McGill University
School of Computer Science
Sable Research Group

Obfuscating Java: the most pain for the least gain

Sable Technical Report No. 2006-5

Michael Batchelder Laurie Hendren

October 16, 2006

Contents

1 Introduction **3**

2 Related Work **4**

3 JBCO Structure **5**

4 Operator-level Obfuscation **5**

4.1 Renaming Identifiers: classes, fields and methods (RI[C,M,F]) 5

4.2 Embedding Constant Values as Fields (ECVF) 6

4.3 Packing Local Variables into Bitfields (PLVB) 6

4.4 Converting Arithmetic Expressions to Bit-Shifting Operations (CAE2BO) 6

4.5 Impact of Operator-level Obfuscations on Decompilers 6

5 Obfuscating Program Structure **7**

5.1 Adding Dead-Code Switch Statements (ADSS) 7

5.2 Finding and Reusing Duplicate Sequences (RDS) 7

5.3 Replacing `if` Instructions with Try-Catch Blocks (RIITCB) 8

5.4 Building API Buffer Methods (BAPIBM) 8

5.5 Building Library Buffer Classes (BLBC) 8

5.6 The impact of program structure obfuscations on decompilers 9

6 Exploiting the Design Gap **9**

6.1 Converting Branches to `jsr` Instructions (CB2JI) 9

6.2 Reordering `load` Instructions Above `if` Instructions (RLAII) 9

6.3 Disobeying Constructor Conventions (DCC) 10

6.4 Partially Trapping Switch Statements (PTSS) 10

6.5 Combining Try Blocks with their Catch Blocks (CTBCB) 11

6.6 Indirecting `if` Instructions (III) 11

6.7 `Goto` Instruction Augmentation (GIA) 11

6.8 The impact of exploiting the semantic gap on decompilers 11

7 Empirical Evaluation **12**

7.1 Impact of Obfuscations on Performance 13

7.2 Impact of obfuscations on control-flow complexity 13

8 Conclusions and Future Work **15**

List of Figures

1 Performance and Complexity Ratios comparing obfuscated programs to their original forms. 14

List of Tables

I Measuring Decompiler Success against Operator-level Obfuscations 7
II Measuring Decompiler Success against Structure Obfuscations 9
III Measuring Decompiler Success against Semantic Gap Obