce their software development cost and time. It is more difficult to prove that parts of an application have been copied, as opposed to the entire application. Hence obtaining compensation for lost revenue through the legal system is more difficult. Small developers may not be able to afford long legal battles against large corporations with substantial resources [34].

Traditionally it has been difficult to reverse-engineer 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. Thus, the job of a reverse-engineer is made harder as she must perform housekeeping tasks such as memory location to variable mapping, that are unnecessary with code that has not been stripped.

1.3 The Threat to Java

Since the advent of Java [18], the threat of reverse-engineering is being taken more seriously. The language was designed to be compiled into a platform independent bytecode format. A lot of the information contained in the source code remains in the bytecode, making decompilation easier [41]. Heavy use is made of standard library routines, so applications tend to be small, which aids reverse-engineering. All of these factors intensify the threat of reverse-engineering to

developers writing applications in Java.

We will argue that the only feasible form of technical protection of mobile code such as Java bytecodes is code obfuscation [9]. We will examine how this technique can be applied to Java bytecodes in practice and how effectively they impede malicious reverse-engineering attacks.

1.4 Related Work

Existing Java obfuscators like `Crema` [50] and `Jobe` [29] change the names of identifiers in Java programs to less meaningful ones. This is a form of *layout transformation* (Section 3.3.1). These kinds of transformations make the Java program more difficult to understand but do not hide control flow. In addition to identifier scrambling, the Java obfuscator written by Aggarwal [1] performs the layout transformation of removing debugging information.

Obfuscators have been written for languages other than Java. The `C Shroud system` is a source code obfuscator for the language C. It performs layout transformations like removing comments and indentation, and scrambling identifiers. Also, it converts the control structures `for`, `while`, `do/while`, `if/else` and `switch` into `if/goto` structures. Changing the control structures helps to disguise the control flow of the program (Section 3.3.3).

On a personal computer (PC), software is completely accessible for observation and modification by the user. This is a similar problem with which Java bytecodes are faced. The issue of protecting software on PC platforms has been addressed by Aucsmith [3]. This scheme uses a self-modifying, self-decrypting and installation unique segment of code (Integrity Verification Kernel). The code segment communicates with other such code segments to create an Interlocking Trust model. The code segments verify each others integrity. Unfortunately this technique cannot be applied directly to Java, since it requires the ability to access and dynamically alter code (for decryption and encryption) while the program is executing.

1.5 Organisation of this Thesis

In this chapter we have outlined the threat reverse-engineering poses to software developers. Legal protection in the form of copyright laws is insufficient for small software developers, who do not have the resources available to fight lengthy legal battles. We assert that the only feasible form of technical protection for mobile codes such as Java is code obfuscation. The remaining chapters are arranged as follows:

Chapter 2 compares and contrasts several technical protection techniques for software: client-server models, encryption, signed native code and code obfuscation.

Chapter 3 examines the technical protection technique of code obfuscation in detail. We discuss the obfuscating transformation classification and evaluation scheme which is presented in Collberg et al. [9].

Chapter 4 focuses on control flow obfuscation, a particular category of code obfuscation which hides the real control flow in a program. Many control flow obfuscations rely on the existence of opaque constructs. We define what opaque constructs are and present some methods to create such opaque constructs.

Chapter 5 addresses the issues involved in applying code obfuscation to Java bytecodes. We examine the Java run-time environment, the class file format, the bytecode instruction set and verifier.

Chapter 6 provides an overview of the architecture of our Java obfuscator. We describe the options provided by our Java obfuscator and examine the advantages and disadvantages of the implementation language. We also provide a high-level description of the steps involved in obfuscating a program.

Chapter 7 presents detail about the actual implementation of our Java obfuscator. We describe the algorithms used, the various design decisions made and the problems encountered.

Chapter 8 evaluates our obfuscator on concrete examples of Java code. We gauge the effectiveness of the transformations used by examining the effect they have on selected software complexity metrics.

Chapter 9 discusses the cost of obfuscation, deobfuscation techniques and possible directions for future research.

Appendix B explains the structure of a predicate library. It presents the simple and graph predicate libraries as examples.

Appendix A explains the structure of an obfuscating transformation module. It presents the insert bogus branch transformation as an example.

Appendix C presents the Java source code for the `Set` abstract data type (ADT). The `Graph` ADT, which is defined in Chapter 4, uses the `Set` ADT.

CHAPTER 2

Technical Protection Techniques

"A lock only ever stopped an honest man."

– Ancient Egyptian Proverb

2.1 Introduction

The idea behind the technical protection of software secrets is to make reverse-engineering economically impractical, if not impossible. Some early attempts at technical protection are described in Suhler et al. [47] and Gosler [17]. However, all the techniques discussed by these authors rely on system specific hardware features. A common technique in the 1980's was to distribute software on specially formatted floppy disks, which standard system file copy routines fail to duplicate correctly. The point of platform independent codes such as Java bytecodes is that only one version of an application needs to be written, which can be executed on "any" platform. Development and support costs will be reduced since there is no need to maintain different versions of the application for different platforms. Making the software rely on special hardware features defeats the purpose of using a portable language. Thus, technical protection must be incorporated into software and be hardware independent if it is to be of any use in a mobile code environment. We discuss the technical protection techniques of client-server models, encryption, signed native code and code obfuscation, as described in Collberg et al. [9].

2.2 Protection by Server-Side Execution

The first step in reverse-engineering an application involves gaining physical access to the code. By preventing physical access to the application, the problem of reverse-engineering is nipped in the bud.

Much research has been performed in the field of client-server mechanisms, for example *Distributed Systems* [37]. In Figure 2.1(a), the application that requires protection is placed on a server and its *services* are provided to users over a remote connection. Thus, access to the application code is prevented. There are two major disadvantages of full server-side execution as compared to an application running entirely on a local machine:

- Network bandwidth and latency are limited, so the performance of the application is decreased because of communication overhead.
- If the network fails to operate correctly, the user will be unable to use the application.

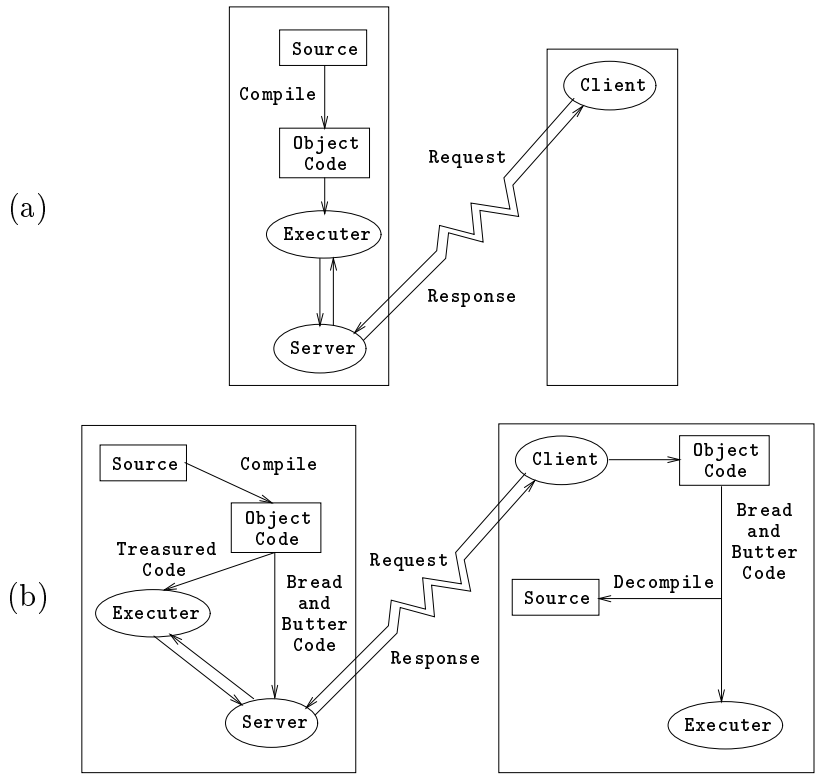


Figure 2.1. *Protection by (a) Server-side and (b) Partial Server-side execution.*

Only some parts of the application may be regarded as proprietary by the developer. It may thus be unnecessary to protect the entire application. The rest

of the application merely uses these proprietary parts and is of no real interest to a competitor, so-called "bread-and-butter-code". In this case full server-side execution of the whole application is an excessive amount of protection. Instead, the application can be broken into a private part, which executes on the server, and a public part, which runs locally on the user's site. Figure 2.1(b) shows how this scheme operates.

Partial server-side execution enjoys the benefits of full server-side execution, in that the proprietary parts of the application are not available for inspection by the user. Care must be taken in separating the server-side and client-side parts of the application to avoid frequent communication.

2.3 Protection by Encryption

In order to eliminate the performance penalties associated with the client-server models of execution mentioned in the previous section, we must examine methods that protect code that is executed wholly on a client machine. Encrypting the distributed code is one method of defeating decompilation (Figure 2.2). Unless decryption takes place in hardware, it will be possible to intercept and decrypt compiled code. Hardware decryption systems have been described in Herzberg and Pinter [22] and Wilhelm [52]. The idea is to have a co-processor (cryptochip) which decrypts instructions before they are executed by the main processor. The decrypted code is never stored in user accessible memory, so the degree of security depends on the scheme used to encrypt the code. Additionally, software can be designed to execute only if the correct cryptochip is present in a system. This means that to produce copies of the software that will execute on any machine, a reverse-engineer must break the encryption scheme and alter the code so that it does not require the presence of a cryptochip.

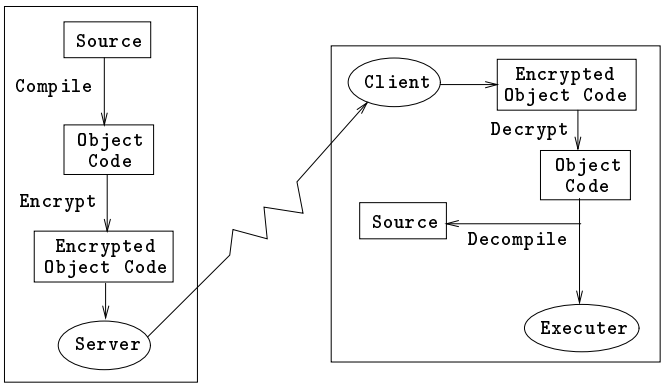


Figure 2.2. *Protection by Encryption.*

The problem with the cryptochip system is that different kinds of processors need different interfacing circuitry to communicate with the cryptochip. Encryption performed in hardware is thus unsuitable as a software protection technique if there is a wide range of platforms on which an application must execute.

Adding a cryptochip to a system would force end-users to pay extra money, just so that software developers can protect their products. This is unlikely to be popular amongst end-users, so it is doubtful that hardware encryption will be employed widely as a software protection method.

2.4 Protection through Signed Native Code

Java bytecodes are executed by a Java virtual machine, which is normally implemented by an interpreter, rather than being directly executed by the processor. This means that Java programs execute an order of magnitude slower than traditional compiled languages like C programs. However, Java bytecodes are portable, while compiled C programs are limited to a particular operating system. To address the performance issue of Java programs, *just-in-time* (JIT) compilers have been developed to translate Java bytecode into native code at run-time. The native code is then executed in lieu of the original bytecodes. There are several JIT compilers available for a variety of operating systems. Several companies have included JIT compilers in their Java development kits, for example, Borland's `JBuilder` [26], Symantec's `Café` [14] and Microsoft's `Visual J++` [27].

To avoid incurring the overhead of compilation every time Java bytecodes are executed, it is possible to store the compiled native codes. This is known as *way-ahead-of-time* (WAT) compilation. Several WAT compilers are under development, such as `Toba` [40] and `NET` [25].

We can use JIT or WAT compilers to transform Java bytecodes into native codes, which are more difficult to decompile. The program requiring distribution is first compiled into Java bytecodes and stored on a server. Users identify their architecture/operating system combination to the server, which then provides the appropriate native code version of the application.

Only having access to native code hinders but does not stop reverse-engineering. For example, there are decompilers to translate 80x86 machine code into C source code [8]. So using native codes for distribution does not provide the same level of security as hardware encryption devices. Additionally, a JIT or WAT compiler is required to translate the Java bytecodes into the native code of each architecture/operating system that is likely to be encountered. This means that the portability of the application is reduced to the systems that the server supports.

There is a further problem with transmitting only native code. Unlike Java bytecodes, native code is not subject to *bytecode verification* before execution [32]. So native code cannot be run with a guarantee that it will execute without performing illegal actions such as corrupting a user's files [5]. This is not a problem

if the developer is a trusted member of the community — the user may accept assurances by the developer that the application is safe. To avoid tampering, the developer has to digitally sign [43] the code as it is transmitted, proving to the user that the code is the original one provided by the developer (Figure 2.3).

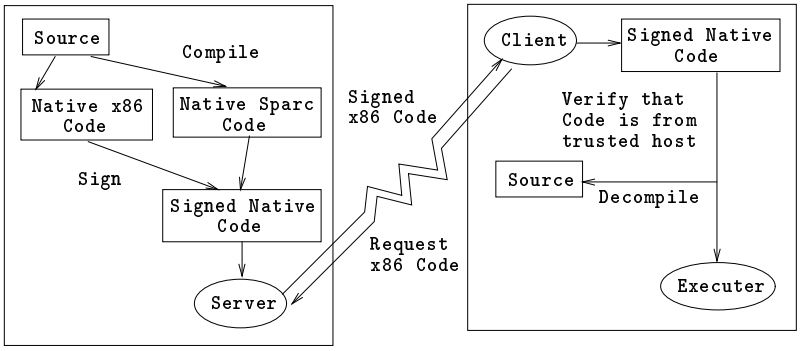


Figure 2.3. *Protection through Signed Native Code.*

2.5 Protection through Code Obfuscation

The final idea that we will consider is code obfuscation. The idea is to transform an application so that it is functionally identical to the original but is much more difficult to understand. The software developer uses a utility called an *obfuscator* to protect her application (Figure 2.4). However, unlike server-side execution, code obfuscation cannot fully protect an application against a malicious reverse-engineering attack. Given enough time and effort, it will be possible to retrieve important algorithms and data structures. A utility called a *deobfuscator* may be used to undo any transformations applied to the application.

The level of security provided by an obfuscator against reverse-engineering depends on:

1. The sophistication of the transformations used by the obfuscator.
2. The power of the deobfuscation algorithms.
3. The amount of resources (time and space) available to the deobfuscator.

Code obfuscation makes code more difficult to understand, yet preserves platform independence. It would be pointless to make a transformed application rely on system specific hardware features if it is written in a portable language. Unlike server-side execution, an obfuscated application does not suffer from delays due to network limitations. It does not require the hardware needed for secure encryption and decryption of code. The Java bytecode verifier guarantees that a

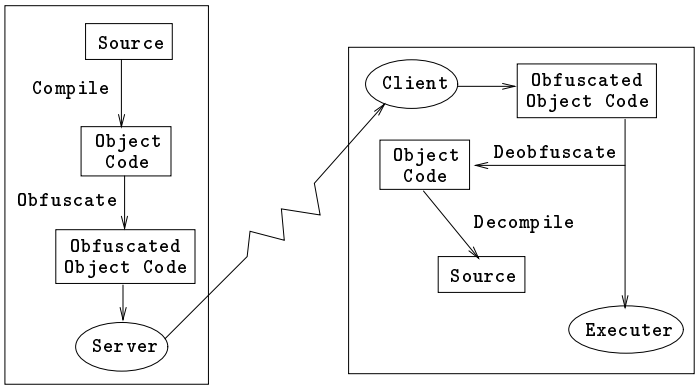


Figure 2.4. *Protection through Code Obfuscation.*

Java application will not perform illegal actions such as erasing a user's files. So unlike native codes, there is no need to digitally sign a Java application to verify that it is "safe" code from a trusted source.

2.6 Discussion

We claim that code obfuscation is superior to the other three techniques presented in this chapter, when protecting *portable mobile* programs. There may be situations where the other techniques are better, such as when the source code contains extremely valuable trade secrets, or when run-time performance is critical. The transformations employed by an obfuscator may not provide a high enough level of protection, or have adverse effects on performance.

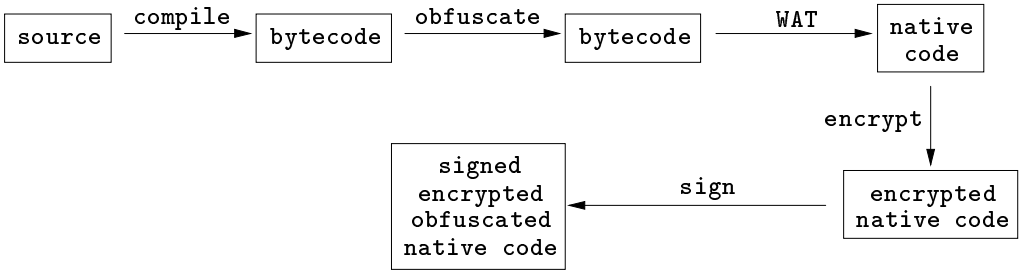
The server-side execution model suffers from network bandwidth and latency limitations. While partial server-side execution reduces delays due to limited network performance, it does not eliminate them. We must therefore examine techniques that protect code that executes wholly on a client machine.

Encryption fails to be effective without specialised hardware and specialised hardware places restrictions on portability. This is unacceptable for a language like Java, which is meant to be executed on a variety of hardware platforms. The use of specialised hardware also incurs higher cost to the end-user. So we must look at hardware-independent means of protecting client executed applications.

The use of signed native code also places restrictions on portability, since it depends on the server to provide Java bytecode to native code translators for many architectures. The server would probably only support the most common architectures. Decompilation of native code is still possible, although it is more difficult to achieve than with Java bytecodes. We must therefore try to make it more difficult to understand any source code that is obtained by decompilation.

Code obfuscation attempts to make decompiled source code more difficult to understand. However, given enough time and effort, it is possible to retrieve important algorithms and data structures from such obfuscated code. The aim is to increase the time and effort required so that it is economically infeasible for a company to reverse-engineer a rival's application.

However, these protection techniques are not mutually exclusive. Code obfuscation can be applied to bytecodes, which are JIT compiled to native code. We can then encrypt and digitally sign this native code.



Reverse-engineering the resulting code is more difficult, since there are many more steps to be undone to recover the source code.

2.7 Summary

In this chapter, we have discussed several technical code protection techniques. We examined their advantages and disadvantages, and concluded that code obfuscation is the most suitable method for protecting portable codes like Java. In the following chapter we will examine code obfuscation in more detail. We will distinguish different categories of code obfuscation and will provide an evaluation scheme to determine the quality of a code obfuscation.

CHAPTER 3

Code Obfuscation

"You can fool all of the people some of the time, and some of the people all of the time, but you can't fool all of the people all of the time."
– attributed to Abraham Lincoln

3.1 Introduction

Code obfuscation does not provide an application with absolute protection against a malicious reverse-engineering attack. The level of security depends on the sophistication of the transformations used by the obfuscator and the power, tools and resources available to the reverse-engineer.

Obfuscating transformations are the basis of code obfuscation. In this chapter, we discuss the obfuscating transformation classification and evaluation scheme which is presented in Collberg et al. [9].

3.2 Obfuscating Transformations

An *obfuscator* is a program used to transform program code. The output of an obfuscator is program code that is more difficult to understand but is functionally equivalent to the original. In order to achieve this, the obfuscator applies *obfuscating transformations* to the program code. Existing obfuscators like `Crema` [50] assume that the original and obfuscated programs must have identical behaviour. In this thesis we assume that it is possible to relax this constraint under certain circumstances. This means that the transformed program can be slower or larger

than the original, or even have side-effects such as creating files. However, the *observable behaviour* (the behaviour as experienced by the user) of the two programs must be identical. Hence, we need to define the notion of an *obfuscating transformation*:

DEFINITION 1 (OBFUSCATING TRANSFORMATION) Let $P \xrightarrow{\mathcal{T}} P'$ be a transformation of a source program P into a target program P'.

$P \xrightarrow{\mathcal{T}} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behaviour*. More precisely, in order for $P \xrightarrow{\mathcal{T}} P'$ to be a legal obfuscating transformation the following conditions must hold:

- 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.

□

3.3 Classification

We classify obfuscating transformations according to *what* kind of information they target and *how* they affect their target. The kinds of information affected can be seen in Figure 3.1.

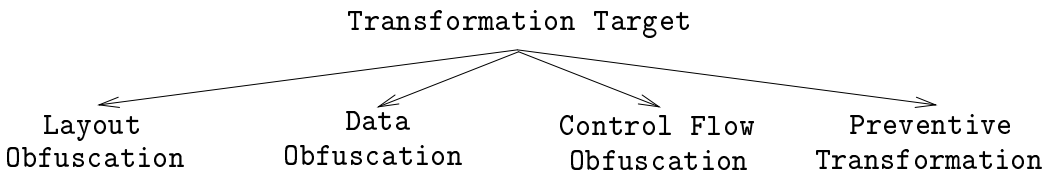


Figure 3.1. *Information targeted by an obfuscating transformation.*

We describe each of the above target categories in the following sections. Note that some of the code obfuscations discussed are traditional compiler optimisations — we just use these optimisations for a different purpose. Array and loop reordering, and procedure inlining are examples of such optimisations [4].

3.3.1 Layout obfuscation

Layout obfuscations affect the information in the program code that is unnecessary to its execution. These obfuscations are typically trivial and reduce the amount of information available to a human reader. Examples include scrambling identifier names and removing comments and debugging information.

3.3.2 Data obfuscation

Data obfuscations operate on the data structures used in the program. We can further classify them according to what operation they perform on the data structures (Figure 3.2).

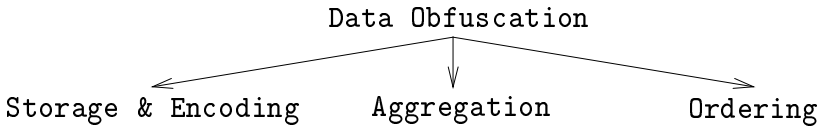


Figure 3.2. *Data obfuscation categories.*

Data storage obfuscations affect how data is stored in memory. An example is converting a local variable into a global one. **Data encoding** obfuscations affect how the stored data is interpreted, for example replacing an integer variable `i` by the expression `8 * i + 3`. Source code would be transformed in the manner of Figure 3.3.

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

Figure 3.3. *An example of a data encoding obfuscation.*

Data aggregation obfuscations alter how data is grouped together. An example is transforming a two-dimensional array into a one-dimensional array and vice-versa.

Data ordering obfuscations change how data is ordered. The normal way in which an array is used to store a list of integers has the ith element in the list at position i in the array. Instead, we could use a function $f(i)$ to determine the position of the ith element in the list.

Data obfuscations will not be discussed further in this thesis. For additional information about data obfuscation, see Bertenshaw [6] and Collberg et al. [10].

3.3.3 Control flow obfuscation

Control flow obfuscations affect the control flow of the program. Again we can classify them further according to what operation they perform (Figure 3.4). Chapter 4 covers control flow obfuscation in detail.

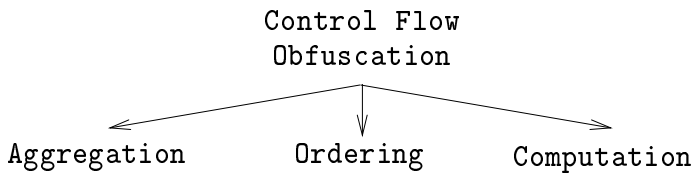


Figure 3.4. *Control flow obfuscation categories.*

Control aggregation obfuscations change 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 obfuscations alter the order in which statements are executed. For example, loops can sometimes be made to iterate backwards instead of forwards.

Control computation obfuscations hide the real control flow in a program. For example, statements which have no effect can be inserted into a program.

3.3.4 Preventive transformation

In contrast to the three preceding techniques, a **preventive transformation** does obscure the program to human readers. Preventive transformations are intended to stop decompilers and deobfuscators from functioning correctly. We can divide preventive transformations into two categories as in Figure 3.5.

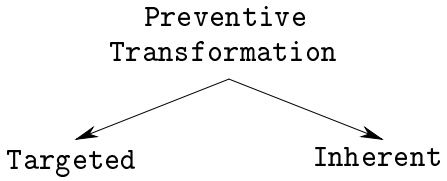


Figure 3.5. *Preventive transformation categories.*

Inherent preventive transformations make known automatic deobfuscation techniques harder to employ, although they add little or no obfuscation to a program. In Figure 3.6, a `for`-loop has been re-ordered to run backwards. An artificial data dependency is added to make it harder for a deobfuscator to undo this control ordering obfuscation.

Targeted preventive transformations are designed to counter specific analysis tools. An example would be the `HoseMocha` [31] program which attacks a weakness in the `Mocha` [51] decompiler. `HoseMocha` inserts extra instructions after the `return` instructions in Java bytecodes. This transformation does not affect

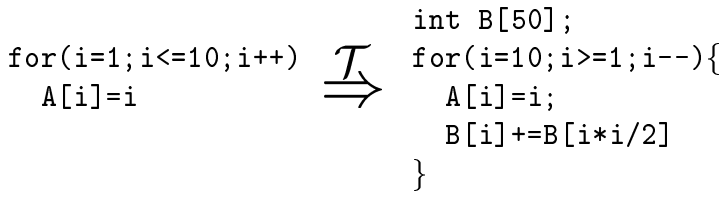


Figure 3.6. *An example of a preventive transformation. The* `for`*-loop has been reordered to run backwards, which is possible since the original loop has no loop-carried dependencies. To prevent an automatic deobfuscator from simply undoing this control ordering obfuscation, we can add a fake data dependency to the reversed loop.*

the behaviour of the application, but it makes `Mocha` crash, thus decompilation is prevented.

3.4 Evaluation

We can measure the effectiveness of obfuscating transformations according to the criteria *potency*, *resilience*, *cost* and *stealth*. In this section we will discuss these measures.

3.4.1 Potency

The concept of one program being harder to understand than another is vague since it is partially based on human perceptions. A large amount of research has been carried out in the *Software Complexity Metrics* branch (see Marciniak [35], pp. 131–163) of Software Engineering concerning this problem. In this field, complexity metrics (see Table 3.1 for some popular ones) have been designed with a view to improving program readability. A software complexity metric usually counts some property of a source code program, with complex programs having a greater amount of this property than simpler programs. For example, the McCabe metric (μ_2 in Table 3.1), which counts the number of decision points in a program, will increase as more `if`-statements are added to a program.

We can use these complexity metrics to derive statements like: "If programs P and P' are identical except that P' contains more of property q than P, then P' is more complex than P." Note that the actual measured values of the metrics are unimportant — it is the relative values of the metrics for P and P' that interest us. That is, we design obfuscations which add more of the property q to a program, since this is likely to increase the complexity of the program. This is our notion of *potency*, which uses complexity metrics as follows:

DEFINITION 2 (TRANSFORMATION POTENCY) Let $\mathcal{T}$ be an obfuscating transformation, such that $P \xrightarrow{\mathcal{T}} P'$ transforms a source program P into a target program P'. Let $E(P)$ be the complexity of P, defined by one of the metrics[1] in Table 3.1.

$\mathcal{T}_{\text{pot}}(P)$, the *potency* of $\mathcal{T}$ with respect to a program P, is a measure of the extent to which $\mathcal{T}$ changes the complexity of P. It is defined as

$$\mathcal{T}_{\text{pot}}(P) \stackrel{\text{def}}{=} E(P')/E(P) - 1.$$

$\mathcal{T}$ is a *potent obfuscating transformation* if $\mathcal{T}_{\text{pot}}(P) > 0$. □

We will measure potency on a three-point scale $\langle$*low*, *medium*, *high*$\rangle$. If $\mathcal{T}_{\text{pot}}(P)$ is a small positive value, then $\mathcal{T}$ is considered to have low potency. Conversely if $\mathcal{T}_{\text{pot}}(P)$ is a large positive value, then $\mathcal{T}$ is said to have high potency. A *potent* obfuscating transformation should increase one or more of the measures in Table 3.1.

3.4.2 Resilience

It may seem that increasing $\mathcal{T}_{\text{pot}}(P)$ is a trivial process. For example, to increase the μ_2 metric (Table 3.1) one need only to add some simple `if`-statements (i.e. predicates) to P:

```
main() {                    main() {
   S1;          ⟹             S1;
   S2;                         if (5==2) S3;
}                              S2;
                               if (1>2) S4;
                            }
```

Figure 3.7. *Trivial predicate insertion.*

Such transformations are virtually useless — they can be easily undone by simple automatic techniques such as peephole optimisations (see Aho et al. [2], pp. 554–558). Constant folding will replace the boolean expression `(5==2)` with `false`. Now it is obvious that S_3 will never be executed, hence this dead code can be eliminated, along with the statement `if (false)` S_3. Thus, we require a concept of *resilience*, which is how well a transformation resists attack from an automatic deobfuscator. The resilience of a transformation $\mathcal{T}$ is a measure of the total effort required to undo $\mathcal{T}$ and is a combination of two measures:

[1] The particular metric (or combination of metrics) to use is not crucial to this definition.

Metric	Metric Name	Citation
μ_1	Program Length	Halstead [19]
	$E(P)$ increases with the number of operators and operands in P.	
μ_2	Cyclomatic Complexity	McCabe [36]
	$E(F)$ increases with the number of predicates in F.	
μ_3	Nesting Complexity	Harrison [20]
	$E(F)$ increases with the nesting level of conditionals in F.	
μ_4	Data Flow Complexity	Oviedo [39]
	$E(F)$ increases with the number of inter-basic block variable references in F.	
μ_5	Fan-in/out Complexity	Henry [21]
	$E(F)$ increases with the number of formal parameters to F, and with the number of global data structures read or updated by F.	
μ_6	Data Structure Complexity	Munson [38]
	$E(P)$ increases with the complexity of the static data structures declared in P: **Data structure** — **Factors increasing complexity** Scalar variable — none Array — dimensions, complexity of element type Record — number and complexity of fields	
μ_7	OO Metric	Chidamber [7]
	$E(C)$ increases with: μ_7^{a} the number of methods in C μ_7^{b} the depth (distance from the root) of C in the inheritance tree μ_7^{c} the number of direct subclasses of C μ_7^{d} the number of other classes to which C is coupled[a] μ_7^{e} the number of methods that can be executed in response to a message sent to an object of C μ_7^{f} the degree to which C's methods do not reference the same set of instance variables Note: μ_7^{f} measures *cohesion*; i.e. how strongly related are the elements of a module	

[a] Two classes are coupled if one uses the methods or instance variables of the other.

Table 3.1. *Overview of some popular software complexity measures. $E(X)$ is the complexity of a software component X, F is a function or method, C is a class, and P is a program.*

Programmer Effort
: The amount of time required to construct an automatic deobfuscator able to effectively reduce the potency of $\mathcal{T}$.

Deobfuscator Effort
: The execution time and memory required by the automatic deobfuscator constructed above to effectively reduce the potency of $\mathcal{T}$.

Typically, a programmer is prepared to spend a set amount of effort to undo an obfuscating transformation $\mathcal{T}$. Constructing a deobfuscator requires $\mathcal{T}$ to be analysed and if it is very difficult to do this analysis, the programmer may give up.

Note that a transformation is *potent* if it manages to confuse a human reader, but it is *resilient* if an automatic deobfuscator is unable to undo the transformation.

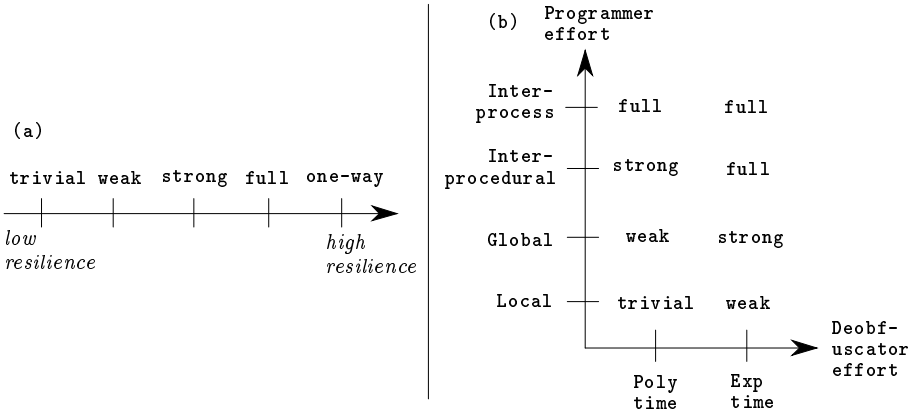


Figure 3.8. *The resilience of an obfuscating transformation.*

We measure resilience on a five-point scale from *trivial* to *one-way* (Figure 3.8 (a)). One-way transformations are special because they cannot be undone. Usually they *remove* information from the program that is useful to the human programmer, but is unnecessary for the correct execution of the program. For example, the formatting of the source code and variable names are not required to execute a program. Other transformations add useless information which does not change observable behaviour, but increases the "information load" on a human reader. These transformations can be undone with varying degrees of difficulty.

Figure 3.8 (b) shows that deobfuscator effort is classified as either *polynomial* or *exponential* time in the size of the program. Programmer effort is measured as a function of the *scope* of $\mathcal{T}$. It is easier to construct countermeasures against an obfuscating transformation that only affects a small part of a procedure, than one that may affect an entire program. The scope of a transformation is described using terminology defined by code optimisation theory (Table 3.2).

Term	Part of program affected
Local	a single basic block of a control flow graph
Global	the entire control flow graph of a procedure
Inter-procedural	the flow of information between procedures
Inter-process	the interaction between independently executing threads of control

Table 3.2. *The scope of a transformation.*

DEFINITION 3 (TRANSFORMATION RESILIENCE) Let $\mathcal{T}$ be a behaviour-conserving transformation, such that $P \xrightarrow{\mathcal{T}} P'$ transforms a source program P into a target program P'. $\mathcal{T}_{\text{res}}(P)$ is the *resilience* of $\mathcal{T}$ target with respect to a program P.

$\mathcal{T}_{\text{res}}(P)$=*one-way* if information is removed from P such that P cannot be reconstructed from P'. Otherwise,

$$\mathcal{T}_{\text{Res}} \stackrel{\text{def}}{=} \texttt{Resilience}(\mathcal{T}_{\substack{\texttt{Deobfuscator} \\ \texttt{effort}}}, \mathcal{T}_{\substack{\texttt{Programmer} \\ \texttt{effort}}}),$$

where `Resilience` is the function defined in the matrix in Figure 3.8 (b). □

Resilience is measured using the five point scale ⟨*trivial*, *weak*, *strong*, *full*, *one-way*⟩.

Suppose that we have a transformation $\mathcal{T}_1$ which has global scope and requires exponential time for the deobfuscator to reduce the potency of the transformation. Then the resilience of $\mathcal{T}_1$ will be *strong* according to Figure 3.8

Assume that we have another transformation $\mathcal{T}_2$ which has inter-process scope but only requires polynomial time deobfuscator effort. In this case the resilience of $\mathcal{T}_2$ will be *full*.

3.4.3 Cost

The *cost* of a transformation is the execution time/space penalty which it incurs on an obfuscated application. We classify the cost on a four-point scale ⟨*free*, *cheap*, *costly*, *dear*⟩, as defined below:

DEFINITION 4 (TRANSFORMATION COST) Let $\mathcal{T}$ be a behaviour-conserving transformation, such that $P \xrightarrow{\mathcal{T}} P'$ transforms a source program P into a target

program P'. $\mathcal{T}_{\text{cost}}(P)$ is the extra execution time/space of P' compared to P.

$$\mathcal{T}_{\text{cost}}(P) \stackrel{\text{def}}{=} \begin{cases} \textit{dear} & \text{if executing } P' \text{ requires } \textit{exponentially} \text{ more resources than } P. \\ \textit{costly} & \text{if executing } P' \text{ requires } \mathcal{O}(n^p),\ p > 1, \text{ more resources than } P. \\ \textit{cheap} & \text{if executing } P' \text{ requires } \mathcal{O}(n) \text{ more resources than } P. \\ \textit{free} & \text{if executing } P' \text{ requires } \mathcal{O}(1) \text{ more resources than } P. \end{cases}$$

□

The actual cost of a transformation depends on the environment in which it is applied. See Figure 3.9 for an example of the different costs of a transformation. Inserting a statement inside an inner loop will have higher cost than inserting the same statement at the top level of a program. Naturally, we will want to minimise the cost of transformations, while maximising the potency and resilience. We achieve this by applying cheap transformations to frequently executed parts of a program. We give the cost of a transformation as if it had been applied at the outermost nesting level of a program.

```
a=5;
for(i=1;i<=n;i++) {
    S1;
}
```
(a)

```
for(i=1;i<=n;i++) {
    a=5;
    S1;
}
```
(b)

Figure 3.9. *Differing transformation costs. In both cases we insert the simple statement* `a=5` *into the code fragments. The variable* `n` *holds an integer that is determined at run-time. In (a) the transformation incurs a constant overhead, since the new code is inserted before the loop and is only executed once. In (b) the overhead is* $\mathcal{O}(n)$*, since the new code is executed on every iteration of the loop.*

3.4.4 Stealth

A resilient transformation may not be susceptible to attacks by automatic deobfuscators, although it may be vulnerable to attacks by humans. If a transformation introduces new code that differs greatly from the code in the original program, it will be easily noticed by a reverse engineer. A predicate (boolean expression) like the one in Figure 3.10 may be very resilient to automatic attacks, but will stick out "like a sore thumb" in most programs.

$$\texttt{if IsPrime}(\overbrace{837523474\ldots3853845347527}^{\text{512-bit integer}})\ \ldots$$

Figure 3.10. *An example of a "sore-thumb" predicate (boolean expression).*

It is essential that obfuscated code resemble the original code as much as possible. Such transformations are *stealthy*. Stealth is a highly context-sensitive metric. A transformation may introduce code which is stealthy in one program but extremely unstealthy in another.

With each transformation $\mathcal{T}$ we associate a set of language features that would be added when $\mathcal{T}$ is applied to a program Q. Language features could include operators, variables, procedure calls to particular routines and whether Q has single or multiple threads of execution. Some of the features of $\mathcal{T}$ may not be among the features used by Q. These features reduce the stealth of $\mathcal{T}$ when it is applied to Q. We formally define the stealth measure as follows:

DEFINITION 5 (STEALTH) Let $\mathcal{T}$ be a behaviour-conserving transformation and Q be a program. $P_s(Q)$ is the set of language features used by Q, while $P_s(\mathcal{T})$ is the set of language features introduced by $\mathcal{T}$. $\mathcal{T}_{\text{ste}}(Q)$ is the stealth of $\mathcal{T}$ when it is applied to Q:

$$\mathcal{T}_{\text{ste}}(Q) \stackrel{\text{def}}{=} \begin{cases} 1.0, \text{ if } |P_s(\mathcal{T}) = 0\,|. \\ \\ 1.0 - \dfrac{|P_s(\mathcal{T}) \setminus P_s(Q)\,|}{|P_s(\mathcal{T})\,|}, \text{ otherwise.} \end{cases}$$

□

We will measure stealth on a three-point scale $\langle$*unstealthy*, *moderate*, *stealthy*$\rangle$. If $\mathcal{T}_{\text{ste}}(Q)$ is close to 1, then $\mathcal{T}$ is considered to be stealthy. Conversely if $\mathcal{T}_{\text{ste}}(Q)$ is close to 0, then $\mathcal{T}$ is unstealthy.

3.4.5 Quality

We can now give a formal definition of the *quality* of an obfuscating transformation:

DEFINITION 6 (TRANSFORMATION QUALITY) $\mathcal{T}_{\text{qual}}(P)$, the quality of a transformation $\mathcal{T}$ when it is applied to a program P, is defined as the combination of the potency, resilience, cost and stealth of $\mathcal{T}$. f is a function that takes a quadruple of transformation measures and returns a scalar.

$$\mathcal{T}_{\text{qual}}(P) = f(\mathcal{T}_{\text{pot}}(P), \mathcal{T}_{\text{res}}(P), \mathcal{T}_{\text{cost}}(P), \mathcal{T}_{\text{ste}}(P)).$$

□

The function f should have the following properties:

1. f increases monotonically as one of $\{\mathcal{T}_{\text{pot}}(P), \mathcal{T}_{\text{res}}(P), \mathcal{T}_{\text{ste}}(P)\}$ increases, provided that the other measures, including $\mathcal{T}_{\text{cost}}(P)$, are held constant.

2. f decreases monotonically as $\mathcal{T}_{\text{cost}}(P)$ increases, provided that $\mathcal{T}_{\text{pot}}(P)$, $\mathcal{T}_{\text{res}}(P)$ and $\mathcal{T}_{\text{ste}}(P)$ are held constant.

Ideally, we want an obfuscating transformation to have high potency, one-way resilience, be free and stealthy. In practice, there is a tradeoff in these four measures. The potency and stealth measures are not independent — as potency increases, stealth decreases and vice versa. This is because the greater the degree that an obfuscating transformation confuses a human reader, the more attention it attracts to itself. Potent transformations tend to signal to a reverse-engineer that they contain code that has been added to the original program. It is thus desirable for these transformations to be as resilient as possible. Conversely, stealthy transformations tend not to be particularly potent and it is not crucial for them to be highly resilient. The cost measure is largely independent to the other three measures but is highly context sensitive. A transformation applied inside an inner loop will have far greater cost than if it is applied outside.

As an example, let us compare layout obfuscations with control flow obfuscations. Layout obfuscations have no effect on the run-time performance of a program — they have free cost. It is often obvious that they have been applied to a program, since information has been removed — they are unstealthy. This does not matter because even if they are detected, these types of transformations cannot be undone — they have one-way resilience. Layout obfuscations vary in their potency. For example, the scramble identifiers transformation has medium potency while the remove formatting transformation has low potency. An identifier name contains much pragmatic information whereas formatting (spaces and blank lines) does not.

Control flow obfuscations have medium to high potency and may result in the program requiring more memory and time to execute — they have cost greater than free. The fact that their resilience is not one-way means that they must be stealthy to avoid being detected and undone by a deobfuscator.

3.5 Using Obfuscation Measures

Our classification scheme has practical and theoretical value. We use a variant of the quality metric in our obfuscator to choose which transformation to apply to a program. Chapter 7 describes the obfuscator in detail. We will develop the abstract definition of transformation quality given in this chapter into the concrete definition used by our obfuscator. To do so, we need to examine what aspects are important when selecting a transformation to apply to a program.

There are many heuristics that can be used to select a transformation but two important issues that need to be considered are:

- $\mathcal{T}$ must blend in with the rest of the code it P.
- $\mathcal{T}$ should give high levels of obfuscation with low execution time penalty (the greatest "bang-for-the-buck").

The first issue can be addressed by choosing those transformations with high stealth when applied to T. The second issue can be handled by selecting transformations that maximise potency and resilience, while minimising cost.

We want a variant of transformation quality which captures these heuristics and is easy to calculate. To do this, we need to assign numerical values to the abstract scales used by the potency, resilience, cost and stealth measures. The values used by our obfuscator are given in Table 3.3.

Metric	**Scale**				
Potency	low	medium	high		
	1	10	100		
Resilience	trivial	weak	strong	full	one-way
	1	10	100	1000	10000
Cost	free	cheap	costly	dear	
	1	10	100	1000	
Stealth	unstealthy	moderate	stealthy		
	1	10	100		

Table 3.3. *Values used by our obfuscator for the transformation measures.*

We can now define the *appropriateness* of a transformation:

DEFINITION 7 (APPROPRIATENESS) Let $\mathcal{T}$ be a behaviour-conserving transformation and P be a program. $\mathcal{T}_{\text{pot}}(P)$, $\mathcal{T}_{\text{res}}(P)$, $\mathcal{T}_{\text{cost}}(P)$ and $\mathcal{T}_{\text{ste}}(P)$ are the potency, resilience, cost and stealth of $\mathcal{T}$ when it is applied to P. ω_1, ω_2 and ω_3 are implementation defined constants that determine the relative importance of the potency, resilience and stealth measures.

$\mathcal{T}_{\text{app}}(P)$ is the appropriateness of $\mathcal{T}$ when it is applied to P.

$$\mathcal{T}_{\text{app}}(P) \stackrel{\text{def}}{=} \frac{\omega_1 * \mathcal{T}_{\text{pot}}(P) + \omega_2 * \mathcal{T}_{\text{res}}(P) + \omega_3 * \mathcal{T}_{\text{ste}}(P)}{\mathcal{T}_{\text{cost}}(P)}$$

□

Suppose that we have two transformations $\mathcal{T}_1$ and $\mathcal{T}_2$ that could be applied to a program P. Their transformation qualities are as follows:

	Potency	Resilience	Cost	Stealth
$\mathcal{T}_1$	high	weak	cheap	stealthy
$\mathcal{T}_2$	medium	one-way	free	unstealthy

Before we can perform the appropriateness calculations for these transformations, we need to assign values to the constants ω_1, ω_2 and ω_3. We use $\omega_1 = \omega_2 = \omega_3 = 1$ in our obfuscator to give equal weighting to the potency, resilience and stealth of an obfuscating transformation, when calculating its appropriateness. The appropriateness of $\mathcal{T}_1$ and $\mathcal{T}_2$ are calculated as follows:

$$\begin{aligned} \mathcal{T}_{1\text{app}}(P) &= \frac{1 * 100 + 1 * 10 + 1 * 100}{10} \\ &= 21 \\ \mathcal{T}_{2\text{app}}(P) &= \frac{1 * 10 + 1 * 10000 + 1 * 1}{1} \\ &= 10010 \end{aligned}$$

Hence, $\mathcal{T}_2$ would be more appropriate than $\mathcal{T}_1$ to apply to P.

3.6 Discussion

We believe that our obfuscation classification scheme is extensive enough to classify any known obfuscation. It has categories for transformations that affect each aspect of a source code program. Layout obfuscations alter the lexical structure of a source code program. Data obfuscations and control flow obfuscations disguise the data structures and control flow of a program respectively. There is a fourth class of transformations, preventive transformations, that target specific deficiencies in decompilers and deobfuscators. This class includes transformations such as inherent preventive transformations, which make particular deobfuscation techniques difficult to employ.

Existing obfuscators fit well into our classification scheme. `Crema` and `Jobe` perform the layout transformation of scrambling identifier names in Java class files. Aggarwal's Java obfuscator [1] also scrambles identifier names and removes the debugging information from a Java class file. This transformation is another kind of layout obfuscation because debugging information is not required to execute a program. The `HoseMocha` obfuscator uses a different approach, exploiting a weakness in the `Mocha` decompiler. `Mocha` crashes if instructions are placed after the last `return` instruction in a method. Hence `HoseMocha` employs a preventive transformation.

3.7 Summary

This chapter has presented a classification and evaluation scheme for obfuscating transformations. With this scheme, it is possible to determine the best transfor-

mations to apply to an application. In the next chapter we focus on control flow obfuscations, a particular category of obfuscating transformations which disguise the real control flow in a program.

CHAPTER 4

Control Flow Obfuscation

"Who controls the past . . . controls the future:
who controls the present controls the past"
– *Nineteen Eighty-Four*, George Orwell

4.1 Introduction

In this chapter we present a few control flow obfuscations, which are a particular category of code obfuscation. Control flow obfuscations disguise the real control flow in a program. For such transformations, a certain amount of computational overhead will be unavoidable. Thus, there is a trade-off between a highly efficient program, and one that is highly obfuscated. Typically, an obfuscator will assist in this trade-off by allowing a user to choose between cheap and expensive transformations.

Before examining examples of control flow obfuscations, we first discuss the notion of opaque predicates, which are an important element of many control flow obfuscations. We summarise portions of the papers by Collberg et al. [9, 11] in this chapter.

4.2 Opaque Constructs

Control-flow transformations must be cheap yet resilient to attack from deobfuscators. Many such transformations rely on the existence of *opaque constructs*. Informally, a variable is opaque if it has a property that is known *a priori* to the

obfuscator, but is difficult for the deobfuscator to determine. Similarly, a predicate (a boolean expression) is opaque if a deobfuscator can determine its outcome only with difficulty, while this outcome is known by the obfuscator. Formally, we define opaque variables and predicates as follows:

DEFINITION 8 (OPAQUE VARIABLES AND PREDICATES) A variable V is *opaque* at a point p in a program, if V has a property q at p which is known at obfuscation time. We write this as V_p^q or V^q if p is clear from context.

A predicate P is opaque at p if its outcome is known at obfuscation time. We write P_p^F (P_p^T) if P always evaluates to `False` (`True`) at p, and $P_p^?$ if P sometimes evaluates to `True` and sometimes to `False`. See Figure 4.1. Again, p will be omitted if it is clear from the context. □

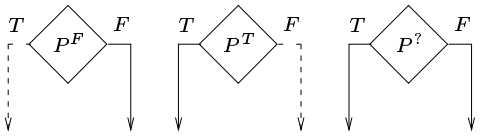


Figure 4.1. *Different types of opaque predicates. Solid lines indicate paths that may sometimes be taken, dashed lines paths that will never be taken.*

Suppose that C is an opaque construct with a property p that is known at obfuscation time. If a deobfuscator can deduce p by analysing C, we say that the deobfuscator *breaks* C. Hence the key to highly resilient control transformations is the ability to create opaque constructs that are difficult for a deobfuscator to break.

4.2.1 Opaque Construct Quality

The quality of control-flow transformations that use opaque constructs depends directly upon the quality of these constructs. We use the same scales to measure the quality of opaque constructs as we do with obfuscating transformations.

The potency of an opaque construct measures how hard it is to understand how the construct is calculated. The greater the complexity of the calculations, the greater the potency of the construct. For example, opaque predicates based on aliasing and threads have higher potency in general than those based on mathematical facts.

The resilience of an opaque construct measures the effort required to create a deobfuscator that is able to break the opaque construct. Resilience is calculated using the scheme in Figure 3.8.

The cost of an opaque construct depends upon where it is inserted into a program, as well as the time and space required to calculate the construct. The constructs described in this chapter use little additional run-time resources.

The stealth of an opaque construct is context-sensitive. If a program performs a lot of pointer manipulations, then opaque predicates based on aliasing will be stealthy. On the other hand, opaque predicates based on mathematical facts will be unstealthy unless a program performs a lot of arithmetic.

4.2.2 Static and dynamic analysis

There are different kinds of deobfuscation techniques, some based on static analysis and some based on dynamic analysis. In this thesis, we are mostly concerned about static analysis attacks on obfuscated programs.

Static analysis of a program is performed without executing the program, whereas dynamic analysis takes place at run-time [16]. Common static analyses include detection of dead code and uninitialised variables. Dynamic analysis is performed by testing the program on sample input data. It is infeasible to test all control paths in a program due to combinatorial explosion. Hence the input data is partitioned into equivalence classes, which correspond to properties such as statement and branch coverage. For example, suppose that there is a predicate in the program that checks if a variable `x` is greater than zero. There are two equivalence classes of input for this predicate, `x > 0` and `x <= 0`.

4.2.3 Trivial and weak opaque constructs

An opaque construct is *trivial* if a deobfuscator can break it via a static *local* analysis. That is, a deobfuscator need only examine a single basic block in the control graph. See Figure 4.2(a) for some examples of trivial opaque constructs.

An opaque variable is also trivial if its value is computed using calls to library functions with simple, well-understood semantics. Languages like Java require all implementations to support a standard set of library classes, so opaque variables of this form are easy to construct. An example is $\mathtt{v}^{\in[0\ldots3]} = (\mathtt{int})(\mathtt{Math.random}() * 4)$. Unfortunately, these opaque variables are equally easy to break. The deobfuscator designer can tabulate the semantics of all simple library functions, and then perform pattern-matching on the function calls in the obfuscated code.

An opaque construct is *weak* if a deobfuscator can break it by a static *global* analysis. In other words the deobfuscator has to examine a whole control flow graph. See Figure 4.2(b) for some examples of weak opaque constructs.

Opaque constructs based on mathematical facts such as $\forall x \in \mathbb{N}, x^2(x+1)^2 \,|\, 4$ have resiliences ranging from trivial to strong, depending on how difficult they are to prove with an automatic theorem prover. The potency of these constructs is medium, rather than high, because well-known mathematical facts can be

```
{  ① int v, a=5; b=6;
   ② v^=11 = a + b;
   ③ if (b > 5)^T  ...
   ④ if (Math.random() < 0)^F  ...
}
```

(a)

```
{  ① int v, a=5; b=6;
   ② if (...) ...
      /* a and b are
         unchanged */
         ⋮
   ③ if (b < 7)^T  a++;
   ④ v^=36 = (a^=6 > 5)?(b*b):b;
}
```

(b)

Figure 4.2. *Examples of (a) trivial and (b) weak opaque constructs. In (a), lines ① and ② set the values of the opaque variables* `v`*,* `a` *and* `b`*. Since the value of* `b` *is not changed before the statement in line ③ is executed, we know that* $\texttt{b} > 5$ *is true. The random numbers generated by the* `Math.random()` *method are in the range* $[0 \ldots 1)$*, hence the expression in line ④ is always false. In (b), line ① assigns values to the opaque variables* `a` *and* `b`*, which are unchanged by the statements in line ②. So in line ③,* $\texttt{b} < 7$ *is true and* `a` *is incremented. In line ④, since* $\texttt{a} > 5$ *is true,* `v` *is assigned the value of* $\texttt{b} * \texttt{b}$*.*

easily identified by a human reading the program. The construct relying on this mathematical fact can then be broken.

Trivial and *weak* opaque constructs can be broken by using local or global static analysis. Ideally, we want opaque constructs that require worst case exponential time in the size of the program to break but only polynomial time to construct. This is a similar situation to cryptography, where decryption is much harder than encryption, unless one has the correct decryption key. We will present two techniques for creating highly resilient opaque predicates — the first is based on aliasing, the second on lightweight processes (threads).

4.2.4 Opaque constructs using objects and aliases

Two or more expressions that denote the same memory address are *aliases* of each other. Figure 4.3 presents a trivial example.

Static analysis is made significantly more complicated when aliasing can occur. Different versions of precise static alias analysis are NP-hard [23] or undecidable [42]. We exploit this fact to construct opaque predicates which are difficult to break by static analysis alone. There are many fast but imprecise alias analysis algorithms that will detect some aliases some of the time, but not all aliases all of the time. Such algorithms are conservative in behaviour. Steensgaard [46] demonstrates an interprocedural flow-insensitive points-to analysis that is very fast in practice for real programs written by humans. The points-to problem is

```
public class Alias {
   public int x;
}

{  ① Alias p = new Alias();
   ② Alias q = p;
   ③ p.x = 5;
   ④ q.x = 6;
}
```

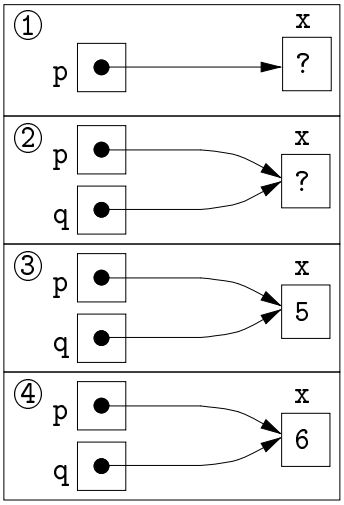


Figure 4.3. *An example of aliasing. The diagram on the right shows the objects to which* `p` *and* `q` *point to after the execution of each line. p and q are aliases after the statement on line ② has been executed. Hence after the statements on lines ③ and ④ have been executed, the value of p.x is the integer 6.*

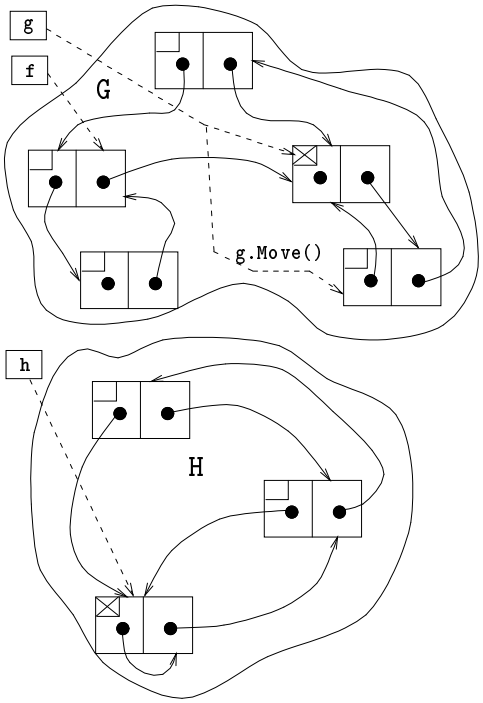
related to the aliasing problem but the analysis requires the use of a special type system.

To exploit the general difficulty of the alias analysis problem, we construct a complex dynamic structure and maintain a set of pointers into this structure. Opaque predicates can then be designed which ask questions that can only be answered if an inter-procedural alias analysis has been performed.

Consider the obfuscated method `P` in Figure 4.4. Mixed in with `P`'s original code are redundant computations guarded by opaque predicates. The method calls manipulate two global pointers `g` and `h` which point into different connected components (`G` and `H`) of a graph. The statement `g=g.Move()` makes `g` point to another node in `G`. The statement `h = h.Insert(new Node)` inserts a new node into `H` and makes `h` point to another node in `H`. `P` (and other methods that `P` calls) is given an extra pointer argument `f` which refers to objects within `G`.

This set-up allows us to construct opaque predicates like those of statements ④ and ⑤ in Figure 4.4. The predicate `f==g` may be either true or false since `f` and `g` move around within the same structure. Conversely, `g==h` must be false since `g` and `h` refer to nodes in different structures.

Statements ⑥–⑨ in Figure 4.4 exploit aliasing. The predicate in statement ⑦ will be true or false depending on whether `f` and `g` point to the same or different objects. The predicate in statement ⑧ must evaluate to true since `f` and `h` cannot alias the same object.




```
public class Node {
   /* Instance methods
      omitted */
   public boolean Token;
   public Node car, cdr;
}

Node g, h;
method P(...,Node f) {
   ① g = g.Move();
      h = h.Move();
   ② h = h.Insert(new Node);
         ⋮
   ③ x.R(..., f.Move());
         ⋮
   ④ if (f == g)^?  ...
   ⑤ if (g == h)^F  ...
         ⋮
   ⑥ f.Token=False;
      g.Token=True;
   ⑦ if (f.Token)^?  ...
         ⋮
   ⑧ f.Token=True;
      h.Token=False;
   ⑨ if (f.Token)^T  ...
}
```

Figure 4.4. *Opaque predicates constructed from objects and aliases. We construct a dynamic structure made from* `Node`*s. Each* `Node` *has a boolean field* `Token` *and two pointer fields (represented by black dots) which can point to other nodes. The structure is designed to consist of two connected components,* `G` *and* `H`*. There are two global pointers,* `g` *and* `h`*, pointing into* `G` *and* `H`*, respectively.*

4.2.5 A Graph ADT

To make the technique based on aliasing more concrete, we present a Java `Graph` abstract data type (ADT), which can be included in an obfuscated application. The `Graph` operations are used to manufacture opaque predicates that are *cheap*, and *resilient*. As long as the original application uses pointers (Java object references), these predicates will also be *stealthy*. The `Graph` ADT uses a `Set` ADT,

which is defined in Appendix C.

We must prevent deobfuscators from identifying the ADT by simple pattern matching. We can foil deobfuscators with the following techniques:

1. The obfuscator should keep a large library of variants of the `Graph` ADT that it could randomly select between. Several variants could be included with and used in different parts of the same application.

2. Invocations of the `Graph` primitives should be obfuscated like the user code, including inlining, outlining, and identifier scrambling.

3. The `Graph` ADT could be merged with the most similar user-defined class. The `Graph` nodes created by the obfuscated application would be indistinguishable from existing objects created by the original application.

For sake of clarity we will not use any of these techniques in our examples.

Consider the `Graph` ADT shown in Figure 4.5. It contains operations for creating a new graph (`Node`), adding new nodes to a graph ($\texttt{addNode}_i$), traversing a graph ($\texttt{selectNode}_i$), and splitting a graph into components (`reachableNodes` and `splitGraph`). Additional operations that could be included are merging graphs, changing the direction of edges, and testing for various graph properties such as connectivity, acyclicity, reachability and isomorphism.

The `Graph` ADT operations are combined into patterns that can be inserted into the application. See Table 4.1 for some examples.

The following patterns ensure that the graph that is built is essentially tree-shaped. That is, if `P` and `Q` point to nodes in a connected graph, then after one of these operations is performed, `P` and `Q` can possibly refer to the same node.

- `Insert` (Table 4.1(a)) inserts a new node at an arbitrary place in the graph.

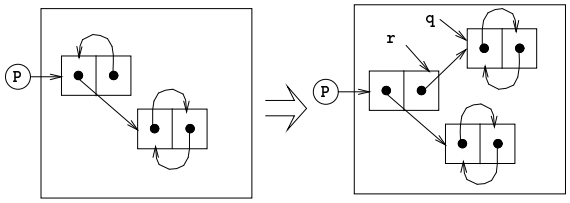
- `Move` (Table 4.1(b)) makes `P` point to an arbitrary graph node reachable from `P`. Note that if the node `P` pointed to before the move becomes unreachable and there are no other pointers to it, the memory it uses will be reclaimed by the garbage collector.

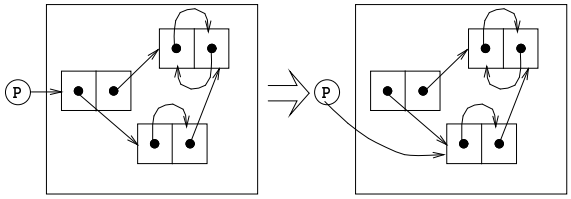
- `Link` (Table 4.1(c)) builds general graphs by adding an edge from some leaf `b` to an arbitrary node `a`. The requirement that we only add edges from leaves ensures that the graph will remain connected.

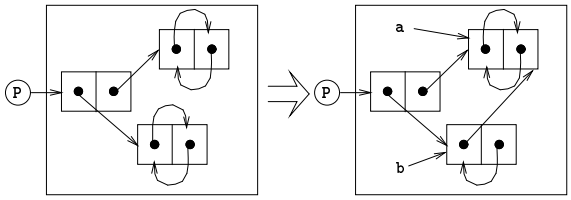
In contrast to the previous three code patterns, `Split` (Table 4.1(d)) divides the graph pointed to by `P` into two separate components. After the split we know that `P` and `Q` point into separate components. No matter which operations are performed separately on these components, `P` and `Q` will never alias one another.

Figure 4.7 shows how these patterns can be used to construct opaque predicates in a real program. We start by creating two graphs pointed to by `p` and `q`.

#	Code Pattern	Example

(a)	Node Insert$_{i,j}$(Node P) {
	  if (P == null)
	  return new Node();
	  else {
	  r = P.selectNode$_i$();
	  q = r.addNode$_j$();
	  return q;
	  };
	}
		

(b)	Node Move$_i$(Node P) {
	  return P.selectNode$_i$();
	}
		

(c)	void Link$_{i,j}$(Node P) {
	  a = P.selectNode$_i$();
	  b = P.selectNode$_j$();
	  if (b.car == b)
	  b.car=a;
	}
		

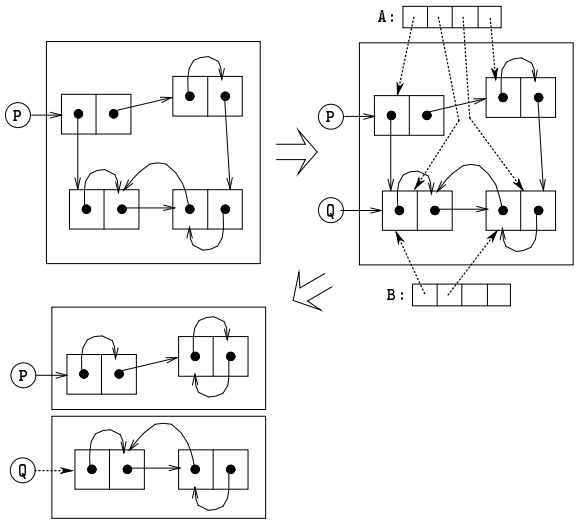
(d)	Node Split$_i$(Node P) {
	  Q = P.selectNode$_i$();
	  A = P.reachableNodes();
	  B = Q.reachableNodes();
	
	  P.splitGraph(
	  A.setDiff(B),B);
	  return Q;
	}
		

Table 4.1. *An example of* Graph *ADT code patterns. The obfuscator inserts these code patterns, which use the* Graph *primitives defined in Figure 4.5, into a program. A number of graphs and pointers into these graphs are maintained, which allows the obfuscator to create resilient opaque predicates.*

```
public class Node {
  public Node car, cdr;

  public Node()
    { this.car = this.cdr = this; }

  /* addNode_i is a family of functions which insert a
    new node after 'this'. */
  public Node addNode_1() {
    Node p = new Node(); p.car = this.car;
    return this.car = p; }
  public Node addNode_2() {
    Node p = new Node; p.cdr = this.car;
    return this.car = p; }

  /* selectNode_i is a family of functions which return a
    reference to a node reachable from 'this'. */
  public Node selectNode_1() { return this; }
  public Node selectNode_2() { return this.car; }
  public Node selectNode_3() { return this.car.cdr; }

  public Node selectNode_4(int n) {
    return (n <= 0) ?  this :
      this.car.selectNode_4b(n-1); }

  public Node selectNode_4b(int n) {
    return (n <= 0) ?  this :
      this.cdr.selectNode_4(n-1); }
```

Figure 4.5. *A simple* `Graph` *ADT to be used for manufacturing opaque predicates. Due to length, the definition of this ADT is continued in Figure 4.6. To ensure there are no* `null` *pointers, terminal nodes point to themselves. This simplifies the implementation of the* `selectNode`$_i$ *family of functions. The class* `Set` *(see Appendix C) implements sets of objects and has operations* `insert` *and* `hasMember`*. The* `Graph` *primitives defined in this figure are used by the code patterns in Table 4.1.*

We now consider the execution of the outer `for`-loop. We add a node to the graph pointed to by `p` if `y` $\in$ [`v.height` $-$ `10` $\ldots$ `v.height`]. This means that only ten extra graph nodes have to be allocated over the entire execution of the loop. When `y` == `v.height` becomes true, the graph pointed to by `p` is split, the new

```
  /* Return the set of nodes reachable from 'this'. */
  public Set reachableNodes()
    { return reachableNodes(new Set()); }
  private Set reachableNodes(Set reached) {
    if (!reached.hasMember(this)) {
      reached.insert(this);
      this.car.reachableNodes(reached);
      this.cdr.reachableNodes(reached); }
    return reached; }

  /* A and B are sets of graph nodes.  Remove any
    references between nodes in A and B. */
  public void splitGraph(Set A, Set B)
    { this.splitGraph(new Set(), A, B); }
  private void splitGraph(Set R, Set A, Set B) {
    if (!R.hasMember(this)) {
      R.insert(this);
      this.car.splitGraph(R, A, B);
      this.cdr.splitGraph(R, A, B);
      if (this.diffComp(this.car, A, B))
        this.car = this;
      if (this.diffComp(this.cdr, A, B))
        this.cdr = this; }
  }

  /* Returns true if the current node and node b */
    are in different components */
  private boolean diffComp(Node b, Set A, Set B) {
    return (A.hasMember(this) && B.hasMember(b)) ||
      (B.hasMember(this) && A.hasMember(b)); }
}
```

Figure 4.6. *A simple* `Graph` *ADT to be used for manufacturing opaque predicates. This figure continues the code in Figure 4.5.*

component that is formed is pointed to by `q`. This is a destructive update since it causes the graph originally pointed to by `q` to no longer be referenced. When the inner `for`-loop is about to be executed, `p != q` is false, since `p` and `q` point to different graph components. Thus, the added `break`-statement is never executed.

```
static void Render (Vector world, View v) {
  Node p = Insert_{1,1}(null); Insert_{1,2}(p);
  Node q = Insert_{1,1}(null);
  for (int y = 0; y < v.height; y++) {
    if (y>=v.height − 10)?
      Insert_{4,2}(p,(int) (y * 1.5));
    if (y == v.height − 10)?
      q = Split_1(p);
    for (int x = 0; x < v.width; x++) {
      if (Move_4(p,x)==Move_{4b}(q,x))^F
          break;
      Ray ray = v.ray(x, y);
      WorldObject o = hitObject(ray, world);
      if (o != null) {
        Color c = o.surface.color(o.hitPt, o.norm, v.eyePt);
        Graphics.drawPoint(c, x, y);
}}}}
```

Figure 4.7. *Inserting* `Graph` *ADT routines. The inserted code is in italics. It is constructed so that* `p` *and* `q` *will never point into the same dynamic structure.*

4.2.6 Opaque constructs using threads

Parallel programs are more difficult to analyse statically than sequential ones. The reason is their *interleaving* semantics: n statements in a parallel region

PAR S_1; S_2; ...; S_n; **ENDPAR**

can be executed in $n!$ different ways. Despite this, some static analyses of parallel programs can be performed in polynomial time [30]. Others require all $n!$ interleavings to be considered.

In Java, parallel regions are constructed using lightweight processes known as *threads*. Java threads have (from our point of view) two very useful properties:

1. Their scheduling policy is not specified strictly by the language specification and will hence depend on the implementation.

2. The actual scheduling of a thread will depend on asynchronous events generated by user interaction, network traffic, etc.

Combined with the inherent interleaving semantics of parallel regions, this means that threads are very difficult to analyse statically.

We use these observations to create opaque predicates that require worst-case exponential time to break. The basic idea is very similar to the one used in Section 4.2.4. A global data structure V is created and occasionally updated,

but kept in a state such that opaque queries can be made. The difference is that V is updated by concurrently executing threads.

Obviously, V can be a dynamic data structure such as the one created in Figure 4.4. The threads would randomly move the global pointers `g` and `h` around in their respective components, by asynchronously executing calls to `move` and `insert`. This has the advantage of combining data races with interleaving and aliasing effects, for very high resilience. An example of this combination of techniques is shown in Figure 4.8.

```
thread S {
  while (1) {
    if (random(1,10) <= 9)
      X = X.Move();
    else
      X = X.Insert(new Node);
    sleep(3);
  }
}
```

```
thread T {
  while (1) {
    if (random(1,20) <= 17) {
      Y = Y.Insert(new Node);
      sleep(5);
    }
    X = X.Move();
    sleep(11);
  }
}
```

```
Node X, Y;
main () {
  X = new Node;
  Y = new Node;
  S.run(); T.run();
      ...
  if (Y == X)^F ⇐ ①
    ...
}
```

Figure 4.8. *An opaque predicate constructed using threads and aliasing. The predicate at point ① will always evaluate to* `False`*, since* `X` *and* `Y` *point to different structures. Two threads* `S` *and* `T` *occasionally wake up to update the structures pointed to by the global variables* `X` *and* `Y`*. Notice that* `S` *and* `T` *are involved in a data-race on* `X`*, but that this does not matter as long as assignments are atomic. Regardless of whether* `S` *or* `T` *wins the race,* `X` *will remain pointing to a different structure than* `Y`*.*

4.3 Computation Obfuscations

Control computation obfuscations fall into the three categories in Figure 4.9, which are discussed in the following sections.

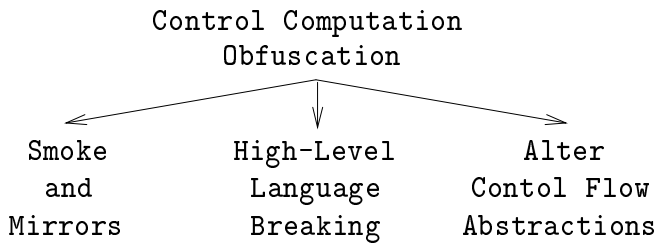


Figure 4.9. *Control computation obfuscation categories.*

4.3.1 "Smoke and Mirrors" obfuscations

Inserting dead code into a program is an example of a "Smoke and Mirrors" obfuscation. The aim is to hide the real control flow behind statements that are irrelevant.

The McCabe [36] and Harrison [20] metrics (μ_2 and μ_3 in Table 3.1) suggest that there is a strong correlation between the perceived complexity of a piece of code and the number of predicates it contains. Fortunately, the existence of opaque predicates makes it easy for us to devise transformations that introduce new predicates in a program.

Consider the basic blocks $A; B;$ in Figure 4.10. In Figure 4.10(a) we insert an opaque predicate P^T into this basic block, splitting up the statements A and B. The P^T predicate is *irrelevant* code since it will always evaluate to True.

In Figure 4.10(b) we again break the basic block into two halves, and then proceed to create two *different* versions B' and B'' of the statement B. We create B' and B'' by applying different sets of obfuscating transformations to B. It will not be obvious to a reverse engineer that the two new versions of B in fact perform the same function. We use a predicate $P^?$ to select between B' and B'' at run-time.

Figure 4.10(c) is similar to Figure 4.10(b), but this time we introduce a bug into B''. The P^T predicate always selects the correct version of the code, B".

4.3.2 High-level language breaking obfuscations

High-level language breaking obfuscations introduce features at the object code level that have no direct source code equivalent. For example languages that do

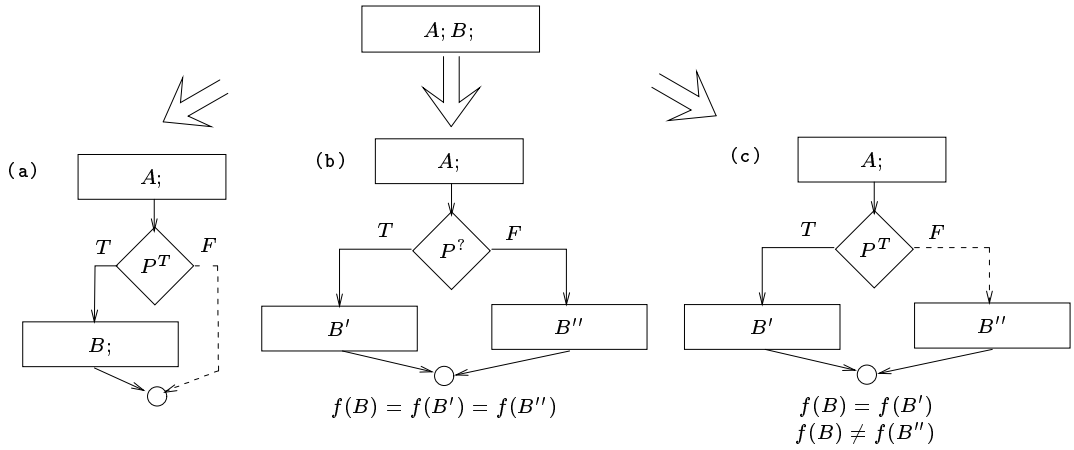


Figure 4.10. *The branch insertion transformation.*

not have `goto`-statements are unable to represent non-reducible flow graphs.

Often a programming language is compiled to a native or virtual machine code which is more expressive than the language itself. If this is the case, we can devise *language-breaking* transformations, which introduce a sequence of virtual machine (or native code) instructions that have no direct correspondence with any source language construct. When faced with such instruction sequences a deobfuscator will either have to try to synthesize an equivalent (but convoluted) source language program, or give up altogether.

For example, Java *bytecode* has a `goto` instruction while the Java *language* has no corresponding `goto`-statement. This means that the Java bytecode can express *arbitrary* control flow, whereas the Java language can only (easily) express *structured* control flow. We say that the control flow graphs produced from Java programs will always be *reducible*, but Java bytecode can express *non-reducible* flow graphs (see Aho et al. [2], pp. 606–608). The main feature of reducible flow graphs is that there are no jumps into the middle of a loop.

Since expressing non-reducible flow graphs becomes very awkward in languages without `goto`s, we construct a transformation which converts a reducible flow graph to a non-reducible one. This can be done by turning a structured loop into a loop with multiple headers. In Figure 4.11(a) we add an opaque predicate P^F to a `while`-loop, to make it appear that there is a jump into the middle of the loop. In fact, this branch will never be taken.

A Java decompiler would have to turn a non-reducible flow graph into one which either duplicates code or which contains extraneous boolean variables. Alternatively, a deobfuscator could guess that all non-reducible flow graphs have been produced by an obfuscator. It can then simply remove the branch which makes the flow graph non-reducible, along with what appears to be an opaque

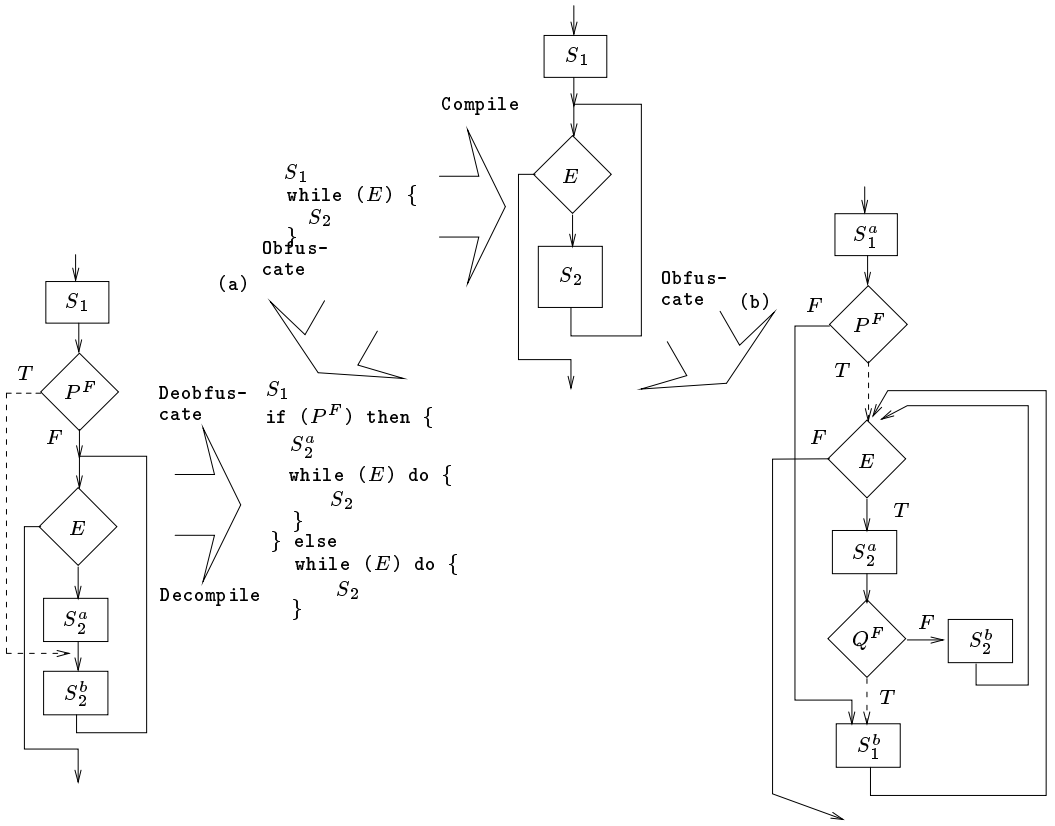


Figure 4.11. *Reducible to non-reducible flow graphs. In (a) we split the loop body S_2 into two parts (S_2^a and S_2^b), and insert a bogus jump to the beginning of S_2^b. In (b) we also break S_1 into two parts, S_1^a and S_1^b. S_1^b is moved into the loop and an opaque predicate P^T ensures that S_1^b is always executed before the loop body. A second predicate Q^F ensures that S_1^b is only executed once.*

predicate guarding this branch. Assuming that all non-reducible flow graphs have been generated by an obfuscator will be wrong if the code has been compiled from a language that does have `goto`-statements. For example, the C language source program in Figure 4.12 contains a `goto`-statement in line ② which makes the flow graph for the program non-reducible. The `goto`-statement is always executed because the value of `x` is set to five in line ①. Thus the body of the `while`-loop in lines ③-⑥ is executed once, which is not the case if the `goto`-statement is removed.

We can use the alternative transformation shown in Figure 4.11(b) to foil deobfuscators. If a deobfuscator blindly removes P^F, the resulting code will be incorrect.

```
① int i = 10, x = 5;
② if (x>0) goto target;
③ while (i < 10) {
④    target:
⑤    printf("Hello\n");
⑥    i++; }
```

Figure 4.12. *A C language program containing a* `goto`*-statement.*

4.3.3 Control flow abstraction altering obfuscations

Control flow abstraction is the process of taking a sequence of low-level instructions and forming an equivalent description at a higher level. We can reverse this process and de-abstract (remove abstraction) from the program. For example, a `for`-loop in the C language source code can be changed into an equivalent loop that uses `if`- and `goto`-statements (Figure 4.13). It is much more difficult for a human reader to understand the transformed program because the destination of the `goto`-statements have to be identified. The semantics of a `for` loop are well-known and much easier to comprehend.

```
int i;                                  int i;
for (i = 0; i < 100; i++)    T          i = 0;
   printf("%d\n", i);       ⇒          loop:
                                          if (i >= 100) goto endloop;
                                          printf("%d\n", i);
                                          i++;
                                          goto loop;

                                       endloop:
```

Figure 4.13. *Control flow de-abstraction. A C language example of transforming a* `for`*-loop into* `if`*- and* `goto`*-statements.*

4.4 Aggregation Obfuscations

Programmers overcome the inherent complexity of programming by introducing abstractions into programs, the *procedural* abstraction being one of the most important. Thus, obscuring procedure and method calls is of the utmost importance to the obfuscator. Inlining and outlining is one of the most effective ways in which methods and method invocations can be obscured. The basic ideas behind inlining and outlining methods are:

- Code which the programmer aggregated into a method (presumably the code segments logically belonged together) should be broken up and scattered over the program.
- Code which seems *not* to belong together should be aggregated into one method.

4.4.1 Inline and outline methods

Inlining is an important compiler optimisation. It is also an extremely useful obfuscation since it removes procedural abstractions from a program. It is a highly resilient transformation being essentially *one-way*. Once a procedure call has been replaced with the body of the called procedure and the procedure itself removed, no trace of the abstraction is left in the code.

Outlining (turning a sequence of statements into a subroutine) is a very useful companion transformation to inlining. Figure 4.14 shows how procedures P and Q are inlined at their call-sites, and then removed from the code. Subsequently, we create a false procedural abstraction by extracting the beginning of Q's code and the end of P's code into a new procedure R.

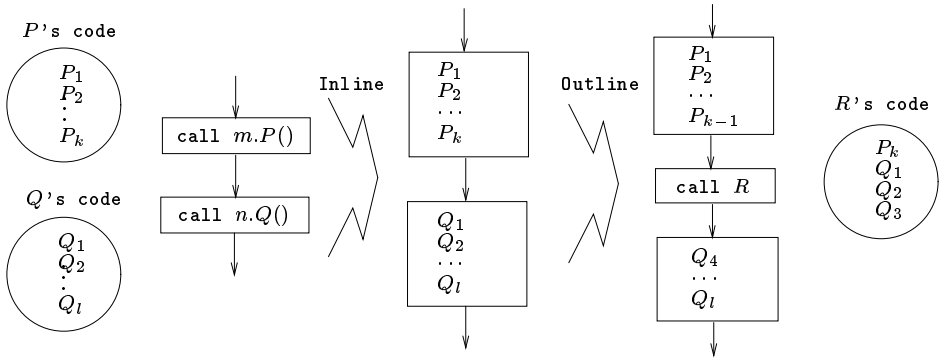


Figure 4.14. *Inlining and outlining transformations.*

In object-oriented languages such as Java, inlining may not always be a fully one-way transformation. For a method invocation $m.P()$, the actual procedure called will depend on the run-time type of m. If more than one method can be invoked at a particular call site, we have to inline all possible methods [15] and select the appropriate code by branching on the type of m (see Figure 4.15). Hence, even after inlining and removal of methods, the obfuscated code may still contain some traces of the original abstractions.

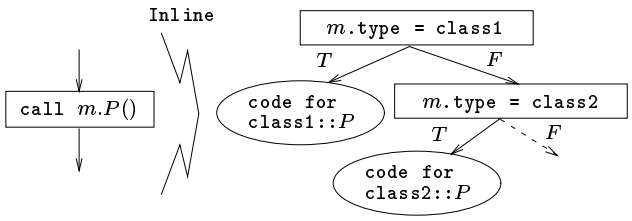


Figure 4.15. *Inlining method calls. Unless we can statically determine the type of* `m`*, all possible methods to which* `m.P()` *could be bound must be inlined at the call site.*

4.5 Ordering Obfuscations

Programmers tend to organise their source code to maximise its *locality*. A program is easier to read and understand if two logically related items are also physically close in the source text. This kind of locality works on every level of the source code: there is locality among terms within expressions, statements within basic blocks, basic blocks within methods, methods within classes, classes within files, etc. All kinds of spatial locality can provide useful clues to a reverse engineer. Thus, we randomise the placement of any item, wherever possible, in the source application. For some types of items, such as methods within classes, this is trivial. In other cases, such as statements within basic blocks, a data dependency analysis (see [4, 53]) will have to be performed to determine which reorderings are legal.

These transformations have low potency (they don't add much obscurity to the program) but their resilience is high, in many cases *one-way*. For example, when the placement of statements within a basic block has been randomised, there will be no traces of the original order left in the resulting code.

Ordering transformations can be particularly useful companions to the "inline-outline" transformation of Section 4.4.1. The potency of that transformation when it is applied to a procedure P can be enhanced by performing the following steps:

1. Inline several procedure calls in P.
2. Randomise the order of the statements in P.
3. Outline contiguous sections of P's statements.

In this way, unrelated statements that were previously part of several different procedures are brought together into bogus procedural abstractions.

In certain cases it is also possible to reorder loops, for example by running them backwards. Such *loop reversal* transformations are common in high-performance compilers [4].

4.6 Discussion

We have given an overview of control aggregation and ordering obfuscations in this chapter but they need to be studied in more depth. The most resilient and stealthy opaque predicates are based on problems which have been shown to be in the NP class of problems or are undecidable. There are certain instances of NP problems are tractable [45], so further research will have to determine which NP problems are suitable for creating opaque predicates.

In the following sections, we discuss the issues that arise in implementing opaque predicates based on mathematical facts and aliasing.

4.6.1 Predicates based on mathematical facts

Opaque predicates based on mathematical facts have high resilience. However, these predicates have varying degrees of stealth, depending on the part of the program in which they are used. Overflow is another problem. For example, if we want to insert the predicate `if (x`^{2}`(x + 1)`^{2} `% 4 == 0)`^{T}, where `x` is an integer variable, we must ensure that $-216 <$ `x` < 215. Otherwise the expression `x`^{2}`(x + 1)`^{2} exceeds the maximum 32 bit signed integer value $(2^{31} - 1)$.

In Java, integer overflow does not cause a run-time error. Instead, the value of an integer expression is truncated to 32 bits. It would be best if a predicate holds even if overflow occurs. However, if overflow does cause a predicate to evaluate to the wrong result, we must analyse the program to determine the values that `x` can have when this predicate is evaluated. In the example above, it might be the case that $-216 <$ `x` < 215 always holds when the predicate is executed, so we do not have to worry about overflow.

If overflow can occur, we can transform the predicate by adding conditions:
`if (((x <= -216) || (x >= 215)) || (x`^{2}`(x + 1)`^{2} `% 4 == 0))`^{T}.
Lazy evaluation of the operands of the `or` operator (||) will mean that the expression `(x`^{2}`(x + 1)`^{2} `% 4 == 0)` is not evaluated unless $-216 <$ `x` < 215. However, guarding against overflow by extending the conditions of a predicate lowers its resilience. In the example above, it is obvious that the entire predicate is true for $-216 <$ `x` < 215, which suggests that the expression `x`^{2}`(x + 1)`^{2} is true if `x` is not in this range. Hence a deobfuscator can deduce that the expression is very likely to be true all of the time.

4.6.2 Predicates based on aliasing

Opaque predicates that rely on the difficulty of alias analysis have high potency and resilience. We have presented a scheme in this chapter that is based on a `Graph` abstract data type. Ideally, an obfuscated application would create several global data structures which are used by opaque predicates throughout

the program. However, additional storage is required to create and modify these dynamic structures. We must make sure that the obfuscated program does not run out of dynamic storage space during execution. Languages with garbage collection will be able to reclaim unreferenced dynamic storage. However, this will not help if dynamic storage is being allocated at a greater rate than it is being deallocated. Currently our obfuscator only uses data structures local to a single procedure and does not worry about the amount of dynamic storage being allocated.

Combining aliasing and threads allows us to create opaque predicates that have high potency, full or one-way resilience, are stealthy and cheap. The use of threads also offers a partial solution to the problem of excessive dynamic storage usage. The threads that maintain the dynamic structures occasionally go to sleep. Hence the rate at which new dynamic storage is requested will be reduced and thus the number of requests for dynamic storage made by these threads during the execution of the obfuscated program will be reduced.

4.7 Summary

This chapter has discussed control flow obfuscations and opaque predicates in detail. In the next chapter, we will examine how selected features of Java bytecodes help with and hinder code obfuscation.

CHAPTER 5

Java Obfuscation in Practice

"Practice makes perfect"

– Proverb

5.1 Introduction

As we have mentioned, Java bytecodes are particularly vulnerable to decompilation. In this chapter we describe the Java run-time environment, class file format, bytecode instruction set and verifier. We examine the features of Java bytecodes, and explain how they help or hinder decompilation and obfuscation.

An obfuscator which operates on Java bytecodes rather than the source code does not need to perform semantic analysis. Also, there are obfuscations which can only be applied at the object code, such as creating non-reducible flow graphs (Section 5.6.3).

The version of Java that we describe in this chapter is version 1.0.2, as implemented in Sun Microsystems, Inc., Java Developer's Kit (JDK) 1.0.2. A full treatment of the Java class file format and instruction set is not given here. See Lindholm and Yellin [32] for a complete description of the Java Virtual Machine.

5.2 Executing Java Code

In this section, we will discuss the issues involved in executing Java programs. The stages that are required can be seen in Figure 5.1.

The Java source program is compiled by the Java compiler into Java class files. These are then loaded by the class loader, either locally or through a network.

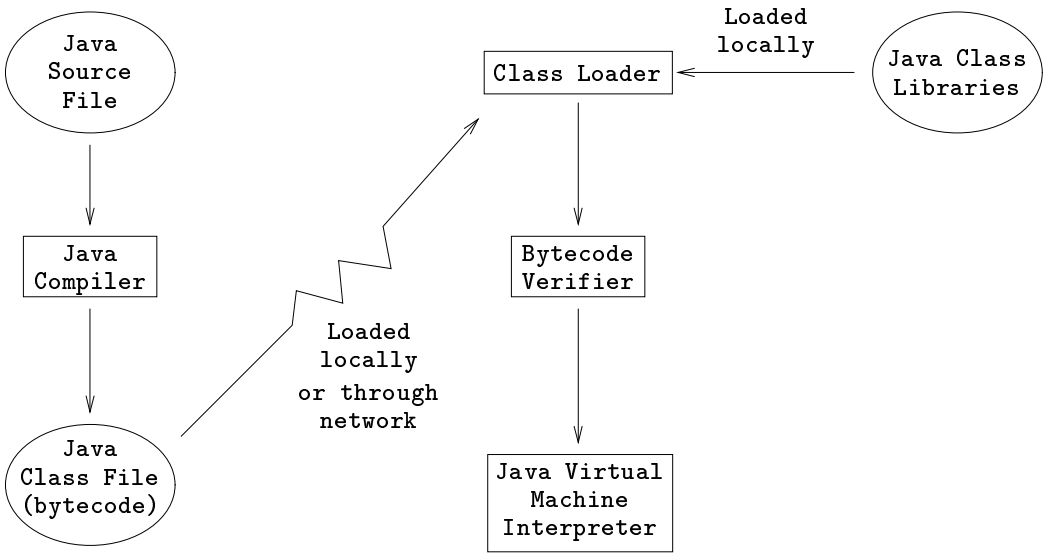


Figure 5.1. *Compilation and execution of a Java program.*

The Java class libraries required are also loaded at this stage. Before the class files are executed, they must be checked by the Java verifier. If no verification errors occur, the classes are executed by the Java virtual machine.

5.2.1 The class loader

To execute a Java program, the interpreter is given the name of the main class in the program. This bytecode class file is then searched for in the file system. Some features of bytecode class files are examined in Section 5.3.

For each class `A` that is loaded into memory, the class loader determines the classes that are used by `A`. If these classes are not already present in memory, they must also be loaded into memory. This action is performed recursively until all the classes used by a program are present in memory. The classes are then checked by the bytecode verifier.

5.2.2 The bytecode verifier

The problem with distributing programs across a network, such as the Internet, is that the recipient may not be able to trust the program. The program may corrupt the user's system, either accidentally through poor programming, or deliberately, in the case of *viruses*. To stop this, Java bytecodes are checked by the verifier before they are executed. We describe these checks in Section 5.5.

5.2.3 The run-time system

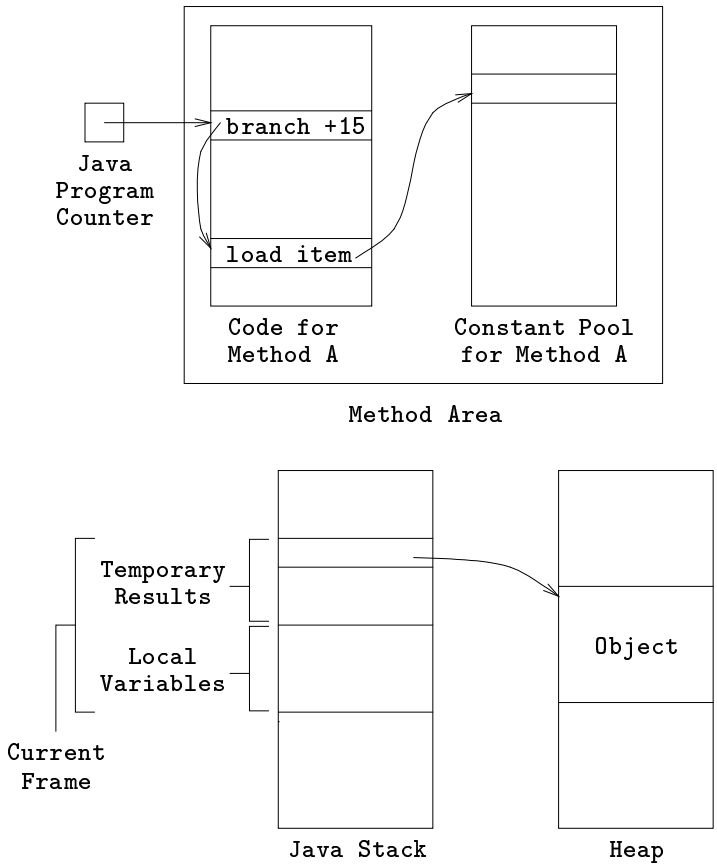


Figure 5.2. *The Java run-time data areas.*

The Java run-time system supports multiple threads of execution. The following data structures are allocated on a per-thread basis:

Java program counter register
: This register holds the address of the instruction currently being executed by a thread.

Java stack
: This is used to store Java virtual machine frames. A frame holds the local variables and temporary results for a method. Each time a method is invoked, a new frame is allocated on the stack. When a method finishes executing, the frame for that method is destroyed.

Note that the space in which the temporary results are stored is known as the *operand stack*. With Java bytecodes, an operator obtains its operands from and pushes the result onto this stack.

The following data structures are shared amongst all threads:

Heap
This is used to store class instances and arrays. Storage space is reclaimed by an automatic management system, such as a garbage collector.

Method area
This area stores per-class structures such as the constant pool, field and method data, as well as the code for the methods.

5.2.4 Run-time checking

Some run-time errors cannot be detected by the verifier, which only performs a static analysis of bytecodes. To help catch these other errors, additional run-time checks are performed. These include common errors such as using an array index that is out-of-bounds and dereferencing a `nil` pointer. If one of these errors occurs during the execution of a Java program, the error is reported rather than causing the Java run-time environment to crash.

Java programs also designed to be run by web browsers such as `Netscape Communicator` [13]. In this case the programs are known as applets rather than applications. Downloading a web page with an applet in it will cause the applet to be executed. In contrast, applications are downloaded and executed in separate phases. Since the downloading and execution of applets is not separate, a user may be unaware that an applet is actually running, especially if no status messages are displayed. An applet's access to a user's system needs to be restricted through an additional level of run-time protection.

5.2.5 The security manager

Bytecode verification and the run-time checks are insufficient protection in themselves, since these checks will not prevent unauthorised access to the user's system resources. For example, an applet may overwrite a user's files. Applets must access the system resources through a *security manager*. The user specifies to the security manager which system resources an applet is allowed to access as well as the mode of access. This is the concept known as the *Java sand-box*. Obviously, an applet must not be able to replace the security manager with one of its own classes, as this would defeat the purpose of the security manager.

Suppose that a user specifies that an applet can only write files to one directory and read files from another. An applet tells the security manager that it is performing a certain operation on a particular file. The security manager checks that this applet is actually allowed to perform this action and either proceeds with the operation, or reports a security violation.

5.3 Java Class File Format

Java class files contain a component known as the *constant pool*. It is a table which contains compile-time constant values and class, method and field references. When combined with the *method* and *field* tables, the constant pool is roughly equivalent to a combination of the *symbol table*, *static initialised data area* and *link table* that are found in native object codes. Thus, the constant pool contains a great deal of information about a Java program.

A Java class file is at the lowest level a stream of bytes. Additional structures are built up out of sequences of these bytes. In the following sections, we examine some of these structures, and the effect they have on obfuscation. The notation in Table 5.1 is used by the following sections for clarity and brevity.

u1	unsigned one byte quantity
u2	unsigned two byte quantity
⟨*entry*⟩	the constant pool index of *entry*

Table 5.1. *Constant pool entry notation.*

5.3.1 Class, method and field references

Type	**Description**
u1	Tag identifying this entry as a class reference
u2	⟨ Name of the class ⟩

Table 5.2. *Class reference constant pool item format.*

When native object codes use libraries, the library code is commonly combined with the compiled program code to make a single executable unit. This is not the case with Java programs since Java library code tends to be system specific. Instead this library code is stored locally on the user's machine as part of the Java run-time system, while the program code is obtained elsewhere, such as downloading it from the Internet. The Java run-time environment loads and links the library class files as required.

Suppose that we want to create a new instance of the class `Thread`. This will appear, for example, as the bytecode instruction `new 5`. The constant pool item at index 5 will be a reference to the class `Thread`, which is a two byte entry. The first byte is the tag that identifies the entry as a class reference. The second byte is the constant pool index of the string '`Thread`', say index 19. The class

reference is used by the Java run-time system to access the information needed to create the new instance of the class `Thread`.

When a class `C` is loaded into memory by the class loader (Section 5.2.1), the classes that it uses must also be loaded. The necessary information is in the form of class references in the constant pool of `C`.

Type	Description
u1	Tag identifying this entry as a method or field reference
u2	⟨ Name of the class being referenced ⟩
u2	⟨ Signature (name and type) of the reference ⟩

Table 5.3. *Method and field reference constant pool item format.*

Method and field references in Java bytecodes use the relevant class name and method or field signature. The Java run-time system uses the class name to find the class data in memory. The signature is used to select a particular method or field. With native object codes, method and field references are invariably implemented using memory pointers (addresses). For example, the C language routine `printf` might appear in native object code as a procedure call to the memory address 35720.

Suppose that we have created a new instance of the class `Thread`. Now we need to invoke the constructor of the class. This might be compiled to the bytecode instruction `invokenonvirtual 9`. The constant pool item at index 9 will be a reference to the constructor method for the class `Thread`, which is a three byte entry. The first byte is the tag that identifies the entry as a method reference. The second byte is the constant pool index of the name of the class. In this case, it will be index 15. The third byte is the constant pool index of the signature of the reference. The Java run-time system uses the method reference to call the required method.

5.3.2 Debugging information

There are many attributes in a Java class file that are used for debugging purposes only. They do not affect how a Java program executes.

The line number table attribute maps an address of a Java bytecode instruction to a line in the source file. This is useful for generating error messages at run-time.

Java local variables are polymorphic. Over a sequence of instructions, a Java local variable may change its type. We discuss this type polymorphism in Section 5.6.1. The local variable table maps a Java local variable to its name, type and range of Java program counter values over which the variable is defined. This

information allows debuggers to determine the value of a local variable during the execution of a method.

5.3.3 Class file limitations

There are many limitations on the format of Java class files. The limitations stem from the fact that 16-bit integers are used to represent item counts and offsets. The most crucial limitation is that the size of a method's code is limited to 65535 bytes. In practice this limitation is not a problem since object-oriented programs are not monolithic pieces of code.

5.4 The Java Bytecode Instruction Set

The Java bytecode instruction set is similar in form to traditional native code instruction sets. The major differences are that:

- Java bytecodes operate on a stack.
- Java bytecodes must pass through a *verification* stage before they are executed (Section 5.5).
- All of the types (including arrays) are described in the bytecode.
- Unlike other native code instruction sets, Java bytecode is strongly typed.

We present a brief description of the Java bytecode instruction set. It is not intended as a complete treatment of the instruction set — we only describe the instructions used by our Java bytecode examples.

object	32-bit object reference
int	32-bit integer
long	64-bit integer
float	32-bit floating point number
double	64-bit floating point number
CP_8	8-bit constant pool index
CP_{16}	16-bit constant pool index
VIdx	8-bit local variable index
CP[I]	value of constant pool entry I
Var[I]	value of local variable I
V_T	value V of type T

Table 5.4. *Bytecode instructions notation.*

We describe bytecode instructions in the following manner, using the notation in Table 5.4 for brevity:

Instruction	Args	Operand Stack		Description
		Before	After	
Xop	...	...	...	$X \in \{\mathrm{i}, \mathrm{l}, \mathrm{f}, \mathrm{d}, \mathrm{a}\}$ ....

The **Args** column lists the arguments of the instruction which are part of the instruction itself, as opposed to being on the operand stack. The **Before** and **After** columns show how the instruction affects the operand stack. In most cases only the topmost items on the stack are affected. Many bytecode instructions are typed — an instruction has separate variants to handle int, long, float, double and reference data types. The instruction variants correspond to the data types in the following manner:

Instruction Variant	Data type operated on
iop	int
lop	long
fop	float
dop	double
aop	reference

5.4.1 Load and store instructions

Instruction	Args	Operand Stack		Description
		Before	After	
Xload_n			Var[n]	$X \in \{\mathrm{i}, \mathrm{l}, \mathrm{f}, \mathrm{d}, \mathrm{a}\}$. Load local variable $n, 0 \leq n \leq 3$.
Xload	n_{VIdx}		Var[n]	$X \in \{\mathrm{i}, \mathrm{l}, \mathrm{f}, \mathrm{d}, \mathrm{a}\}$. Load local variable n.
Xstore_n		V		$X \in \{\mathrm{i}, \mathrm{l}, \mathrm{f}, \mathrm{d}, \mathrm{a}\}$. Store value V in local variable $n, 0 \leq n \leq 3$.
Xstore	n_{VIdx}	V		$X \in \{\mathrm{i}, \mathrm{l}, \mathrm{f}, \mathrm{d}, \mathrm{a}\}$. Store value V in local variable n.

The short versions of the load and store variable instructions (X`load_`n and X`store_`n) can only access a limited range of local variables. However they require fewer bytes to represent and are executed quicker by the Java virtual machine than the long versions (X`load` and X`store`).

5.4.2 Push constant instructions

Instruction	Args	Operand Stack		Description
		Before	After	
bipush	B		B	Push 8-bit signed integer B.
iconst_n			n	Push int constant $n, 0 \leq n \leq 5$.
fconst_n			n	Push float constant $n, n \in [0, 1, 2]$.
ldc_1	I_{CP_8}		CP[I]	Push the item from the constant pool.
ldc2_w	$I_{\mathrm{CP}_{16}}$		CP[I]	Push the long/double from the constant pool.

The short instructions to push constants onto the stack execute quickly. These instructions include `iconst_`n and `fconst_`n. However, these cover a small range of constants. Large integers and floating point numbers have to be loaded from the constant pool.

5.4.3 Class, method and field instructions

Instruction	Args	Operand Stack		Description
		Before	After	
new	$C_{\mathrm{CP}_{16}}$		R	Create a reference R to a new instance of the class C.
getfield	$F_{\mathrm{CP}_{16}}$	O_{object}	V	Push the value V of the field F belonging to the object O.
getstatic	$F_{\mathrm{CP}_{16}}$		V	Push the value V of the static field F.

All of these instructions reference the constant pool. The `new` instruction uses a class reference, while the `getfield` and `getstatic` instructions use a field reference.

Instruction	Args	Operand Stack		Description
		Before	After	
invokenonvirtual invokevirtual	$M_{\mathrm{CP}_{16}}$	$R, A_1, \ldots, A_n$	(V)	Call method M with parameters $A_1, \ldots, A_n$ through object reference R.

Suppose that the object R belongs to the class C. If the actual method that M refers to can be resolved at compile-time, then the `invokenonvirtual` instruction is used. For example, class instance constructors and `final` methods, which cannot be overridden by a subclass of C, will be called via the `invokenonvirtual` instruction. Conversely, if M can be overridden by a subclass of C, then M will be called via the `invokevirtual` instruction. M may return a value V.

5.4.4 Arithmetic instructions

Instruction	Args	Operand Stack		Description
		Before	After	
Xadd		A, B	R	$X \in \{\mathrm{i}, \mathrm{l}, \mathrm{f}, \mathrm{d}\}$. $R = A + B$.
Xmul		A, B	R	$X \in \{\mathrm{i}, \mathrm{l}, \mathrm{f}, \mathrm{d}\}$. $R = A * B$.
Xdiv		A, B	R	$X \in \{\mathrm{i}, \mathrm{l}, \mathrm{f}, \mathrm{d}\}$. $R = A \,/\, B$.
Xrem		A, B	R	$X \in \{\mathrm{i}, \mathrm{l}, \mathrm{f}, \mathrm{d}\}$. $R = A \,\%\, B$.
iinc	$n_{\mathrm{VIdx}}, V_{\mathrm{int}}$			Increment local int variable n by V.
i2d		I_{int}	D_{double}	Convert I from an int into a double.

Arithmetic instructions are strongly typed. If the types of the operands of an instruction are different, the bytecode verifier (Section 5.2.2) will detect this and generate an error.

5.4.5 Branch instructions

Instruction	Args	Operand Stack		Description
		Before	After	
goto	R_{int}			Goto address R.
if_icmplt	R_{int}	$A_{\mathrm{int}}, B_{\mathrm{int}}$		If $A < B$, goto address R.
if_icmple	R_{int}	$A_{\mathrm{int}}, B_{\mathrm{int}}$		If $A \leq B$, goto address R.
if_icmpne	R_{int}	$A_{\mathrm{int}}, B_{\mathrm{int}}$		If $A \neq B$, goto address R.
ifeq	R_{int}	A_{int}		If $A = 0$, goto address R.
ifne	R_{int}	A_{int}		If $A \neq 0$, goto address R.
if_acmpeq	R_{int}	$A_{\mathrm{object}}, B_{\mathrm{object}}$		If $A \neq B$, goto address R.
jsr	R_{int}			Jump to the subroutine at address R.

The operand of a Java bytecode branch instruction (including `jsr`) is actually an offset from the current address. However, we have translated this offset into an absolute address, which makes our Java bytecode examples easier to understand.

5.4.6 Return instructions

Instruction	Args	Operand Stack		Description
		Before	After	
Xreturn		V	empty	$X \in \{\mathrm{i}, \mathrm{l}, \mathrm{f}, \mathrm{d}, \mathrm{a}\}$. Return the value V from a method.
return			empty	Return from a method.
ret	n_{VIdx}			Return from a subroutine to the address in local variable n.

The operand stack is empty after a return instruction in a method is executed. At this stage the Java frame for the method is destroyed because the frame is no longer needed.

5.4.7 Array instructions

Instruction	Args	Operand Stack		Description
		Before	After	
arraylength		R	V_{int}	Push the length V of an array R.
aaload		R, J_{int}	V_{object}	Push the object V stored at index J of array R.
iaload		R, J_{int}	V_{int}	Push the int V stored at index J of array R.

There are different types of Java arrays, for example `int` arrays and `float` arrays. For each type of array, there are separate sets of instructions to load or store an array element. Unlike many other native code instruction sets, Java array elements are not accessed using an offset from a base address in memory. An array index must be non-negative and less than the length of the array or a run-time error will occur.

5.4.8 Other instructions

Instruction	Args	Operand Stack		Description
		Before	After	
athrow		R_{object}	R_{object}	Invoke the exception handler for the exception object R.
checkcast	$I_{\text{CP}_{16}}$	R_{object}	R_{object}	Cast the object to the class given by CP[I] if possible. Otherwise a ClassCastException is thrown.
dup		V	V, V	Duplicate the top of the operand stack.
pop		V		Remove the item at the top of the operand stack.

The `checkcast` instruction causes a run-time exception if an attempt is made to cast an object of class A to an object of class B, which is not a subclass of A. The `dup` and `pop` instructions only operate on data types that can be represented in 32 bits. There is a separate instruction which handles `long` and `double` operands.

5.5 Java Bytecode Verifier

Java bytecodes are intended to be distributed across the Internet among a variety of platforms [18]. The problem with this distribution method is that the recipient may not be able to trust the program. The program may corrupt the user's system, either accidentally through poor programming, or deliberately, in the case of *viruses*. For this reason, Java bytecodes are subjected to rigorous bytecode *verification* [5]. This verification occurs before any potentially unsafe code is executed. The use of static as opposed to dynamic checking allows for greater run-time efficiency. However, it is still necessary to perform some run-time checks to catch errors such as dereferencing a nil pointer and using an array index that is out-of-bounds.

There are certain system-specific routines which cannot be written directly in Java. These routines, which typically deal with *file handling*, are contained within the Java virtual machine itself (Figure 5.1). Such system-specific routines must be implemented in the native code of the particular platform and are termed *native* code by the Java virtual machine. They execute faster than Java bytecodes but it is not possible to subject them to verification. Thus, the secure distribution of upgrades and extensions to the Java virtual machine is a problem. A possible solution involving digitally signed code from trusted software developers was mentioned in Section 2.4.

It is possible in Java to declare classes, methods and fields as *final*, which means that the data and algorithms they contain can never be replaced by creating a subclass. Consider the class `Integer`, which has routines to convert integers into long integers, floating point numbers and strings. Suppose it was possible to create a subclass of `Integer` with *native* code routines to convert integers into pointers. We could then use this subclass to manufacture pointers to memory locations that we are not supposed to access, such as system code areas. Since the relevant routines are *native*, the verifier would not be able to detect this security violation. Thus to maintain the security of Java, the verifier must ensure that *final* classes are not subclassed, and *final* methods are not overridden.

Many of the constraints of the verifier prevent improper manipulation of data types. This ensures that it is not possible to manufacture pointers, such as pointing into the user's operating system. The constraints are:

- If more than one execution path leads to an instruction, the operand stack is the same size and contains the same types of objects before the execution of that instruction.

- No local variable is read from unless it is known to contain a value of an appropriate type.

- Methods are invoked with the appropriate arguments.

- Fields are assigned only using values of appropriate types.
- All local variable uses and stores are valid.
- All opcodes have appropriate type arguments on the operand stack and in the local variables.

5.6 Unusual Bytecode Features

In this section we discuss particular features of the Java bytecode instruction set which make analysing Java bytecode more difficult, namely local variable typing and the implementation of the Java conditional operator.

It is trivially true that every valid Java source code program must compile to a valid Java class file. A valid Java class file must pass through the verifier without causing any errors. However, not every valid Java class file has a direct correspondence to a valid Java source code program. This is because the Java bytecode instruction set supports a richer set of language features than the language Java. These features include goto and subroutine instructions.

5.6.1 Local variable typing

For each method the Java run-time environment provides an array of local variables. Each variable has an associated type and only operations with the appropriate type may be performed on that variable. For example, it is not possible to store a floating point number into a local variable and load it as an integer. However, it is legal to overwrite the contents of a variable with a different typed value. In this case, the variable will change type. Figure 5.3 illustrates this point.

We see in Figure 5.3 that depending on the destination of the conditional branch instruction at address 1, local variable 1 can hold either an integer or a floating point number. If instructions at addresses 4 to 6 are executed, local variable 1 will hold an integer. If the instructions at addresses 9 and 10 are executed, local variable 1 will hold a floating point number. Thus, local variable 1 cannot be given a consistent type when the instruction at address 11 is about to be executed.

5.6.2 The conditional operator

The conditional operator is not unique to Java, and in fact is inherited from the language C. The form of the conditional operator is

boolean expression ? $expression_1$: $expression_2$

If *boolean expression* is true, then the value of the expression above is $expression_1$. Otherwise, the value is $expression_2$. The types of $expression_1$ and $expression_2$

Java Source

```
public static void TypeChange(int i) {
   if (i != 0)
      int j = 1;
   else
      float k = 0.0;
   int j = 2;
}
```

Java Bytecode

#	Instruction	Comment	Type of Local Variable 1
0	`iload_0`	load local variable 0 (i)	-
1	`ifeq 9`	compare with 0 and if equal, goto address 9	-
4	`iconst_1`	push int constant 1	-
5	`istore_1`	store in local variable 1 (j)	`int`
6	`goto 11`	goto address 11	`int`
9	`fconst_0`	push float constant 0	-
10	`fstore_1`	store in local variable 1 (k)	`float`
11	`iconst_2`	push int constant 2	-
12	`istore_1`	store in local variable 1 (j)	`int`

Figure 5.3. *An example of a local variable changing type.* `i` *is local variable 0.* `j` *is local variable 1 in instructions 5 and 12.* `k` *is local variable 1 in instruction 10.*

must be the same. An example of code that uses the conditional operator is in Figure 5.4.

Note that in Figure 5.4, the conditional operator is implemented by the bytecode instructions between addresses 6 and 18.

In order to apply control flow obfuscations, we need to know which bytecode instructions correspond to a particular statement in the source code. This allows us to avoid inserting a branch that can jump into the middle of a Java statement. Otherwise the modified code would be rejected by the Java verifier. The code would violate the constraint that all the execution paths that lead to a particular point must have operand stacks with the same number and type of items on them. For example, in Figure 5.4 when the bytecode instruction at address 19 is about to be executed, the operand stack contains two integers. This is true no matter which instruction was executed after the conditional branch instruction

Java Source

```
① int i, j, k;
② i = 1;
③ j = 0;
④ k = 7 + ((i == j)?(9 * i):j);
```

Java Bytecode

#	Instruction	Comment	Operand Stack Size
0	iconst_1	push int constant 1	1
1	istore_1	store in local variable 1 (i)	0
2	iconst_0	push int constant 0	1
3	istore_2	store in local variable 2 (j)	0
4	bipush 7	push int constant 7	1
6	iload_1	load local variable 1 (i)	2
7	iload_2	load local variable 2 (j)	3
8	if_icmpne 18	compare and if not equal, goto address 18	1
11	bipush 9	push int constant 9	2
13	iload_1	load local variable 1 (i)	3
14	imul	multiply integers	2
15	goto 19	goto address 19	2
18	iload_2	load local variable 2 (j)	2
19	iadd	add integers	1
20	istore_3	store in local variable 3 (k)	0

Figure 5.4. *An example of the Java conditional operator. In line ④ of the source code,* `i` $\neq$ `j`*, so* `k` *is assigned the value* `7 + j`*. In the bytecode,* `i`*,* `j` *and* `k` *are local variables 1, 2 and 3 respectively.*

at address 8.

The standard algorithm for grouping a sequence of instructions into basic blocks relies on identifying which instructions are leaders (see Aho et al. [2], pg. 528). An instruction is a leader if:

- it is the first instruction in the sequence, or
- it is the destination of a branch instruction, or
- it follows a branch instruction.

The instructions are partitioned into basic blocks that start with a leader and contain no other leaders. All of the instructions which follow this leader will be in a single basic block. A basic block includes all of the instructions up to the instruction before the next leader. Normally, a basic block contains instructions that represent one or more complete statements from the original source code. Hence basic blocks can help to determine which instruction addresses can be used as the destination of a branch.

The basic block grouping of instructions will not work if the source code uses conditional operators. Suppose that the basic block algorithm is used to analyse the Java bytecode in Figure 5.4. The bytecode is divided into four basic blocks (Figure 5.5 (a)). The instructions which implement the conditional operator are split between basic blocks 1, 2 and 3, so it would not be valid to insert a branch to the start of basic block 2.

We observe that the operand stack is empty before and after the execution of the instructions that represent a statement. By keeping track of how each instruction affects the operand stack size, we can partition a sequence of instructions into statement sized basic blocks. This modified basic block algorithm is given in Section 7.4.2. In Figure 5.5 (b), we see that the instructions that implement the conditional operator on line ④ in Figure 5.4 are placed into basic block 3.

Basic Block	Start Address	End Address
1	0	8
2	11	15
3	18	18
4	19	20

(a)

Basic Block	Start Address	End Address	Original Line
1	0	1	②
2	2	3	③
3	4	20	④

(b)

Figure 5.5. *Comparison of the standard and modified basic block algorithms. The (a) standard and (b) modified basic block algorithms are applied to the bytecode in Figure 5.4.*

5.6.3 Goto instructions

The language Java can only express programs that can be represented by *reducible* flow graphs (Section 4.3.2), since it has no `goto`-statement. The main feature of reducible flow graphs is that a loop has a single entry point — there are no jumps into the middle of a loop. In contrast, the Java bytecode instruction set does have a `goto` instruction. Hence Java bytecodes can express arbitrary control flow graphs (Figure 5.6).

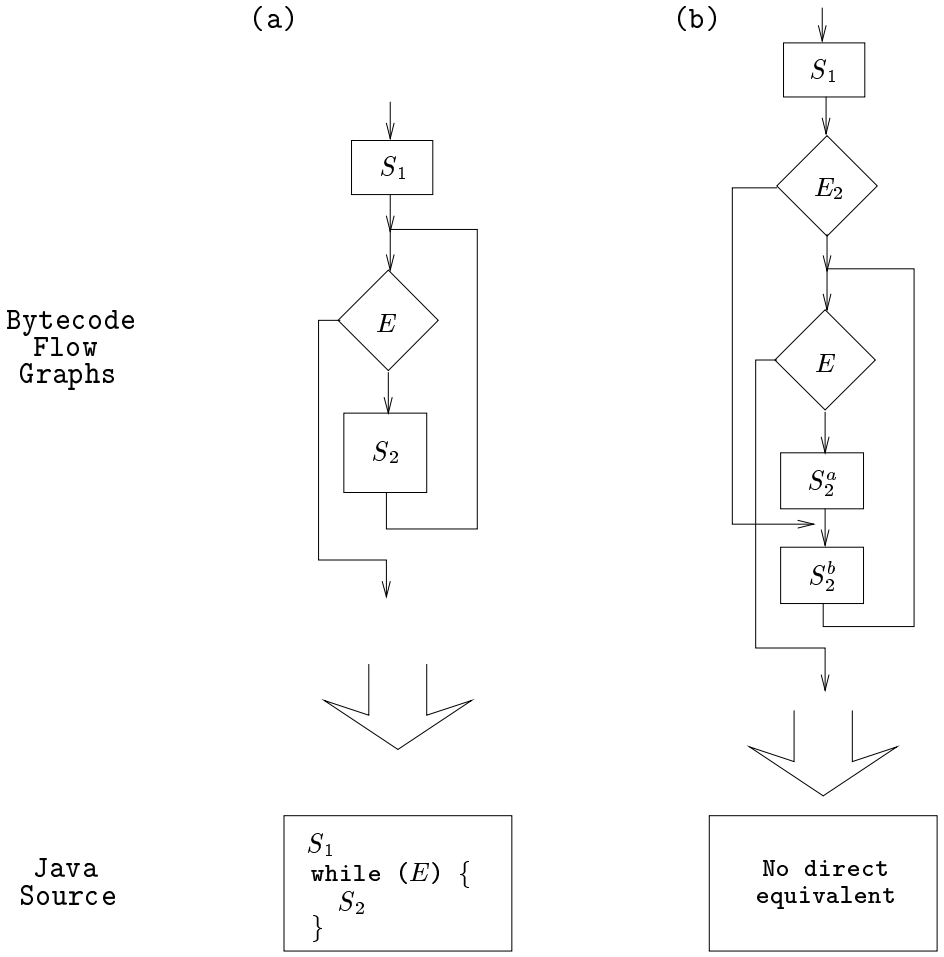


Figure 5.6. (a) *Reducible and* (b) *Non-reducible control flow graphs. A normal loop is in (a), while there is a jump into the middle of the loop in (b).*

5.6.4 Subroutine instructions

The bytecodes `jsr` (jump to subroutine), and `ret` (return from subroutine) are used to implement the `try-finally` statement. An example of a `try-finally` statement is in Figure 5.7 and the corresponding bytecode is in Figure 5.8.

The instructions representing the statements in a `finally` clause are placed into a subroutine. This avoids having to insert the `finally` clause code in three separate places: after the `try` clause code and after both exception handlers. Thus, the overall size of a method is reduced by replacing repeated code with a subroutine.

Java Source

```
public static void foo(int bar) {
  try
     { if (bar == 1) throw new Error(); }   ⇐ ①
  catch (Error e)
     { System.err.println(e); }             ⇐ ②
  finally
     { System.out.println(bar); }           ⇐ ③
}
```

Figure 5.7. *An example of the* `try-finally` *statement. If an* `Error` *exception is thrown during the execution of the* `try` *block (point ①), the* `catch` *block (point ②) is executed, followed by the* `finally` *block (point ③). If the exception is not raised, then after the* `try` *block has finished executing, the* `finally` *block is executed.*

5.7 Discussion

In the following sections, we will discuss how the features of Java mentioned in the previous sections hinder or aid the creation of obfuscating transformations. Note that most of the transformations that are mentioned in this section are rather simple when compared to those discussed in Chapter 4.

5.7.1 The class file format

Unlike native object codes, the names of classes, methods and fields are essential to the operation of the Java virtual machine, as mentioned in Sections 5.2.1 and 5.3. Hence it is not possible to "strip" Java bytecodes of this information. Determining the classes, methods and fields used by a Java class file is therefore easy. However, it is possible to reduce the amount of information in the names that is available to a human reader. This transformation is known as identifier scrambling (Section 3.3.1).

Consider a Java class `MyClass` which is defined in a class file named `MyClass.class`. Suppose that we want to scramble the identifiers in this class. We examine the constant pool for the strings representing the name of the class and the names of the fields and methods defined by this class. We can replace these strings with different names, while recording the name substitutions that are performed. The references in the bytecode will be unaffected, since the constant pool indices remain unchanged. Note that we must also change the name of the class file. So if we rename the class `MyClass` to `x14fgh`, the class file must be renamed as `x14fgh.class`

Java programs can be composed of more than one class file. If the names in the class `MyClass` are changed (for the purpose of scrambling identifiers), we

Java Bytecode

#	Instruction		Comment
0	`iload_0`		load local variable 0 (bar)
1	`iconst_1`		push int constant 1
2	`if_icmpne`	24	compare and if not equal, goto address 24
5	`new`	3	create new object of class Error
8	`dup`		duplicate top of operand stack
9	`invokenonvirtual`	8	call Error constructor
12	`athrow`		throw Error exception
	Exception handler (Error class)		
13	`astore_3`		save exception in local variable 3 (e)
14	`getstatic`	11	push System.err field reference
17	`aload_3`		load local variable 3 (e)
18	`invokevirtual`	9	call System.err.println
21	`goto`	24	goto address 24
	Call finally clause subroutine		
24	`jsr`	34	jump to subroutine at address 34
27	`return`		return from method
	Exception handler (other classes)		
28	`astore_1`		save exception in local variable 1
29	`jsr`	34	jump to subroutine at address 34
32	`aload_1`		load local variable 1
33	`athrow`		throw exception
	Finally clause subroutine		
34	`astore_2`		store return address in local variable 2
35	`getstatic`	7	push System.out field reference
38	`iload_0`		load local variable 0 (bar)
39	`invokevirtual`	6	call System.out.println
42	`ret`	2	return to address saved in local variable 2

Figure 5.8. *Compiled bytecode for the Java source code in Figure 5.7.* `bar` *is local variable 0 and* `e` *is local variable 3.*

must alter the classes that use `MyClass`. This will preserve the linking between classes. To do this, we search the constant pools of classes for a class reference to `MyClass`, as well as for references to methods and fields defined by `MyClass`. The same name substitutions performed on the class `MyClass`, must also be applied to the references we have found in the other classes.

The *JavaBeans* specification [48] defines a set of standard software component

interfaces for Java. This allows a user to create an application out of components downloaded from different vendors. It is not possible to globally change identifier names in an application which is made of JavaBeans components. This is because some of the identifiers are used to interface to a vendor's component. The best we can do is scramble the identifiers which are local to the user's application.

Due to limitations in the class file format, the amount of code that can be added by obfuscating transformations is restricted. For example, the inlining transformation (Section 3.3.3) could cause a method to exceed the code size limit. This can be partially offset if outlining is used in conjunction with inlining.

5.7.2 The bytecode verifier

The analysis of Java class files performed by the verifier is conservative in nature. For example, it is not possible to have a branch to a non-existent instruction number in the bytecode, even if it can be determined at compile-time that the branch is never taken. Thus, we have to careful that when inserting bogus code we do not cause a violation of the verifier's constraints.

Languages like C, which do permit pointer operations, can be difficult to understand (from a human's point of view) and analyse (from a machine's point of view). Unfortunately for the obfuscator designer, pointer arithmetic is not permitted in Java. The verifier will report an error if a non-object reference operator is used on an object reference. There are also no instructions to convert between object references and other data types such as integers. Thus, this area of obfuscation is not open to Java programs.

5.7.3 Applets and applications

The run-time environments for Java applets and applications are different in nature. This affects the types of transformations that can be performed on applets as opposed to applications. For example, applets are not allowed to create their own class loader. Otherwise an applet could bypass the security manager which controls access to the resources on a user's machine. An applet would then have unlimited access to the user's machine. For example, it could corrupt a user's files.

With Java applications, it is possible to create a class loader which instead of reading a class file from a disk or from across a network, obtains it directly from memory. This would allow code to be generated at run-time, foiling static analysis of an application's class files.

5.7.4 The bytecode instruction set

The code generated for the conditional operator leaves items on the operand stack when a conditional branch is executed. This complicates analysis and hinders obfuscation. An obfuscator must keep track of the contents of the operand stack if it attempts to reorder blocks of instructions, so that the contents of the operand stack are used correctly.

Local variables can change their type which makes analysis more difficult. This is why Java class files contain optional debugging information that maps local variables to their types (Section 5.3.2). An important stage during the analysis of Java class files by an obfuscator is determining the types of the local variables in a method (Section 7.5.1).

The `goto` and `subroutine` instructions may be used in the construction of high-level language-breaking transformations, since the Java language does not have `goto`- or `subroutine`-statements. It is possible to generate bytecodes that use subroutines unrelated to a `finally` clause. Chapter 8 has sample outputs from decompilers which attempt to translate such a subroutine.

5.8 Summary

In this chapter, we have seen how the features of the Java virtual machine both hinder and help obfuscation. In the next two chapters, we will examine the implementation of our Java obfuscator.

CHAPTER 6

Overview of the Java Obfuscator

"Our life is frittered away by detail . . . simplify, simplify."

– Henry David Thoreau

6.1 Introduction

This chapter provides an overview of our Java obfuscator, which is designed to operate on Java version 1.0.2 class files, as implemented in Sun's JDK 1.0.2. We describe the options provided by our Java obfuscator and examine the advantages and disadvantages of the implementation language. We also provide a high-level description of the steps involved in obfuscating a program.

6.2 Using the Java Obfuscator

The Java obfuscator is invoked by typing the following command at the UNIX[1] C-shell prompt:

`jo` *flags*

where *flags* is a list of flags chosen from the items below:

- **-file** *file_name*
  Specifies the main class file. The -file prefix is necessary if flags precede the main class file name.

[1] UNIX is a trademark of Bell Laboratories.

- **-noobf** *patterns*
  Specifies the names of the class files that must not be obfuscated. For example '`-noobf myapp/Main libs/`' would mean that the class `myapp/Main` and all classes from the package `libs` would not be obfuscated. The default pattern is '`java/`', which prevents any Java library routines from being obfuscated.

- **-maxcost** *cost*
  The maximum time-space cost that the user is prepared to have incurred by transformations. The default cost is 100. We explain this value in Section 7.6.4.

- **-obflevel** *level*
  The obfuscation level required by the user. The default level is 100. We also explain this value in Section 7.6.4.

- **-path** *path_names*
  Specifies the directories that are to be searched for classes. The default path is '.', which is the current directory.

- **-preservename** *names*
  The user class, method and field names that should not be scrambled. A name is of the form `class_name`, `class_name@field_or_method_name` or `@field_or_method_name`. The default names are '`@main`', '`@<init>`', '`@<clinit>`', '`@<finalize>`', '`@run`'. We explain the significance of these names at the end of this section.

- **-priority** *pri_list*
  The names of classes and methods, which are paired with their obfuscation priorities. This option allows the user to obfuscate the most important classes and methods of the application. There is currently no way to target specific parts of a method's code.

  The source code objects that are not in *pri_list* will be assigned a lower priority than those that are in the list. The default is to assign equal obfuscation priorities to all source code objects.

- **-y or -n or -m** *obfuscation_tags*
  Specifies obfuscations that, respectively, must, must not or may be performed. This option is useful for testing purposes.

  Obfuscation tags are selected from:

  insertBogusBranch, insertDeadCode, changeToSubroutine, publicise, removeDebugInfo, scrambleIdents, all.

'`all`' selects all obfuscations. The obfuscation tags are processed from left to right. For example, '`-n all -y publicise`' will disable all but the publicise obfuscation. The default is '`-m all`'.

We describe the transformations which each of these tags corresponds to in the next section.

Certain methods defined by a class replace methods which are defined in the system libraries. They are said to *override* the system-defined methods. The `-preservename` flag is necessary to ensure that the names of these overriding methods are not changed, which is essential because methods are invoked by name in Java. To keep the command line interface to our obfuscator as simple as possible, we decided to identify these special methods by name only but forget about their parameters. This means, for example, that both of the methods below

```
void main(String argv[])
int main()
```

will not have their names scrambled by our Java obfuscator. In the interests of making programs easy to understand during development, programmers should avoid using the names of special methods for their own routines. Hence the limitation we have imposed on our obfuscator should not be a problem in practice.

By default, our obfuscator will not scramble the names of the following special methods:

- `<init>`
  The methods that are used internally by the Java run-time system to implement instance constructor methods of a class have the name '`<init>`'. When a new instance of a class is created, the constructor for that class is invoked.

- `<clinit>`
  This method is used internally by the Java run-time system to implement a class initialiser, which is invoked when the class is loaded into memory.

- `void main(String argv[])`
  The Java run-time system invokes this method when a Java program is first executed. The `argv` array is used to pass parameters to the program.

- `void run()`
  This method is used by a class to define the main body of code that is intended to be executed as a thread.

6.2.1 Obfuscations provided

Obfuscating transformations are identified by a tag. The tags of the implemented transformations are defined as follows and their qualities are summarised in Ta-

ble 6.1:

- **insertBogusBranch**
  This transformation disguises the control flow of a method by adding a branch to the code. An opaque predicate is used to ensure that the branch, which may be either backwards or forwards, is never taken. The code for this transformation is given in Appendix A.1. Figure 4.11 illustrates how this transformation is applied to code.

- **insertDeadCode**
  This transformation disguises the control flow of a method by adding code that will never be executed. This is achieved by using an opaque predicate to select the correct code to execute. Figure 4.10 demonstrates the effect of this transformation.

- **changeToSubroutine**
  This transformation alters the control flow of a method by placing some code in a subroutine (Section 5.6.4). The transformation makes the code slightly harder for a human reader to understand and is easily undone by inlining the subroutine. That is, the transformation has low potency and weak resilience. Since Java bytecode but not source code has subroutines, this transformation is unstealthy. The jump to and return from subroutine instructions added to the transformed code have little overhead, so the transformation cost is cheap.

- **publicise**
  The access level scheme in Java helps programmers to avoid breaching module abstractions. For example, suppose that a class defines a private field A. If a method in another class attempts to access A, an `IllegalAccessError` occurs causing the program to halt. However, a correctly written program will not cause such errors.

  The publicise transformation takes advantage of how the Java access level scheme operates by changing the access levels of classes, methods or fields to `public`. Since the `public` access level does not restrict access, it provides no abstraction information to a programmer. This transformation affects the control aggregation of a program — how the methods are divided into `public` interface routines and `private` implementation routines.

  The transformation has low potency because the access level of a method is not vital in determining whether the method is an interface or an implementation routine. However, resilience is one-way because information is removed from the program. It may not be obvious that access levels are altered, so this transformation is stealthy. The access level of a method has no effect on its execution time, so the cost of the transformation is free.

- **removeDebugInfo**
  This transformation eliminates debugging information (Section 5.3.2) from a method. Debugging information is unnecessary for the execution of the program, thus this is a layout transformation. Debugging information is useful to a debugger but is less useful to a human, so this transformation has low potency. Information is removed, so resilience is one-way. It is not possible to tell if a method originally possessed debugging information, hence this transformation is stealthy. Removing debugging information has no effect on the execution time of a program, so the cost of this transformation is free.

- **scrambleIdents**
  Java uses names to reference classes, fields and methods (Section 5.3.1). However the names do not have to be meaningful to a human reader. The scramble identifiers obfuscation replaces the original class, field and method names in a program with randomly generated names. Using randomly generated names makes this transformation unstealthy. Instead, it is possible to rename identifiers to plausible but "wrong" names. For example, a variable called '`PreviousNode`' could be changed into '`NextNode`'. An identifier contains a lot of information for a human, so this transformation has medium potency. Information is removed from the program, so resilience is one-way. The cost is free because the execution time of a program is unaffected.

Tag	**Kind**	**Potency**	**Resilience**	**Stealth**	**Cost**
insertBogusBranch	Control computation	Depend on the quality of the opaque predicate used			
insertDeadCode	Control computation	Depend on the quality of the opaque predicate used			
changeToSubroutine	Preventive	low	weak	unstealthy	cheap
publicise	Control aggregation	low	one-way	stealthy	free
removeDebugInfo	Layout	low	one-way	stealthy	free
scrambleIdents	Layout	medium	one-way	unstealthy	free

Table 6.1. *The qualities of the transformations provided by the obfuscator.*

The opaque predicates that the **insertBogusBranch** and **insertDeadCode** transformations depend upon are stored in a library of opaque predicates. This makes it easy to add more opaque predicates to our obfuscator and also helps

when new types of opaque predicates are being implemented. See Appendix B for details of the opaque predicate library.

Obfuscating transformations are also implemented using a module system, which facilitates adding new transformations to our obfuscator. See Appendix A for details of this module system. Appendix A.1 presents an example of an obfuscating transformation module.

6.2.2 Command line example

We present an example of invoking the obfuscator from the UNIX command line and explain the actions of the flags.

```
jo Example.class
   -maxcost 1
   -obflevel 1
   -path /usr/local/lib/java/classes
   -n all
   -y scrambleIdents
```

'`Example.class`' is the main class file of the Java program which is being obfuscated. The maximum run-time penalty that can be incurred by transformations is 1 and the obfuscation level of the program is 1. These two settings ensure that only a single transformation can be performed on the program. Section 7.6.1 explains why this is the case.

The obfuscator searches for classes used by the program in the current directory and in the directory '`/usr/local/lib/java/classes`'. Note that searching the current directory for the required class files is the default behaviour.

The `-n all` and `-y scrambleIdents` flags allow the obfuscator to apply only the scramble identifiers transformation to the program.

6.3 Implementation Language

The language used to implement the obfuscator is the SICStus[2] implementation of Prolog, a logic programming language, for the DEC Alpha[3]. It was chosen for the following reasons:

- Prolog has built-in lists, structures (tuples), memory management and garbage collection.
- A built-in *Definite Clause Grammar* (DCG) notation, which is convenient for expressing the grammar rules used by our Java bytecode parser and unparser.

[2] ©1995 Swedish Institute of Computer Science (SICS).
[3] DEC Alpha is a trademark of Digital Equipment Corporation.

- Prolog routines may be called in a reverse fashion. That is, inputs and outputs may be swapped and the routine still functions correctly. Thus, the Java bytecode parser also acts as a class file constructor.

The disadvantages of Prolog are that:

- Compared to compiled languages like C, Prolog has poor run-time performance, since it is interpreted. This may be partially offset by the use of the built-in SICStus Prolog compiler.
- Recursion must be used instead of iteration when implementing iterative refinement algorithms, as Prolog does not possess appropriate iteration constructs.
- The weak type system makes debugging difficult. A Prolog routine will simply fail if its parameters have the wrong type.
- Accessing global information is awkward. The information needs to be stored into and retrieved from an internal database. This is inefficient since searching is required to find data, rather than a simple pointer dereference as with a language like C.

The advantages of using Prolog as an implementation language outweigh the disadvantages. The key disadvantage of Prolog is its poor run-time performance compared to compiled languages, which we believe is not a major issue. A developer will obfuscate an application once before distributing it. The time taken by this obfuscation step is insignificant compared to the total development time of the product, and thus will not cause production delays.

Efficiency is not as important as correctness when constructing a prototype. It is hard to debug an obfuscated program, which is similar in difficulty to debugging optimised code. In some cases it is more difficult to debug an obfuscated program because some obfuscating transformations, like the insert dead code transformation, introduce code containing a deliberate bug. The task of debugging is made more difficult because we now need to distinguish between these intentional bugs and those bugs generated because the obfuscator itself contains a bug. Thus, it is vital that the obfuscator's code is correct.

The choice of a programming language can affect how correct a program is. The easier it is to implement an algorithm, less errors should be made. Many analysis algorithms used by obfuscators and optimizers are symbolic in nature, which fits in well with Prolog. It is much easier to implement these algorithms in a logic programming language like Prolog than in imperative languages like C.

We might consider that the algorithms used in our obfuscator are proprietary. We therefore want to protect how our obfuscator operates. In this case a slow obfuscator is harder to reverse-engineer than a fast one. If a long time is required for output to be generated, analysing input and output data will not be a feasible strategy for reverse-engineering.

6.4 Obfuscator Structure

The obfuscation of a Java application involves the stages in Figure 6.1, which will be discussed in greater depth in the next chapter.

1. Parse the Java class files that make up the application (Section 7.3).
   The information in Java class files is converted from bytes into Prolog structures in memory.

2. Build an internal representation (Section 7.4).
   The class inheritance hierarchy of the application is built and a control flow graph of each method's code is generated.

3. Construct mappings (Section 7.5).
   A static analysis of the application is used to construct mappings that assist in determining which obfuscations to apply to particular parts of the application. This is similar to the analysis used by code optimizers.

4. Apply obfuscating transformations to the application (Section 7.6).
   Using the mappings constructed in stage 3, obfuscations are selected and applied to the application.

5. Reconstitute the application (Section 7.7).
   The Prolog structures that represent the application are converted back into Java class files.

6.5 Discussion

We believe that the command line interface is a flexible method of using our obfuscator. However, it is not the simplest interface to use, and ideally there should be a graphical user interface (GUI). This GUI would then communicate with the command line interface. Such a GUI is under development.

The `-preserve` option in the command line is awkward to use. Our obfuscator should automatically be able to determine which methods in the Java system libraries are overridden by a program. However, there are a large number of methods that could be overridden. Consequently a large amount of time and memory would be required to perform the analysis.

The obfuscation priority option is also awkward to use and does not allow the user to select parts of a method's code to obfuscate. A partial solution is to use profiling data to avoid heavily obfuscating the most frequently executed parts of the program. This reduces the effect of obfuscation on the run-time performance of the program. We have built a profiler but have not completely integrated it into the obfuscation system.

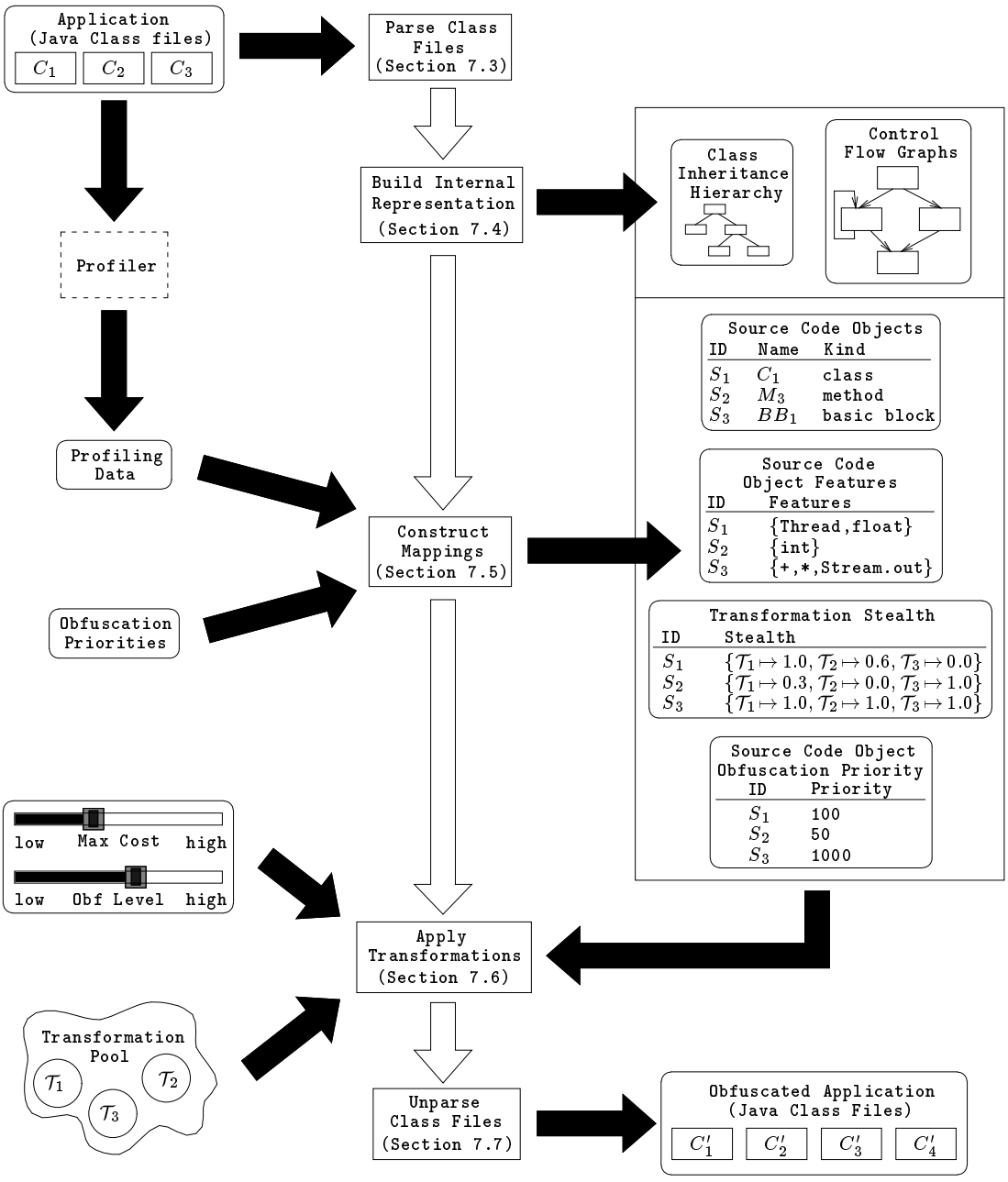


Figure 6.1. *Overview of our Java obfuscator. The profiler is not yet fully integrated into the system.*

The use of Prolog as an implementation language was poor in hindsight. It is a fairly good language for prototyping because list handling is built-in and lists are a fundamental data structure used by the obfuscator. However, the run-time performance of Prolog is poor when compared to imperative languages like C. Also, the weak type system of Prolog makes it a difficult language for large multi-person projects because errors due to misusing types are hard to locate.

6.6 Summary

This chapter has provided a top-level description of our Java obfuscator. In the next chapter, we will examine the obfuscator in further detail.

CHAPTER 7

Implementation of the Java Obfuscator

"Give us the tools, and we will finish the job."

– Winston Churchill

7.1 Introduction

This chapter continues the description of the Java obfuscator. We describe the algorithms used, the various design decisions made, and the problems encountered. We first examine the concept of a source code object, which allows us to specify part of a program to obfuscate. The following sections then deal with the successive stages in the obfuscation of a program.

7.2 Source Code Objects

Some of the obfuscating transformations implemented by our Java obfuscator operate on parts of a program, rather than the entire program. To allow our obfuscator to select specific parts of a program to transform, we need to define the notion of a source code object:

Definition 9 (Source Code Object) A Java application consists of the following source code objects:

- the application itself
- the classes defined by the application
- the fields of each class

- the methods of each class
- the basic blocks of each method

□

Source code objects are identified by the tags in Table 7.1, which are Prolog structures.

Kind	**Tag**
Program	program(Files)
Class	class(File)
Field	field(File@Name:Sig)
Method	method(File@Name:Sig)
Basic Block	bb((File@Name:Sig)/BBNum)

Table 7.1. *Source code object tags.* `Files` *is a list of file names of the classes in the program.* `File` *is a class file name.* `Name` *is a field or method name.* `Sig` *is a field or method signature.* `BBNum` *is a basic block number.* `@`, `:` *and* `/` *are Prolog operators which are used to separate the components of a tag.*

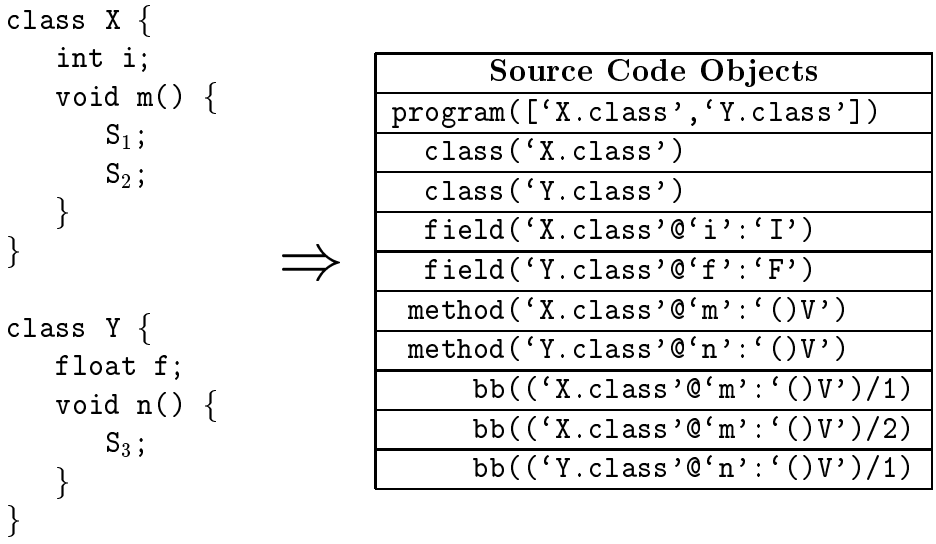


Figure 7.1. *Source code objects example.*

An example of source code objects that correspond to a Java program is given in Figure 7.1. The signatures `'I'` and `'F'` correspond to the types `int` and `float`. The signature `'()V'` denotes a method that has no parameters and has no return value.

7.3 Parsing the Java Class Files

This stage was written by Dr. Christian Collberg. It uses the built-in DCG notation of SICStus Prolog. The parser converts the information in a class file into Prolog structures and stores this information in the internal Prolog database. The obfuscator processes Java class files rather than source files since this means that there is no need for semantic analysis. Furthermore, certain obfuscations can only be applied at the object code level, for example creating non-reducible flow graphs (Section 5.6.3). Obfuscations such as removing comments and formatting (Section 3.3.1) occur automatically when the Java source files are compiled into bytecodes.

There are translators from many languages (including Ada and Scheme) to Java source or bytecode [49]. This means that our obfuscator can be applied to a variety of languages by utilising the appropriate translator.

7.4 Building an Internal Representation

Our obfuscator builds the following internal representations of an application — an inheritance hierarchy of classes and a control flow graph for each method. Additional information, such as the name and signature of a method, is left in the constant pool.

We describe the algorithms used to create these structures next. As new obfuscations are added, further analyses and structures need to be included in the obfuscator. For example, data obfuscations require us to keep track of how variables are related.

7.4.1 Class inheritance hierarchy

Java is an object-oriented language, with a single inheritance scheme. Thus, a tree suffices to represent the class inheritance hierarchy, with the class `Object` at the root. The actual implementation uses a "parent-pointer" tree representation: each class is paired with its parent and stored in a list. Two routines extract information from this structure:

- `childrenOf` (SuperClass, Children)
  The input to this routine is a name of a class in *SuperClass*. The output is a list of class names in *Children*. These classes have the class *SuperClass* as a parent.

- `parentsOf` (Child, Parents)
  The input to this routine is a name of a class in *Child*. The output is a list of class names in *Parents*. These classes have the class *Child* as a child.

Note that classes in Java will have only a single parent. This is not true for languages with multiple inheritance.

7.4.2 Control Flow Graph

A control flow graph (CFG) represents the control flow in a method. This information is required to apply control flow obfuscations to a program. In Section 5.6.2 we discussed why the standard control flow graph building algorithm (see Aho et al. [2], pg. 528) was unsuitable for languages that contain a C-style conditional operator.

The algorithm to build a control flow graph is in Figure 7.2. $\mathcal{M}$ is the method that is being analysed.

```
BuildCFG(M)
    L := FindLeaders(M);
    B := PartitionBasicBlocks(M, L);
    B' := AddExitBlock(B);
    B'' := AddCFGEdges(B');
    RETURN B'';
```

Figure 7.2. *The control flow graph building routine.*

- `FindLeaders` returns a list of the leaders in a method's code. A leader is the first instruction in a basic block. For all possible execution paths through a method's code, we keep track of the operand stack size as each instruction is executed. Suppose that after the current instruction is executed, the operand stack is empty. Then any instructions that would be executed after the current one are leaders.

- `PartitionBasicBlocks` partitions the instructions in a method's code so that each partition starts with a leader and contains no other leaders.

- `AddExitBlock` adds an exit basic block to the CFG. This basic block is a common exit point, representing the fact that control has passed out of the method.

- `AddCFGEdges` adds edges between basic blocks in the CFG. For each basic block an edge is added from the basic block to its successors. We determine which instructions could be executed after the last instruction in the current basic block is executed. The basic blocks that these instructions belong to are the successors of the current basic block.

With the `AddCFGEdges` routine, certain instructions are handled differently when successors are being determined:

- The `return` instruction causes control to be returned to the invoker of a method.
- The `ret` instruction returns from a subroutine. Java uses subroutines to implement `finally` clauses. Our obfuscator includes an obfuscation to move some of the statements in a method into a subroutine. However, our obfuscator does not alter existing subroutines. Hence our basic block building algorithm does not need to determine subroutine return addresses.
- The `athrow` instruction causes an exception to be thrown. Control is passed to an appropriate handler for the particular type of exception. If the method does not catch the exception, it is passed to the superclass of the current class. Thus, the successor of a basic block containing an `athrow` instruction may be external to the current method, which cannot be easily represented by the method's CFG.

To simplify analysis, basic blocks that contain a `return`, `ret` or `athrow` instruction have the exit basic block as their sole successor. In fact, we decided not to obfuscate a method if it contains exception handlers, since it extremely difficult to ensure that the behaviour of such methods is correct after an obfuscating transformation is applied. Future versions of our obfuscator will remove this restriction.

7.5 Constructing Mappings

The mappings defined below are constructed to assist in determining which source code objects require obfuscation and which obfuscations to apply.

P_s For each source code object S, $P_s(S)$ is the set of language features in S. $P_s(S)$ is used to find appropriate obfuscating transformations for S.

A For each source code object S, $A(S) = \{\mathcal{T}_1 \mapsto V_1, \ldots, \mathcal{T}_n \mapsto V_n\}$ is a mapping from transformations $\mathcal{T}_i$ to values $V_i = \mathcal{T}_{\rangle\,\mathrm{ste}}(S)$, describing the stealth of $\mathcal{T}_i$ when it is applied to S.

I For each source code object S, $I(S)$ is the *obfuscation priority* of S. $I(S)$ describes how *important* it is to obfuscate S. If S contains an important trade secret, $I(S)$ will be high. If S contains mainly "bread-and-butter" code, $I(S)$ will be low.

We will use the trivial Java example in Figure 7.3 to show how these mappings are calculated.

Java Source

```
public class Echo {
   public static void main(String argv[]) {
      int i;
      for (i = 0; i < argv.length; i++)
         System.out.println(argv[i]);
   }
}
```

Java Bytecode

#	Instruction	Comment	Basic Block #
0	iconst_0	push int constant 0	1
1	istore_1	store in local variable 1 (i)	1
2	goto 17	goto address 17	2
5	getstatic 7	push the value of System.out	3
8	aload_0	load local variable 0 (argv)	3
9	iload_1	load local variable 1 (i)	3
10	aaload	load array element	3
11	invokevirtual 5	call System.out.println	3
14	iinc 1,1	increment local variable 1 (i) by 1	4
17	iload_1	load local variable 1 (i)	5
18	aload_0	load local variable 2 (argv)	5
19	arraylength	push the length of the array argv	5
20	if_icmplt 5	compare integers and if first is less than second, goto address 5	5
23	return	return from method	6

Figure 7.3. *Trivial Java code example to demonstrate mapping algorithms.* `argv` *and* `i` *are local variables 0 and 1 respectively. The basic block number is calculated by the algorithm in Section 7.4.2.*

7.5.1 Language features

$P_s(S)$ is the set of language features belonging to source code object S. We use these features to determine which transformation should be applied to S. Currently the language features that we identify are in Table 7.2.

Not all of these features are applicable to a particular source code object. Table 7.3 shows which features apply to which kind of source code object. Source

Feature	Description
kind	Source code object kind = ⟨`program, class, field, method, bb`⟩
`name`	Class, method or field name
`type`	Method parameters or field type
`classes`	Set of class names
`classRefs`	Set of classes used by a class or method
`fields`	Set of fields defined by a class
`fieldRefs`	Set of fields used by a class or method
`methods`	Set of methods defined by a class
`methodRefs`	Set of methods used by a class or method
`operators`	Set of operator and data type pairs
`loops`	Set of loops contained in a method
`locals`	Set of local variables usable on entry to a basic block
`predicates`	Set of predicates remaining to be used by a basic block

Table 7.2. *Language features for Java.*

code objects form a hierarchy (Figure 7.4). The features of a source code object will be the union of the features of the source code objects that are below it in the hierarchy. For example, a class obtains its `classRefs` set by performing a union operation on all the `classRefs` sets of the methods that the class defines.

The `loops` feature for `method` source code objects is determined using the algorithm for constructing natural loops (see Aho et al. [2], pg. 604). The information in the `loops` feature allows our obfuscator to apply obfuscations such as inserting a branch into a loop.

The algorithm for calculating the `locals` feature for basic block (`bb`) source code objects is based on the iterative algorithm for calculating *reaching definitions* (see Aho et al. [2], pg. 625). The `locals` feature allows our obfuscator to generate code that uses the local variables of a method. A local variable needs to be initialised before it is first accessed. Also, the variable must be accessed with an operator which has a matching type. Java local variables have type polymorphism (Section 5.6.1), so the type of a local variable must be determined before this variable can be used. For example, if local variable 1 has the type `float`, we have to use the load float operator to read the value stored in this variable. Consider a basic block B with predecessors $A_1, \ldots, A_n$ in the control flow graph. If a local variable V is known to have been initialised in $A_1, \ldots, A_n$, and the type of V is the same in $A_1, \ldots, A_n$, then it is a member of the `locals` set of B.

Consider the Java code in Figure 7.3. The language features for the method `main` are in Table 7.4.

Kind	Feature	Description
`program`	`classes`	Set of classes that make up the program
	`classRefs`, `fields`, `fieldRefs`, `methods`, `methodRefs` and `operators` are obtained from each class in the program.	
`class`	`name`	Name of the class
	`fields`	Set of fields defined by the class
	`methods`	Set of methods defined by the class
	`classRefs`, `fieldRefs`, `methodRefs` and `operators` are obtained from each method defined by the class.	
`method`	`name`	Name of the method
	`type`	Type of the method
	`loops`	Set of loops in the code of the method
	`classRefs`, `fieldRefs`, `methodRefs` and `operators` are obtained from each basic block of instructions in the method.	
`field`	`name`	Name of the field
	`type`	Type of the field
`bb`	`locals`	Set of local variables usable on entry to the basic block
	`predicates`	Set of predicates that have yet to be used in the basic block. Initialised to all the predicates in the predicate library
	`classRefs`, `fieldRefs`, `methodRefs` and `operators` are obtained from the instructions in the basic block.	

Table 7.3. *The language features inheritance scheme for source code objects.*

7.5.2 Transformation stealth mapping

The stealth of a transformation is highly context sensitive. The definition of stealth in Section 3.4.4 may be refined so that the measure applies to individual source code objects instead of the whole program. Code that would be unstealthy if added to one method, might not be unstealthy if added to another.

Suppose we have a transformation $\mathcal{T}$ which is to be applied to source code object S. To calculate $\mathcal{T}_{\text{ste}}(S)$, we need to determine $P_s(\mathcal{T})$ which is the set of language features that would be added when $\mathcal{T}$ is applied to S. We also need to know $P_s(S)$, which is the set of language features that is used by S. Table 7.3 shows the set of language features and the particular source code object to which they are applicable. With this information we can use the definition for stealth in Section 3.4.4, except that we perform the calculation replacing $P_s(Q)$ by $P_s(S)$.

The transformation stealth mapping $A(S)$ for a source code object S is de-

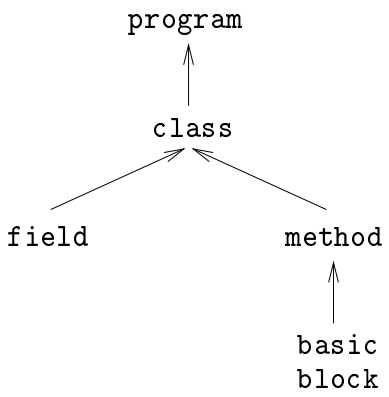


Figure 7.4. *The source code objects hierarchy. The code in a method is composed of basic blocks of instructions. A class can define fields and methods. A program is made up of classes, which form an inheritance hierarchy (Section 7.4.1).*

Feature	Value
`name`	main
`type`	[array(object('String'))]:void
This method has an array of `'String'` objects as its single parameter. There is no return value.	
`loops`	5->3->4
There is a single loop in this method. The loop header is basic block 5 (which tests whether the loop should continue to execute). The loop body consists of basic blocks 3 and 4.	
`classRefs`	none
`fieldRefs`	field('java/lang/System'@out:object('java/io/PrintStream')
This method references the field `'out'`, which is an instance of the class `'java/io/PrintStream'`. This field is from the class `'java/lang/System'`.	
`methodRefs`	method('java/io/PrintStream'@println:([object('String')]:void))
This method invokes the method `'println'`, which takes an object of class `'String'` as its single parameter. There is no return value. The invoked method is from the class `'java/io/PrintStream'`.	
`operators`	const/int, =/int, ++/int
This method uses an assignment and an increment operator. It also pushes an integer constant onto the stack.	

Table 7.4. *Language features of the method* `main` *of the Java code example in Figure 7.3.*

termined by calculating $\mathcal{T}_{\text{ste}}(S)$ for each transformation $\mathcal{T}$.

Consider the example in Figure 7.3. Let the source code object S be the method `main`. Suppose that a transformation $\mathcal{T}$ adds the following language features when it is applied to S:

Feature	Value
`classRefs`	class('java/lang/Thread')
A reference is made to the Java `Thread` class.	
`methodRefs`	method('java/lang/Thread'@'start':([]:void)
The method `'start'`, from the class `'java/lang/Thread'` is invoked.	
`operators`	+/int, ++/int, const/int
The addition and multiplication operators are used. Additionally, an integer constant is pushed onto the stack.	

$|P_s(\mathcal{T})| = 5$, which is the sum of the sizes of all of the sets of language features that $\mathcal{T}$adds to S. The operators `++/int` and `const/int` are present in $P_s(S)$, which means that $|P_s(\mathcal{T}) \setminus P_s(S)| = 3$. So:

$$\begin{aligned}\mathcal{T}_{\text{ste}}(S) &= 1.0 - \frac{|P_s(\mathcal{T}) \setminus P_s(S)|}{|P_s(\mathcal{T})|} \\ &= 1.0 - \frac{3}{5} \\ &= 0.4\end{aligned}$$

7.5.3 Obfuscation priority

Some parts of an application will contain valuable trade secrets, while other parts will be ordinary "bread-and-butter" code. Our obfuscator provides a command line flag that allows the user to determine the relative obfuscation priority for the source code objects in an application (Section 6.2).

Obfuscation priority is inherited according to the hierarchy in Figure 7.4. The program is given the most important obfuscation priority, which ensures that obfuscations that affect the entire program can take place. Unless set differently by the user, fields and methods will have the same obfuscation priority as the class that defines them. Basic blocks that belong to a method will have the same obfuscation priority as that method. There is currently no way to change the obfuscation priority of a basic block, since the division of method code into basic blocks is an internal representation of our obfuscator. It is therefore difficult for the user to convey obfuscation priority information about basic blocks to our obfuscator. This will be fixed when a more powerful GUI is added to our obfuscator.

7.6 Applying Obfuscating Transformations

The top-level loop of our obfuscator has the structure in Figure 7.5. $\mathcal{A}$ is the application undergoing obfuscation.

```
WHILE NOT Done(A) DO
    S := SelectCode(A);
    T := SelectTransform(S);
    A := Apply(T,S);
```

Figure 7.5. *Top level obfuscator loop.*

- `Done` determines when the required level of obfuscation or the maximum acceptable execution time penalty has been reached.
- `SelectCode` returns the source code object with the greatest obfuscation priority.
- `SelectTransform` returns the most appropriate transformation to obfuscate the particular source code.
- `Apply` applies the transformation to the source code object and updates the application accordingly.

We discuss each of these routines in the following sections.

7.6.1 The Done predicate

The `Done` predicate evaluates to true when there is no more remaining obfuscation to be performed or no more execution time penalty can be imposed. That is, the initial obfuscation level and maximum acceptable execution time penalty have been exceeded. The code for the `Done` predicate is:

```
Done(A)
    RETURN (A_obflevel <= 0) OR (A_cost <= 0);
```

- $\mathcal{A}$ is the source code object tag which represents the application.
- $\mathcal{A}_{obflevel}$ is the remaining obfuscation level required by the application.
- $\mathcal{A}_{cost}$ is the remaining execution time penalty that can be added to the application.

$\mathcal{A}_{obflevel}$ and $\mathcal{A}_{cost}$ are initialised to user determined (Section 6.2) values, namely the maximum required obfuscation level and maximum acceptable execution time penalty. Suppose that the obfuscating transformation $\mathcal{T}$ is applied to the application A. $\mathcal{A}_{obflevel}$ is decreased by a value calculated from $\mathcal{T}_{\text{pot}}(\mathcal{A})$ and $\mathcal{T}_{\text{res}}(\mathcal{A})$. $\mathcal{A}_{cost}$ is decreased by $\mathcal{T}_{\text{cost}}(\mathcal{A})$. Section 7.6.4 presents the algorithm used to update $\mathcal{A}_{obflevel}$ and $\mathcal{A}_{cost}$.

7.6.2 The SelectCode algorithm

The initial obfuscation priorities of the source code objects are determined by a command line option (Section 6.2). These values are updated as the source code objects are obfuscated. The source code object tags, paired with their obfuscation priorities, are stored in a priority queue. The priority queue is ordered on descending obfuscation priority, so the source code object with the greatest obfuscation priority is found at head of the queue. If there is more than one source code object with the greatest obfuscation priority value, we choose a random one to obfuscate.

7.6.3 The SelectTransform algorithm

Suppose we have a source code object S that we wish to obfuscate. We want to apply the most appropriate transformation $\mathcal{T}$ to S, which will have maximum potency and resilience for a minimum cost. Given $\mathcal{T}_{\text{qual}}(S)$, we can use the definition for *appropriateness* in Section 3.6 to determine $\mathcal{T}_{\text{app}}(S)$, except that we perform the calculation with $P_s(S)$ instead of $P_s(P)$. An example of using the quality of a transformation to calculate its appropriateness was given in Section 3.6.

The potency, resilience and cost of an obfuscating transformation $\mathcal{T}$ use the abstract scales which are defined in Section 3.4, such as $\langle$*low*, *medium*, *high*$\rangle$ potency. Before calculations using these measures can be performed, these scales need to be given concrete numerical values. Our obfuscator uses the values given in Table 3.3.

If $\mathcal{T}$ does not use an opaque predicate, then potency, resilience and cost of $\mathcal{T}$ can be determined statically and can therefore be stored in the transformation module that defines $\mathcal{T}$. Appendix A discusses the structure of transformation modules. On the other hand, the stealth of $\mathcal{T}$ is calculated at run-time for each source code object. Stealth is a much more context sensitive measure than the other measures. The calculation of stealth was discussed in Section 7.5.2.

If $\mathcal{T}$ uses an opaque predicate P, then the quality of $\mathcal{T}$ will depend upon the quality of P. The other measures will be determined at run-time from P. The information needed to do so is stored in a predicate library. We discuss the structure of predicate libraries in Appendix B.

For each source code object, we keep a priority queue containing obfuscating transformation tags paired with their appropriateness. The priority queue is ordered on decreasing appropriateness values, so the most appropriate transformation to apply to a source code object is found at the head of the queue. If there is more than one transformation with the greatest appropriateness value, we choose a random transformation to apply to the source code object.

7.6.4 Applying an obfuscating transformation

The algorithm in Figure 7.6 applies the transformation $\mathcal{T}$ to a source code object S. $\omega_1, \omega_2, \omega_3, \omega_4$ are constants that may be adjusted to "fine-tune" obfuscator performance.

```
Apply(T,S)
    S := Obfuscate(T, S);
    I(S) := I(S) − (ω1 * T_pot(S) + ω2 * T_res(S));
    A_obflevel := A_obflevel − (ω3 * T_pot(S) + ω4 * T_res(S));
    A_cost := A_cost − T_cost(S);
RETURN A;
```

Figure 7.6. *The apply transformation algorithm.*

- `Obfuscate` applies the transformation T to the source code object S, updating the internal data structures of the obfuscator. See Appendix A.1 for an example of an obfuscating transformation.
- Since S has been obfuscated, its obfuscation priority ($I(S)$) must be decreased. The amount that it is decreased by depends on the potency and resilience of the applied transformation, which are scaled by ω_1 and ω_2 respectively.
- The required remaining obfuscation level of the application ($A_{obflevel}$) is decreased. The amount that it is decreased by depends on the potency and resilience of the applied transformation, which are scaled by ω_3 and ω_4 respectively.
- The acceptable remaining cost that can be incurred by obfuscations ($\mathcal{A}_{cost}$) is reduced by the cost that the transformation incurs.

Our obfuscator uses $\omega_1 = \omega_2 = \omega_3 = \omega_4 = 1$ to give equal weighting to potency and resilience in the calculations. $A_{obflevel}$ and $\mathcal{A}_{cost}$ must be reduced to below zero before the obfuscator stops. The default values of the `maxcost` and

`obflevel` options (Section 6.2) are 100 for both options. Given the values used by our obfuscator in Table 3.3 for the abstract transformation measure scales, we see that the transformations applied by our obfuscator must have a total obfuscation level (potency plus resilience) exceeding 100 and a total cost exceeding 100. This can be achieved in a number of ways:

Number of Transformations	**Potency of each Transformation**	**Resilience**	**Cost**
1	1	100	100
1	100	1	100
10	10	10	10

Of course, not all of the transformations applied by the obfuscator have to have exactly the same potency, resilience and cost.

7.7 Re-constituting the Application

This stage uses the same code as the parsing stage (Section 7.3). The fact that most Prolog routines can be called in a reverse fashion means that it is unnecessary to write a wholly separate class file parser and unparser.

7.8 Discussion

There are advantages and disadvantages in obfuscating Java programs at the bytecode level. The implementation of our obfuscator is simplified because we do not need a semantic analysis stage. By using appropriate translators, it is possible to obfuscate a large number of languages, including Ada and Scheme. The downside in working with Java bytecodes is that the information contained in Java class files needs to be abstracted to a higher level in order to be useful. A large number of the algorithms of our obfuscator are concerned with extracting and processing this information.

We believe that our class inheritance hierarchy scheme is flexible enough to cope with object-oriented languages that have a multiple inheritance scheme. Java has a single inheritance scheme, although it does have *interfaces*. A class which implements an interface must define all of the methods in that interface. This information could be added to our inheritance hierarchy but would make it more complicated. Depending on the obfuscating transformations implemented by an obfuscator, this additional inheritance information may be unnecessary.

The internal mappings calculated by our obfuscator can be extended to handle additional obfuscating transformations. Currently there is room for improvement in the representation of the mappings. For example, the appropriateness mapping defines a priority queue for each source code object. This is costly in terms

of memory. However, this implementation fulfils the requirement that given a source code object S, we can determine the appropriateness of each obfuscating transformation applied to S.

The control flow graph building algorithm is quite complicated and executes slowly. It differs from the standard algorithm (see Aho et al. [2], pg. 528) due to the need to handle the conditional operator correctly. The standard algorithm breaks up the instruction implementing the conditional operator into several basic blocks. The operand stack is not empty on exit from some of these basic blocks. Any basic blocks that are added must therefore ensure that they do not affect the operand stack. We therefore decided to divide instructions into basic blocks in which the operand stack is empty on entry to and exit from a basic block. In hindsight, the extra complexity of the new algorithm was not worthwhile. Instead we should have used the standard algorithm and preserved the contents of the operand stack when we add basic blocks to a method.

Our obfuscator currently does not handle methods with exception handlers. Support for such methods will be present in a future version of our obfuscator.

7.9 Summary

Now that we have examined the implementation of our obfuscator, we are ready to evaluate it. In the next chapter, we will apply our obfuscator to concrete examples of Java code.

CHAPTER 8

Examples

"Example is always more efficacious than precept."

– Samuel Johnson

8.1 Introduction

This chapter presents some examples of the transformations employed by our obfuscator. We also use the decompilers `Mocha` [51] and `WingDis` [12] to attempt to recover the Java source code from the transformed Java bytecode. By measuring the effect the transformations have on selected software complexity metrics, we can gauge their effectiveness. As mentioned in Section 3.4.1 the exact values of these metrics is unimportant — it is the observed change that is of interest.

By applying the following transformations individually to separate code examples, we can demonstrate their effects clearly:

- The change to subroutine transformation
- The insert dead code transformation
- Non-standard code patterns

We then present an example of code that is processed by our obfuscator. Unlike the other three examples, an arbitrary number of transformations is used.

When we present transformed bytecode, the instructions that have been added or changed are marked by an asterisk (*).

8.2 Change to Subroutine

The change to subroutine transformation alters the control flow of a method by replacing some code with a call to a subroutine. This is a high-level language breaking transformation, since Java bytecodes but not Java source code have subroutines (Section 5.6.4).

Java Source

```
public class Test {
   public static double example(int i) {
      double j = 2.0;
      if (i == 1)
         j *= 2.5;
      return i / j;
   }
}
```

Java Bytecode

#	Instruction	Comment
0	`ldc2w 6`	push double floating-point constant 2.0
3	`dstore_1`	store in local variable 1 (j)
4	`iload_0`	load local variable 0 (i)
5	`iconst_1`	push int constant 1
6	`if_icmpne 15`	compare and if not equal, goto address 15
9	`dload_1`	load local variable 1 (j)
10	`ldc2_w 4`	push double floating-point constant 2.5
13	`dmul`	multiply double floating-point numbers
14	`dstore_1`	store in local variable 1 (j)
15	`iload_0`	load local variable 0 (i)
16	`i2d`	convert int to double floating-point number
17	`dload_1`	load local variable 1 (j)
18	`ddiv`	divide double floating-point numbers
19	`dreturn`	return double from method

Figure 8.1. *Code before the change to subroutine transformation is applied.* `i` *and* `j` *are local variables 0 and 1 respectively.*

The obfuscator was invoked with the command:

```
jo Test.class -maxcost 1 -obflevel 1
              -path /usr/local/lib/java/classes
              -n all -y changeToSubroutine
```

The original code in Figure 8.1 was transformed into the code in Figure 8.2.

Java Bytecode

#	Instruction	Comment
0	`ldc2_w 6`	push double floating-point constant 2.0
3	`dstore_1`	store in local variable 1 (j)
4	`iload_0`	load local variable 0 (i)
5	`iconst_1`	push int constant 1
6	`if_icmpne 15`	compare and if not equal, goto address 12
9	`*jsr 17`	jump to subroutine at address 17
12	`iload_0`	load local variable 0 (i)
13	`i2d`	convert int to double floating-point number
14	`dload_1`	load local variable 1 (j)
15	`ddiv`	divide double floating-point numbers
16	`dreturn`	return double from method
17	`*astore_3`	store return address in local variable 3
18	`*dload_1`	load local variable 1 (j)
19	`*ldc2_w 4`	push double floating-point constant 2.5
20	`*dmul`	multiply double floating-point numbers
21	`*dstore_1`	store in local variable 1 (j)
22	`*ret 3`	return to address saved in local variable 3

Figure 8.2. *An example of code being placed in a subroutine.* `i` *and* `j` *are local variables 0 and 1 respectively. Local variable 3 is used to store the subroutine return address.*

The source code recovered by `Mocha` and `WingDis` are presented in Figures 8.3 and 8.4. The source code output by `Mocha` contains `pop` and `endfinalize` statements, which are not part of the Java language. It also fails to show where the subroutine is invoked. The variable `local` is not declared. The source code output by `WingDis` contains a `jsr`-statement to the label `B4`. The `jsr`-statement is not part of the Java language. Furthermore the label `B4`, which is used by `WingDis` to represent the subroutine start address, is not defined. The subroutine that should contain the statement `j *= 2.5` is missing.

The Halstead metric (μ_1 in Table 3.1) is affected by this transformation, since we are adding additional operators and operands to the code. One way

in which the transformed program can be represented as source code is shown in Figure 8.5. Note that we need to add `jsr`- and `ret`-statements to the Java language. These statements jump to and return from a subroutine. The `jsr Subroutine` statement adds 1 operator and 1 operand, while the `ret`-statement adds 1 operator to the program. Hence the μ_1 measure is increased by $2+1=3$. According to this metric, the transformed program has greater complexity than the original.

```
public static double example(int i)
{
   double d;
   d = 2;
   if (i == 1)
   return (double)i / d;
   pop local
   d *= 2.5;
   endfinalize local
}
```

Figure 8.3. *Output from* `Mocha` *for the bytecode in Figure 8.2.*

```
public static double example(
   int int0) {

   double double1;
   obj obj3;

   double1= 2;
   if (int0 == 1)  {
      jsr B4;
   }
   return ((double)int0 / double1);

}
```

Figure 8.4. *Output from* `WingDis` *for the bytecode in Figure 8.2.*

```
public static double example(int i) {
   double j = 2.0;
   if (i == 1)
      jsr Subroutine
   return i / j;

Subroutine:
   j *= 2.5;
   ret;
}
```

Figure 8.5. *Possible source code representing the bytecode in Figure 8.2. Note that* `jsr-` *and* `ret-`*statements have been added to the Java language in order to represent subroutines.*

8.3 Insert Dead Code

The insert dead code transformation disguises the control flow of a method by adding an opaque predicate to the code, which ensures that the dead code is never executed. This transformation is an example of a "Smoke and Mirrors" obfuscation which is a particular kind of control computation transformation.

Java Source

```
public class Test2 {
   public static int combine(int list[]) {
      int temp = 0;
      for (int i = 0; i < list.length - 1; i++)
         temp += list[i] * list[i + 1];
      return temp;
   }
}
```

Figure 8.6. *Source code before the insert dead code transformation is applied.*

The obfuscator was invoked with the command:

```
jo Test2.class -maxcost 1 -obflevel 1
               -path /usr/local/lib/java/classes
               -n all -y insertDeadCode
```

The bytecode in Figure 8.7 was transformed into the bytecode in Figures 8.8 and 8.9.

Java Bytecode

#	Instruction	Comment
0	`iconst_0`	push int constant 0
1	`istore_1`	store in local variable 1 (temp)
2	`iconst_0`	push int constant 0
3	`istore_2`	store in local variable 2 (i)
4	`goto 22`	goto address 22
7	`iload_1`	load local variable 1 (temp)
8	`aload_0`	load local variable 0 (list)
9	`iload_2`	load local variable 2 (i)
10	`iaload`	load array element
11	`aload_0`	load local variable 0 (list)
12	`iload_2`	load local variable 2 (i)
13	`iconst_1`	push int constant 1
14	`iadd`	add integers
15	`iaload`	load array element
16	`imul`	multiply integers
17	`iadd`	add integers
18	`istore_1`	store in local variable 1 (temp)
19	`iinc 2,1`	increment local variable 2 (i) by 1
22	`iload_2`	load local variable 2 (i)
23	`aload_0`	load local variable 0 (list)
24	`arraylength`	load length of array
25	`iconst_1`	push int constant 1
26	`isub`	subtract integers
27	`if_icmplt 7`	compare integers and if first is less than second, goto address 7
30	`iload_1`	load local variable 1 (temp)
31	`ireturn`	return int from method

Figure 8.7. *Bytecode for the source code in Figure 8.6.* `list`, `temp` *and* `i` *are local variables 0, 1 and 2 respectively.*

The source code recovered by `Mocha` and `WingDis` are presented in Figures 8.10 and 8.11. Both decompilers functioned correctly, since the insert dead code transformation is designed to confuse a human reader, not the decompiler.

The statement `i += list[i] * list[i + 1]` in Figure 8.7 can be transformed into `i -= list[i] / list[i + 1]`, by changing the increment operator (`+=`) into the decrement operator (`-=`) and the multiplication operator (`*`) into the division operator (`/`). This is the dead code which is added to the pro-

Java Bytecode

#	Instruction	Comment
0	`iconst_0`	push int constant 0
1	`istore_1`	store in local variable 1 (temp)
2	`iconst_0`	push int constant 0
3	`istore_2`	store in local variable 2 (i)
4	`*goto 47`	goto address 47
7	`*iload_2`	load local variable 2 (i)
8	`*iload_2`	load local variable 2 (i)
9	`*iconst_1`	push int constant 1
10	`*iadd`	add integers
11	`*imul`	multiply integers
12	`*iconst_2`	push int constant 2
13	`*irem`	remainder of integer division
14	`*ifne 32`	compare with 0 and if not equal, goto address 32
17	`iload_1`	load local variable 1 (temp)
18	`aload_0`	load local variable 0 (list)
19	`iload_2`	load local variable 2 (i)
20	`iaload`	load array element
21	`aload_0`	load local variable 0 (list)
22	`iload_2`	load local variable 2 (i)
23	`iconst_1`	push int constant 1
24	`iadd`	add integers
25	`iaload`	load array element
26	`imul`	multiply integers
27	`iadd`	add integers
28	`istore_1`	store in local variable 1 (temp)
29	`*goto 44`	goto address 44

...

Figure 8.8. *An example of inserting dead code (part A). Due to the length of the code, it is continued in Figure 8.9.* `list`, `temp` *and* `i` *are local variables 0, 1 and 2 respectively.*

gram, which is stealthy and contains introduced bugs. The opaque predicate `(i * (i + 1) % 2 == 0)`^{T} is used to ensure that the dead code is never executed. We constructed the predicate based on a result from elementary number theory — $\forall x \in \mathbb{N}, 2 \mid (x + x^2)$. So this predicate is always true.

The Halstead, McCabe and Harrison metrics (μ_1, μ_2 and μ_3 in Table 3.1) are all affected by this transformation, since we are adding a new predicate to

Java Bytecode

#	Instruction	Comment
	...	
32	*iload_1	load local variable 1 (temp)
33	*aload_0	load local variable 0 (list)
34	*iload_2	load local variable 2 (i)
35	*iaload	load array element
36	*aload_0	load local variable 0 (list)
37	*iload_2	load local variable 2 (i)
38	*iconst_1	push int constant 1
39	*iadd	add integers
40	*iaload	load array element
41	*idiv	divide integers
42	*isub	subtract integers
43	*istore_1	store in local variable 1 (temp)
44	iinc 2,1	increment local variable 2 (i) by 1
47	iload_2	load local variable 2 (i)
48	aload_0	load local variable 0 (list)
49	arraylength	load length of array
50	iconst_1	push int constant 1
51	isub	subtract integers
52	if_icmplt 7	compare integers and if first is less than second, goto address 7
55	iload_1	load local variable 1 (temp)
56	ireturn	return int from method

Figure 8.9. *An example of inserting dead code (part B). This code continues from Figure 8.8.* `list`, `temp` *and* `i` *are local variables 0, 1 and 2 respectively.*

the program, as well as dead code. μ_1 is increased by 19, which is the number of operators and operands in the new statements `if (i * (i + 1) % 2 == 0)` and `i -= list[i] / list[i + 1]`. Note that the Halstead cost of reading the value of an array element is one array access operator and one array index variable operand. μ_2 increases by one because the predicate `(i * (i + 1) % 2 == 0)` has been added to the program. Since this predicate has been added inside a loop, μ_3, which measures the nesting level of conditionals in a program, is increased. All three metrics have been increased, so it appears that the complexity of the transformed program has increased.

```
public static int combine(int an[])
{
   int i = 0;
   for (int j = 0; j < an.length - 1; j++)
   {
      if (j * (j + 1) % 2 == 0)
         i += an[j] * an[j + 1];
      else
         i -= an[j] / an[j + 1];
   }
   return i;
}
```

Figure 8.10. *Output from* Mocha *for the bytecode in Figures 8.8 and 8.9.*

```
public static int combine(
   int[] int0) {

   int int1;
   int int2;

   int1= 0;
   int2= 0;
   while (int2 < (int0.length - 1))  {
      if (((int2 * (int2 + 1)) % 2)!= 0)  {
         int1= (int1 - (int0[int2] / int0[(int2 + 1)]));
      }
      else {
         int1= (int1 + (int0[int2] * int0[(int2 + 1)]));
      }
      int2 += 1;
   }

   return int1;

}
```

Figure 8.11. *Output from* WingDis *for the bytecode in Figures 8.8 and 8.9.*

8.4 Non-standard Code Patterns

The paper by Proebsting and Watterson [41] discusses decompilation of Java by code pattern matching. By using non-standard patterns of bytecode instructions, it is possible to defeat such decompilers. That is, non-standard code patterns may be used as preventive transformations.

One class of non-standard code patterns arises from performing code motion on very-busy expressions, which are expressions that are used on every possible control flow. Performing code motion on these types of expressions is a standard global optimisation performed by compilers and is known as *hoisting* (see Aho et al. [2], pg. 714).

We did not implement this transformation in our obfuscator due to lack of time. However, we did apply the transformation by hand to small examples of Java code.

Java Source

```
public static int operate(int i, int j) {
   if (i == 0)
      return j + 2;
   else
      return j * 2;
}
```

Java Bytecode

#	Instruction	Comment
0	iload_0	load local variable 0 (i)
1	ifne 8	compare with 0 and if not equal, goto address 8
4	iload_1	load local variable 1 (j)
5	iconst_2	push int constant 2
6	iadd	add integers
7	ireturn	return int from method
8	iload_1	load local variable 1 (j)
9	iconst_2	push int constant 2
10	imul	multiply integers
11	ireturn	return int from method

Figure 8.12. *Code before hoisting very-busy expressions.* `i` *and* `j` *are local variables 0 and 1 respectively.*

The source code in Figure 8.12 is compiled into bytecode. We then perform

code motion on the integer variable `j` and the integer constant `2`, which are very busy expressions. The corresponding bytecode instructions, `iload_1` and `iconst_2`, are moved to before the conditional branch instruction. The result of this code motion can be seen in Figure 8.13.

Java Bytecode

#	Instruction	Comment
0	*iload_1	load local variable 1 (j)
1	*iconst_2	push int constant 2
2	iload_0	load local variable 0 (i)
3	*ifne 8	compare with 0 and if not equal, goto address 8
6	iadd	add integers
7	ireturn	return int from method
8	imul	multiply integers
9	ireturn	return int from method

Figure 8.13. *An example of hoisting very-busy expressions.* `i` *and* `j` *are local variables 0 and 1 respectively.*

The source code recovered by `Mocha` and `WingDis` are in Figures 8.14 and 8.15. The source code output by `Mocha` contains symbols such as `expression`, which are not part of the Java language. The source code output by `WingDis`, while correct syntactically, is incorrect semantically since the variable `StackVar` is not defined by the method.

```
public static int example(int i, int j)
   {
      expression j
      expression 2
      if (i != 0) goto 8 else 6;
      +
      return
      *
      return
   }
```

Figure 8.14. *Output from* `Mocha` *for the bytecode in Figure 8.13.*

The Halstead metric (μ_1 in Table 3.1) is affected by this transformation. It can be seen that since the expressions `j` and the integer `2` are pushed before the

```
public static int example(
   int int0,
   int int1) {


   if (int0!= 0)  {
      return (StackVar * StackVar);
   }
   else {
      return (int1 + 2);
   }

}
```

Figure 8.15. *Output from* `WingDis` *for the bytecode in Figure 8.13.*

`if`-statement is executed, the number of operands in the program is *decreased* by 2. That is μ_1 is decreased by 2. Since the transformation is a targeted preventive transformation, the complexity of the transformed program does not need to increase. The aim is to confuse decompilers but not necessarily confuse a human reader.

Transforming programs using code motion does not affect the control flow of the programs, since we do not insert any new instructions. Thus, the transformation of hoisting very-busy expressions is a preventive rather than a high-level language-breaking transformation. Furthermore, it is a targeted preventive transformation, since it is effective against decompilers that do not search for this particular code pattern.

Code motion also produces patterns of its own in the transformed code. We can break up these code patterns by applying additional transformations. For example, instead of pushing the very-busy expressions onto the operand stack consecutively, we can push the first expression onto the operand stack and execute some other instructions. We can then push the second expression onto the operand stack and then execute the instructions that use the very-busy expressions. These additional transformations hinder decompilation by increasing the number of patterns that are required for complete pattern matching. Alternatively the decompiler has to transform the code into a form on which it can pattern-match.

8.5 Sample Obfuscator Output

The source code in Figure 8.16 was compiled into the bytecode in Figures 8.17 and 8.18.

Java Source

```
public class Bag {
   public static StringBuffer Process(Vector Items) {
      StringBuffer S = new StringBuffer();

      for (int i = 0; i < Items.capacity(); i++) {
         Bag B = (Bag) Items.elementAt(i);
         S.append(i);
         if (B.capacity > 100)
            S.append(" is big");
         else
            S.append(" is small");
      }
      return S;
   }
}
```

Figure 8.16. *Original source code before obfuscation.*

Our obfuscator was invoked on the bytecode in Figures 8.17 and 8.18 with the command:

```
jo Bag.class -maxcost 100 -obflevel 1000
             -path /usr/local/lib/java/classes
```

The bytecode in Figures 8.17 and 8.18 was transformed into the bytecode in Figures 8.19, 8.20 and 8.21.

The source code recovered by `Mocha` is presented in Figure 8.22. The source code recovered by `WingDis` is in Figures 8.23 and 8.24. The transformations are far more effective in confusing humans and decompilers when combined than when they are applied separately to a program.

Java Bytecode

#	Instruction		Comment
0	new	7	create new object of class StringBuffer
3	dup		duplicate top of operand stack
4	invokenonvirtual	12	call StringBuffer constructor
7	astore_1		store in local variable 1 (S)
8	iconst_0		push int constant 0
9	istore_2		store in local variable 2 (i)
10	goto	57	goto address 57
13	aload_0		load local variable 0 (Items)
14	iload_2		load local variable 2 (i)
15	invokevirtual	8	call Items.elementAt
18	checkcast	5	cast the Object reference into a Bag reference
21	astore_3		store in local variable 3 (B)
22	aload_1		load local variable 1 (S)
23	iload_2		load local variable 2 (i)
24	invokevirtual	13	call S.append
27	pop		pop top of operand stack
28	aload_3		load local variable 3 (B)
29	getfield	9	push Bag.capacity field
32	bipush	100	push int constant 100
34	if_icmple	47	compare integers and if first is less than or equal to second, goto address 47
37	aload_1		load local variable 1 (S)
38	ldc_1	1	push reference to the String " is big. "
40	invokevirtual	14	call S.append
43	pop		pop top of operand stack
44	goto	54	goto address 54
47	aload_1		load local variable 1 (S)
48	ldc_1	2	push reference to the String " is small. "
50	invokevirtual	14	call S.append
53	pop		pop top of operand stack
54	iinc	2,1	increment local variable 2 (i) by 1
57	iload_2		load local variable 2 (i)
58	aload_0		load local variable 0 (Items)
59	invokevirtual	10	call Items.capacity
62	if_icmplt	13	compare integers and if first is less than second, goto address 13

...

Figure 8.17. *Bytecode for the source code in Figure 8.16 (part A). Due to the length of the code, it is continued in Figure 8.18.* `Items`, `S`, `i` *and* `B` *are local variables 0 to 3.*

Java Bytecode

#	Instruction	Comment
	...	
65	`aload_1`	load local variable 1 (S)
66	`areturn`	return object reference from method

Figure 8.18. *Bytecode for the source code in Figure 8.16 (part B). This continues the code in Figure 8.17.* `Items`, `S`, `i` *and* `B` *are local variables 0 to 3.*

Java Bytecode

#	Instruction	Comment
0	`*new 7`	create new object of class StringBuffer
3	`*dup`	duplicate top of operand stack
4	`*invokenonvirtual 12`	call StringBuffer constructor
7	`*astore_1`	store in local variable 1 (S)
8	`*iconst_0`	push int constant 0
9	`*istore_2`	store in local variable 2 (i)
10	`*new 48`	create new object of class Node
13	`*dup`	duplicate top of operand stack
14	`*invokenonvirtual 49`	call Node constructor
17	`*astore 4`	store in local variable 4 (p)
19	`*aload 4`	load local variable 4 (p)
21	`*invokevirtual 52`	call p.selectNode2
24	`*astore 5`	store in local variable 5 (r)
26	`*aload 5`	load local variable 5 (r)
28	`*invokevirtual 56`	call r.addNode1
31	`*pop`	pop top of operand stack
32	`*aload 4`	load local variable 4 (p)
34	`*invokevirtual 59`	call p.selectNode1
37	`*astore 6`	store in local variable 6 (q)
39	`*aload 4`	load local variable 4 (p)
41	`*invokevirtual 52`	call p.selectNode2
44	`*astore 6`	store in local variable 6 (q)
	...	

Figure 8.19. *Obfuscated bytecode (part A). Due to the length of the code, the example is continued in Figures 8.20 and 8.21.* `Items`, `S`, `i`, `B`, `p`, `r`, `q`, `a` *and* `b` *are local variables 0 to 8.*

Java Bytecode

#	Instruction		Comment
	...		
46	*aload	4	load local variable 4 (p)
48	*invokevirtual	63	call p.reachableNodes
51	*astore	7	store in local variable 7 (a)
53	*aload	6	load local variable 6 (q)
55	*invokevirtual	63	call q.reachableNodes
58	*astore	8	store in local variable 8 (b)
60	*aload	4	load local variable 4 (p)
62	*aload	7	load local variable 7 (a)
64	*aload	8	load local variable 8 (b)
66	*invokevirtual	69	call a.setDiff
69	*aload	8	load local variable 8 (b)
71	*invokevirtual	73	call p.splitGraph
74	*aload	4	load local variable 4 (p)
76	*aload	6	load local variable 6 (q)
78	*if_acmpeq	94	compare and if equal, goto address 94
81	goto	149	goto address 149
84	*iload_2		load local variable 2 (i)
85	*iload_2		load local variable 2 (i)
86	*imul		multiply integers
87	*iconst_2		push int constant 2
88	*idiv		divide integers
89	*iconst_2		push int constant 2
90	*irem		remainder of integer division
91	*ifeq	108	compare with 0 and if equal, goto address 108
94	*aload_0		load local variable 0 (Items)
95	*iload_2		load local variable 2 (i)
96	*iconst_1		push int constant 1
97	*iadd		add integers
98	*invokevirtual	8	call Items.elementAt
101	*checkcast	5	cast the Object reference into a Bag reference
104	*astore_3		store in local variable 3 (B)
105	*goto	117	goto address 117
	...		

Figure 8.20. *Obfuscated bytecode (part B). This code continues from Figure 8.19 and is continued in Figure 8.21.* `Items`, `S`, `i`, `B`, `p`, `r`, `q`, `a` *and* `b` *are local variables 0 to 8.*

Java Bytecode

#	Instruction		Comment
	...		
108	`aload_0`		load local variable 0 (Items)
109	`iload_2`		load local variable 2 (i)
110	`invokevirtual`	8	call Items.elementAt
113	`checkcast`	5	cast the Object reference into a Bag reference
116	`astore_3`		store in local variable 3 (B)
117	`*jsr`	159	jump to subroutine at address 159
120	`aload_3`		load local variable 3 (B)
121	`getfield`	9	push Bag.capacity field
124	`bipush`	100	push int constant 100
126	`if_icmple`	139	compare integers and if first is less than or equal to second, goto address 139
129	`aload_1`		load local variable 1 (S)
130	`ldc_1`	1	push reference to the String " is big. "
132	`invokevirtual`	14	call S.append
135	`pop`		pop top of operand stack
136	`goto`	146	goto address 146
139	`aload_1`		load local variable 1 (S)
140	`ldc_1`	2	push reference to the String " is small. "
142	`invokevirtual`	14	call S.append
145	`pop`		pop top of operand stack
146	`iinc`	2,1	increment local variable 2 (i) by 1
149	`iload_2`		load local variable 2 (i)
150	`aload_0`		load local variable 0 (Items)
151	`invokevirtual`	10	call Items.capacity
154	`if_icmplt`	84	compare integers and if first is less than second, goto address 84
157	`aload_1`		load local variable 1 (S)
158	`areturn`		return object reference from method
159	`*astore`	9	store return address in local variable 9
161	`*aload_1`		load local variable 1 (S)
162	`*iload_2`		load local variable 2 (i)
163	`*invokevirtual`	13	call S.append
166	`*pop`		pop top of operand stack
167	`*ret`	9	return to address saved in local variable 9

Figure 8.21. *Obfuscated bytecode (part C). This code continues from Figure 8.20.* `Items`, `S`, `i`, `B`, `p`, `r`, `q`, `a` *and* `b` *are local variables 0 to 8. Local variable 9 is used to hold the subroutine return address.*

```
public static StringBuffer Process(Vector vector)
{
   StringBuffer stringBuffer;
   int i;
   Bag bag;
   Node node1;
   Node node3;
   stringBuffer = new StringBuffer();
   i = 0;
   node1 = new Node();
   Node node2 = node1.selectNode2();
   node2.addNode1();
   node3 = node1.selectNode1();
   node3 = node1.selectNode2();
   Set set1 = node1.reachableNodes();
   Set set2 = node3.reachableNodes();
   node1.splitGraph(set1.setDiff(set2), set2);
   if (node1 == node3) goto 94 else 149;
   if (i * i / 2 % 2 == 0) goto 108 else 94;
   bag = (Bag)vector.elementAt(i + 1);
   bag = (Bag)vector.elementAt(i);
   if (bag.capacity > 64)
      stringBuffer.append(" is big.  ");
   else
      stringBuffer.append(" is small.  ");
   i++;
   if (i < vector.capacity()) goto 84 else 157;
   return stringBuffer;
   pop node1
   stringBuffer.append(i);
   endfinalize node1
}
```

Figure 8.22. *Output from* Mocha *for the bytecode in Figures 8.19, 8.20 and 8.21.*

```
   public static StringBuffer Process(
      java.util.Vector Vector0) {

// JavaDis cannot analyze control flow of this method fully
      StringBuffer StringBuffer1;
      int int2;
      Bag Bag3;
      int int4;
      Node Node4;
      Node Node5;
      Node Node6;
      Set Set7;
      Set Set8;

   B0:
      StringBuffer1= new StringBuffer();
      int2= 0;
      Node4= new Node();
      Node5= Node4.selectNode2();
      Node5.addNode1();
      Node6= Node4.selectNode1();
      Node6= Node4.selectNode2();
      Set7= Node4.reachableNodes();
      Set8= Node6.reachableNodes();
      Node4.splitGraph(Set7.setDiff(Set8), Set8);
      if (Node4 == Node6) goto B13;
   B11:
      goto B22;
B12:
      if ((((int2 * int2) / 2) % 2)== 0) goto B15;
B13:
      Bag3= (Bag)Vector0.elementAt((int2 + 1));
   B14:
      goto B16;
B15:
      Bag3= (Bag)Vector0.elementAt(int2);
```

Figure 8.23. *Output from* `WingDis` *for the bytecode in Figures 8.19, 8.20 and 8.21 (part A). Due to length, the output is continued in Figure 8.24.*

```
                ...
B16:
      jsr B24;
      if (Bag3.capacity <= 64) goto B20;
      StringBuffer1.append(" is big.  ");
                ...B19:
      goto B21;
B20:
      StringBuffer1.append(" is small.  ");
B21:
      int2 += 1;
B22:
      if (int2 < Vector0.capacity()) goto B12;
      return StringBuffer1;
   B24:
      int4= stackVar;
      StringBuffer1.append(int2);
      ret int4;

   }
```

Figure 8.24. *Output from* `WingDis` *for the bytecode in Figures 8.19, 8.20 and 8.21 (part B). This code continues from Figure 8.23.*

CHAPTER 9

Conclusion

"This is not the end. It is not even the beginning of the end. It is the end of the beginning."

– Winston Churchill

9.1 The Cost of Obfuscation

Obfuscating an application can affect its execution behaviour. There are three main issues to be considered:

- **Increased code size**
  Since control structures are added to the application, the obfuscated program will be larger than the original.

- **Increased data size**
  Opaque predicates based on alias analysis rely on the obfuscated application building complex data structures at run-time. Thus, the obfuscated program generates more dynamic data than the original.

- **Increased cycle time**
  Introduced code, other than dead code, must be executed by the interpreter. Thus, the obfuscated program will require more instruction cycles to execute than the original.

Increased cycle time is the least serious problem. Most of the introduced instructions are dead code guarded by opaque predicates and are never executed.

The predicates themselves will be simple operations such as pointer or integer comparisons, which affect run-time performance only slightly.

Increased code size can have a severe effect on programs intended to be distributed across a network. The downloading time of a program will be increased and it may execute slower due to deteriorated cache and paging behaviour. The extra memory required to hold the obfuscated code may mean that the instructions in a commonly executed section of code cannot all fit into the cache. The need to keep altering the contents of the cache will degrade performance.

The most serious problem is the increased data size. More dynamic data means that the workload for the garbage collector is increased. There is also the possibility that an obfuscated application will not execute correctly since it exhausts all the available dynamic data storage space.

An obfuscator has to account for these factors when transforming a program. There is a tradeoff between a highly obfuscated program and increased run-time resource requirements. Profiling can determine the most frequently executed parts of a program, and we avoid applying expensive transformations to these parts. Not all of a program contains trade secrets, so different parts of a program can have different levels of obfuscation. The obfuscator's effort can be focussed on the most valuable parts of a program, so we maximise the effect of transformations.

9.2 Future Work

There are many areas of our obfuscator that have to be extended. Given the time constraints of a Masters thesis, we were unable to complete a full implementation of the obfuscator as described in Collberg et al. [9].

9.2.1 Additional features required

Currently, our obfuscator is unable to transform methods which contain exception handlers. It is difficult to ensure that the behaviour of such methods is correct after an obfuscating transformation is applied to these methods. However, a future version of the obfuscator will be able to handle methods that contain exceptions.

We need to implement more control obfuscations in our obfuscator. Other categories of transformations, such as data obfuscations, also need to be investigated. Additional obfuscations will require more data flow analyses to be performed on a program. For example, we need to be able to find very-busy expressions in order to perform the transformation in Section 8.4.

9.2.2 Improvements to existing features

Currently the statements which calculate opaque predicates are inserted adjacent to each other in the program. These statements should be distributed throughout the program, so that identifying such predicates is made more difficult. Hence we need to add a control ordering transformation that reorders statements.

When we insert opaque predicates based on mathematical facts, we do not check whether overflow occurs. We should either determine that overflow does not occur whenever the predicate is evaluated, or we can guard the predicate with additional conditions.

A graphical user interface (GUI) is under development to make our obfuscator easier to use. The GUI will be constructed as a front-end to the existing command line interface. The user communicates with the GUI, which communicates with the command line interface and provides feedback to the user.

We want our obfuscator to avoid using expensive transformations on the most frequently executed parts of a program, which can be identified by a profiler. We need to make the basic block representation available to this profiler so that it can maintain execution counts for the basic blocks. Such a profiler has been built but is not yet completely integrated into the obfuscator.

9.3 Summary

Our Java obfuscator has focused mainly on control flow obfuscations. However, there is much more work to be performed in this area. Of particular interest are opaque constructs, which are covered in depth in Collberg et al. [11].

This thesis has given a broad outline of data obfuscations and preventive transformations. Further work on data obfuscations is in Bertenshaw [6] and Collberg et al. [10].

APPENDIX A

Obfuscation Modules

"I am a brain, Watson. The rest of me is a mere appendix."

– *The Marazin Stone*, Sir Arthur Conan Doyle

An obfuscating transformation module is defined by a Prolog source code file. The file defines the following routines:

- `AnalyseObfuscation` ($\mathcal{T}$, D)
  This routine analyses the transformation $\mathcal{T}$ to determine its features. These features tell the obfuscator to which kinds of source code object the transformation can be applied. D is the additional data needed to analyse the transformation. Currently it is an empty language features set, which the `AnalyseObfuscation` routine adds elements to form the features set of $\mathcal{T}$.

- `AnalyseQualities` ($\mathcal{T}$, S)
  Associated with $\mathcal{T}$ are the potency, resilience and cost measures. This routine returns these measures when the transformation is applied to the source code object S.

- `Obfuscate` ($\mathcal{T}$, S)
  Applies $\mathcal{T}$ to the source code object S and updates the language features ($P_s(S)$) and the appropriateness set $A(S)$.

- Any supporting routines required by the transformation.

The use of a module system makes adding new transformations to the obfuscator easy. All that is required is to add the file name of the new obfuscating transformation module to the list of Prolog source files to be loaded by the Prolog interpreter. After loading the file, the new obfuscation will be available.

A.1 The Insert Bogus Branch Transformation

As an example of an obfuscating transformation, we present the insert bogus branch transformation. This transformation inserts into a method's code a branch that is never taken. The branch may be either forwards or backwards, and may convert the control flow graph of the code into an irreducible control flow graph. That is, this transformation can be either a "Smoke and Mirrors" obfuscation or a high-level language breaking obfuscation.

This transformation uses a P^F predicate to ensure that the inserted bogus branch is never taken. So the obfuscation qualities of the insert bogus branch transformation depend upon those of the predicate that is selected. In fact the qualities of the opaque predicate that is inserted are used as the qualities of the transformation.

We have presented pseudo-code instead of the actual Prolog code, for sake of clarity and brevity.

Interface Routines

`AnalyseObfuscation(insertBogusBranch,` D`)`
  $P_s(\mathcal{T})$ `:=` $D \cup$ {`kind = [bb]`};
  RETURN $P_s(\mathcal{T})$;

The `kind` of source code object (Section 7.1) that the insert bogus branch transformation processes are basic blocks (bb).

`AnalyseQualities(insertBogusBranch,` S`)`
  $\mathcal{T}_{\text{pot}}(S)$ `:= variable;`
  $\mathcal{T}_{\text{res}}(P)$ `:= variable;`
  $\mathcal{T}_{\text{cost}}(P)$ `:= variable;`
  RETURN `(`$\mathcal{T}_{\text{pot}}(S)$`,` $\mathcal{T}_{\text{res}}(P)$`,` $\mathcal{T}_{\text{cost}}(P)$`);`

S is a source code object. The potency, resilience and cost of the insert bogus branch transformation are determined from the opaque predicate that is used.

`Obfuscate(insertBogusBranch,` S`)`
  S' `:= InsertBogusBranch(`S`);`
  RETURN S';

This routine applies the actual obfuscation routine for the insert bogus branch transformation to source code object S.

Support Routines

```
InsertBogusBranch(B)
   P := SelectLibPred(B, {evaltrue = 0.0});
   M := the method to which the basic block B belongs;
   B_target := SelectTargetBB(M, B);
   B' := InsertBogusBranch(M, B, B_target, P);
   RETURN B';
```

B is a basic block. The `SelectLibPred` routine chooses the most appropriate opaque predicate P to insert into the control flow graph. An additional constraint is the fact that P must always evaluate to false. That is, the probability that it evaluates to true is 0.

The `SelectTargetBBNum` routine chooses a basic block B_{target} which is to be the destination of the inserted branch. The `InsertBogusBranch` routine inserts a branch to B_{target}. This branch is inserted before B and is guarded by P, which always evaluates to false. Hence this branch will never be taken — it is a "bogus" branch.

```
SelectTargetBB(M, B)
   C := the control flow graph of M;
   L := the set of basic blocks that belong to C;
   E := the exit basic block of C;
   L' := C \ {B, E};
   B_target := a random basic block from L';
   RETURN B_target;
```

The exit basic block of a method M is not selected as a target of the bogus branch because it does not actually contain any instructions. It is added to the control flow graph of a method to represent the fact that the execution path has left the method (Section 7.4.2).

The basic block B is also not selected because the bogus branch is inserted before B. It is pointless if B is the destination of the branch because it is obvious that the branch is bogus.

```
InsertBogusBranch(M, B, B_target, P)
   C := the control flow graph of the method M;
   I := a new branch instruction, whose destination is the
      basic block B_target;
   P' := AppendInstr(I, P);
   C' := InsertBBBefore(C, P', B);
```

```
Update the control flow graph C':
   Make P' the predecessor of B;
   Make B and B_target the successors of P';

   For each original predecessor A_i, 1 ≤ i ≤ n of B do:
      Replace B with P' as the successor of A_i;
      Add A_i to the set of predecessors of P;
Make C' the control flow graph of M;

RETURN B;
```

M is the method to which the basic block B belongs. The `AppendInstr` routine adds the instruction I to the end of the predicate basic block P. The `InsertBBBefore` routine inserts the predicate basic block P' before B in the control flow graph C.

The rest of this routine updates the control flow graph of the method M.

APPENDIX B

The Opaque Predicate Library

Many control-flow altering transformations implemented by our obfuscator use opaque predicates. We therefore provide a library of opaque predicates in our obfuscator. The scheme for selecting and inserting opaque predicates is presented in this section.

B.1 Selecting an Opaque Predicate

We want to choose the most potent, resilient, cheap and stealthy opaque predicate to insert into a program. We may also specify the kind of the opaque predicate (P^T, P^F or $P^?$) that is required. So the first task of the predicate library is to ensure that only the right kind of predicate is chosen. The appropriateness measure can then be used to determine which predicate has the best combination of the obfuscation measures.

B.2 Opaque Predicate Attributes

The attributes of opaque predicates are listed in Table B.1.

Note that the `features` attribute is determined by the obfuscator at run-time, so it does not need to be hard-coded into the opaque predicate libraries. This means that the opaque predicate libraries require less disk space to store. Also the opaque predicate libraries are more flexible. If the set of language features examined by the obfuscator is extended, there is no need to alter the opaque predicate library files.

The `params` attribute is a list of variables used by the opaque predicate code. These variables must be replaced with actual variables from the program, which

Attribute	Description
`name`	opaque predicate name
`code`	opaque predicate code
`evaltrue`	proportion of times the opaque predicate will evaluate to true
`features`	opaque predicate code language features
`params`	set of variables in the opaque predicate code
`quality`	quality of the opaque predicate

Table B.1. *Opaque predicate attributes.*

transforms the opaque predicate code into a form that can be inserted into the program.

The `quality` attribute is partly calculated at run-time. The potency, resilience and cost of an opaque predicate are stored in the predicate library, while the stealth is calculated from the `features` attribute of the predicate and the language features set of the source code object.

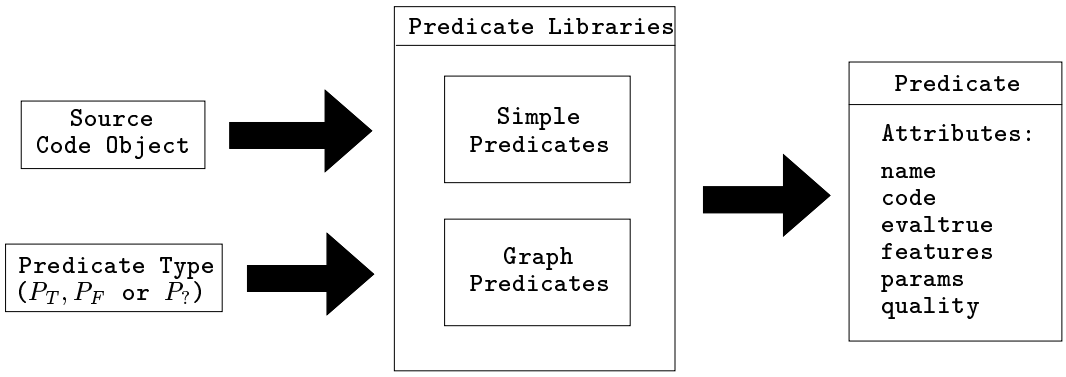


Figure B.1. *The function of the predicate libraries.*

B.3 Inserting an Opaque Predicate

Before we can insert an opaque predicate P into a program, we need to replace the variables in P with actual variables from the program. Suppose that P is to be inserted before a basic block B. B is a node in the control flow graph of a method M, which is defined by the class C. $A_1, \ldots, A_n$ are the predecessors of B in the control flow graph of M. The variables from the program that we can use in P will be the variables that can be used in B. That is, we can use the

variables from the `locals` set of B and the fields of C to replace the variables in the template code of P. If more variables are needed than are provided by the sources above, we can add additional local variables to M, or additional fields to C.

Suppose that we have replaced the variables in the template code for P with variables from the program. To insert P before B, we need to alter the control flow graph of M using the algorithm in Figure B.2.

```
UpdatePredCFG(P, B)
   Make P the only predecessor of B;
   Make B the only successor of P;
   For each original predecessor A_i, 1 ≤ i ≤ n of B, do
      In the set of successors of A_i, replace B by P;
      Add A_i to the set of predecessors of P;
```

Figure B.2. *Algorithm to alter the CFG when a predicate is inserted.*

B.4 The Simple Opaque Predicates Library

Table B.2 lists the opaque predicates based on mathematical facts that are implemented by our obfuscator.

name	code	evaltrue	params
simple1	`if(7y^2 - 1 == x^2)`	0.0	`x::int, y::int`
simple2	`if((x^2 + x) % 2 == 0)`	1.0	`x::int`
simple3	`if((x^3 - x) % 3 != 0)`	0.0	`x::int`
simple4	`if((x % 2 == 0) || ((x^2 - 1) % 8 == 0))`	1.0	`x::int`
simple5	`if(x^2/2 % 2 != 0)`	0.0	`x::int`

Table B.2. *The simple opaque predicates library.*

Simple opaque predicates are assumed to have medium potency, strong resilience and low cost. This information is stored in the `quality` attribute.

The simple opaque predicates library also contains a routine to change the kind of a predicate from P^T to P^F and vice versa. Consider the predicate `if(7y^2 - 1 == x^2)`, whose `evaltrue` attribute is 0.0. We can change the kind of this predicate from P^F to P^T by changing the comparison from `==` to `!=`.

B.5 The Graph Opaque Predicates Library

The graph operations defined by this library use the `Graph` ADT primitives defined in Section 4.2.5. These operations are not opaque predicates in their own right. Instead they either invoke the `Graph` ADT primitives or are *patterns*, which are sequences of primitives. The operations manipulate a graph, while maintaining certain properties, so that the graph can be used to construct an opaque predicate.

Predicates based on graph operations are assumed to have high potency, full resilience and low cost. This information is stored in the `quality` attribute.

The attributes of graph operations are listed in Table B.3.

Attribute	Description
`external`	set of external variables to be substituted
`kind`	kind of entry
`pattype`	type of pattern
`variants`	maximum variant values

Table B.3. *Additional attributes for graph operations.*

The `kind` field has four values, which distinguishes the four categories of graph operations:

1. `initialisation`
   These operations invoke the primitives which create the initial graph.

2. `test`
   These operations test whether certain properties of a graph hold. One property is whether two node pointers reference the same node, which is always false if the pointers refer to nodes in separate connected components.

3. `primitive`
   These operations invoke the primitives which manipulate a graph, such as the `addNode` family which inserts a new node into a graph.

4. `pattern`
   Patterns are sequences of graph primitives. The effects of these patterns on a graph are known, so we can construct opaque predicates by applying a number of patterns to a graph and then testing this graph for a particular property. For example, the `Split` pattern breaks a graph into two separate connected components.

The `external` attribute is a set of variables used by a graph operation in addition to the variables in the `params` attribute. This attribute allows the

`selectNode`$_4$ and `selectNode`$_{4b}$ primitives, which use mutual recursion on an integer parameter `n`, to be implemented (Figure 4.5). The additional variables are to be replaced when the entire opaque predicate is inserted into a method's code. The variables in the `params` attribute are replaced during the construction of the opaque predicate.

Graph operations whose `kind` field has the value `pattern`, also define the `pattype` and `variants` attributes.

The `pattype` attribute determines how the pattern affects the connectivity of a graph. This attribute has three values:

1. `splitting`
   After the pattern is applied, a graph will consist of two separate components.

2. `joining`
   After the pattern is applied, a graph will consist of a single component.

3. `static`
   The pattern does not affect the connectivity of a graph.

Each pattern in the library is parameterised. The parameters determine which variants of the primitives can be used. The `variants` attribute lists the primitive variants that can be used and the variables which need to be replaced. For example, if `selectNode(1)/[a/r, b/p]` appears in the code for a pattern, it will be replaced with the code for the `selectNode(1)` primitive. The `a` and `b` parameters for the primitive will be replaced by the variables `r` and `p` respectively.

The graph operations do not have an `evaltrue` attribute. Instead we keep track of the properties of a graph as the code patterns are applied to it. This information is supplied by the `pattype` attribute. Suppose that we want to construct an opaque predicate that is always false. We create a new graph and apply code patterns to it. We ensure that the graph is divided into two separate components, by applying the `splitGraph` pattern to the graph and then only applying code patterns that do not join the two components. If `p` points to a node in one component and `q` points to a node in the other component, then `p == q` will always be false.

The graph opaque predicate library is in Tables B.4 and B.5. `N` stands for `object('Node')` and `S` stands for `object('Set')`.

name	kind	code	params	external
newNode	initialisation	`new Node()`		
test	test	`if(p == q)`	`p::N, q::N`	
addNode(1)	primitive	`p.addNode1()`	`p::N`	
addNode(2)	primitive	`p.addNode2()`	`p::N`	
selectNode(1)	primitive	`p.selectNode1()`	`p::N`	
selectNode(2)	primitive	`p.selectNode2()`	`p::N`	
selectNode(3)	primitive	`p.selectNode3(n)`	`p::N`	`n::int`
selectNode(4)	primitive	`p.selectNode3b(n)`	`p::N`	`n::int`
reachable	primitive	`p.reachable()`	`p::N`	
splitGraph	primitive	`p.splitGraph(a,b)`	`p::N, a::S, b::S`	
setDiff	primitive	`a.setDiff(b)`	`a::S, b::S`	

Table B.4. *The* `primitive` *routines of the graph opaque predicates library.*

name variants	code	params	pattype

insert(I,J)
I ∈ [1 . . . 4],
J ∈ [1 . . . 2]
	`if (P == null)`
	`  return new Node();`
	`else {`
	`  r = selectNode(I)/[p/P];`
	`  return addNode(J)/[p/r];`
	`}`
		`P::N,`
		`r::N`
			static

move(I) I ∈ [1 . . . 4]	`return selectNode(I)/[p/P];`	`P::N`	static

link(I,J)
I ∈ [1 . . . 4],
J ∈ [1 . . . 2]
	`q = selectNode(I)/[p/P];`
	`r = selectNode(J)/[p/P];`
	`if (r.car == r)`
	`  r.car = q;`
		`P::N,`
		`q::N,`
		`r::N`
			joining

split(I)
I ∈ [1 . . . 4]
	`Q = selectNode(I)/[p/P];`
	`a = reachable/[p/P];`
	`b = reachable/[p/q];`
	`c = setDiff;`
	`splitGraph/[p/P, a/c, b/b];`
	`return Q;`
		`P::N,`
		`Q::N,`
		`a::S,`
		`b::S,`
		`c::S`
			splitting

Table B.5. *The* `patterns` *of the graph predicates library. For all of these patterns, the* `kind` *attribute is* `pattern` *and the* `external` *attribute is an empty set.*

APPENDIX C

The Set Abstract Data Type

In this chapter we give the Java source code for the `Set` ADT. The implementation is not a complete ADT for sets — only those operations required by the `Graph` ADT defined in Section 4.2.5 are defined. Note that the `Set` ADT uses the `Hashtable` class from the Java library routines.

```
public class Set {
   protected Hashtable theSet;

   public Set() { this.theSet = new Hashtable(); }

   /* Accessors */
   public void setSet(Hashtable newSet) { this.theSet = newSet; }
   public Enumeration elements() { return this.theSet.elements(); }

   /* Returns true if the set contains the element,
      otherwise returns false */
   public boolean hasMember(Object elt) {
      return this.theSet.containsKey(elt);
   }

   /* Adds an element to the set */
   public Set insert(Object elt) {
      this.theSet.put(elt, elt);
      return this;
      }
```

```
/* Removes the items in set B from the current set */
public Set setDiff(Set B) {
   Set newSet = new Set();
   Enumeration enum = this.elements();
   while (enum.hasMoreElements()) {
      Object elt = enum.nextElement();
      if (!B.hasMember(elt)) newSet.insert(elt);
      }
   return newSet;
   }
}
```

Bibliography

[1] Neil Aggarwal. A Java bytecode obfuscator, May 1997. `http://www.monmouth.com/~neil/Obfuscate.html`.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986. ISBN 0-201-10088-6.

[3] David Aucsmith. Tamper-resistant software: An implementation. In *Lecture Notes in Computer Science No. 1174*, pages 317–333, May/June 1996.

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26 (4):345–420, December 1994. `http://www.acm.org/pubs/toc/Abstracts/0360-0300/197406.html`.

[5] Joseph A. Bank. Java security. `http://swissnet.ai.mit.edu/~jbank/javapaper/javapaper.html`, December 1995.

[6] Alan Bertenshaw. String obfuscation in java. Undergraduate project in Computer Science, The University of Auckland, 1997.

[7] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[8] Cristina Cifuentes and K. John Gough. Decompilation of binary programs. *Software – Practice and Experience*, 25(7):811–829, July 1995.

[9] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, Department of Computer Science, The University of Auckland, New Zealand, July 1997. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/Collb% ergThomborsonLow97a/index.html`.

[10] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *IEEE International Conference on Computer Languages 1998, ICCL'98.*, Chicago, IL, May 1998. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/Collb% ergThomborsonLow97d/index.html`.

[11] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *ACM SIGPLAN-SIGACT*
Symposium on Principles of Programming Languages 1998, POPL'98, San Diego, CA, January 1998. `http://www.cs.auckland.ac.nz/~collberg/Research/Publications/Collb% ergThomborsonLow98a/index.html`.

[12] WingSoft Company. WingDis 2.03– the Java Decompiler, March 1997. `http://www.wingsoft.com/wingdis.shtml`.

[13] Netscape Communications Corporation. Netscape communicator, March 1998. `http://home.netscape.com/comprod/products/communicator/index.html`.

[14] Symantec Corporation. Café - Visual Java Development and Debugging Tools, March 1997. `http://cafe.symantec.com/index.html`.

[15] Jeffrey Adgate Dean. *Whole–Program Optimization of Object-Oriented Languages.* PhD thesis, University of Washington, 1996.

[16] R. A. DeMillo, W. M. McCracken, R. J. Martin, and J. F. Passafiume. *Software Testing and Evaluation.* The Benjamin/Cummings Publishing Company, Inc., 1987.

[17] James R. Gosler. Software protection: Myth or reality? In *CRYPTO'85 — Advances in Cryptology*, pages 140–157, August 1985.

[18] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996. ISBN 0-201-63451-1.

[19] M. H. Halstead. *Elements of Software Science.* Elsevier North-Holland Inc., 1977. ISBN 0-44-00205-7.

[20] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Notices*, 16(3):63–74, 1981.

[21] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, September 1981.

[22] Amir Herzberg and Shlomit S. Pinter. Public protection of software. *ACM Transactions on Computer Systems*, 5(4):371–393, November 1987.

[23] Susan Horwitz. Precise flow-insensitive May-Alias analysis is NP-hard. *TOPLAS*, 19(1):1–6, January 1997.

[24] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In IEEE, editor, *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture, December 2–4, 1996, Paris, France*, pages 90–97, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, December 1996. IEEE Computer Society Press.

[25] Cheng-Hsueh A. Hsieh, Marie T. Conte, Teresa L. Johnson, John C. Gyllenhaal, and Wen mei W. Hwu. Optimizing NET compilers for improved Java performance. *IEEE Computer*, 30(6):67–75, June 1997.

[26] Borland International. JBuilder, September 1997. `http://www.borland.com/jbuilder/index.html`.

[27] Microsoft International. Visual J++, July 1997. `http://www.microsoft.com/products/prodref/221_ov.htm`.

[28] Rex Jaeschke. Encrypting C source for distribution. *Journal of C Language Translation*, 2(1), 1990. `http://jclt.iecc.com`.

[29] Eron Jokipii. Jobe – the Java obfuscator, 1996. `http://www.primenet.com/~ej/index.html`.

[30] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, May 1996.

[31] Mark D. LaDue. HoseMocha, January 1997. `http://www.oasis.leo.org/java/development/bytecode/obfuscators/Hos% eMocha.dsc.html`.

[32] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997. `http://java.sun.com:80/docs/books/vmspec/html/VMSpecTOC.doc.html`.

[33] Douglas Low. Protecting Java code via code obfuscation. *ACM Crossroads*, Spring issue 1998. `http://www.acm.org/crossroads/xrds4-3/codeob.html`.

[34] Apple's QuickTime lawsuit. `http://www.macworld.com/pages/june.95/News.848.html` and `may.95/News.705.html`, May – June 1995.

[35] John J. Marciniak, editor. *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc, 1994. ISBN 0-471-54004-8.

[36] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, December 1976.

[37] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, 2nd edition, 1993. ISBN 0-201-62427-3.

[38] John C. Munson and Taghi M. Kohshgoftaar. Measurement of data structure complexity. *Journal of Systems Software*, 20:217–225, 1993.

[39] E. I. Oviedo. Control flow, data flow, and program complexity. In *Proceedings of IEEE COMPSAC*, pages 146–152, November 1980.

[40] Todd A. Proebsting, Greg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications, a way ahead of time (wat) compiler. In *Third USENIX Conference on Object-Oriented Technologies*, June 1997. `http://www.cs.arizona.edu/sumatra/papers/coots97.ps`.

[41] Todd A. Proebsting and Scott A. Watterson. Krakatoa: Decompilation in Java (Does bytecode reveal source?). In *Third USENIX Conference on Object-Oriented Technologies*, June 1997.

[42] G. Ramalingam. The undecidability of aliasing. *TOPLAS*, 16(5):1467–1471, September 1997.

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

[44] Pamela Samuelson. Reverse-engineering someone else's software: Is it legal? *IEEE Software*, pages 90–96, January 1990.

[45] Uwe Schöning. Complexity cores and hard problem instances. In *Proceedings of SIGAL'90*, pages 232–240, August 1990.

[46] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 32–41, January 1996.

[47] Paul A. Suhler, Nager Bagherzadeh, Miroslaw Marlek, and Neil Iscoe. Software authorization systems. *IEEE Software*, 3(5):34–41, September 1986.

[48] Inc. Sun Microsystems. JavaBeans: The Only Component Architecture for Java, 1998. `http://java.sun.com/beans/index.html`.

[49] Robert Tolksdorf. Programming languages for the Java virtual machine, September 1997. `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`.

[50] Hans Peter van Vliet. Crema — The Java obfuscator, 1996. `http://www.ph-erfurt.de/information/java/dev/misc/Crema/index.html%` .

[51] Hans Peter van Vliet. Mocha — The Java decompiler, 1996. `http://wkweb4.cableinet.co.uk/jinja/mocha.html`.

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

[53] Michael Wolfe. *High Performance Compilers For Parallel Computing*, chapter 7. Addison-Wesley, 1996. ISBN 0-8053-2730-4.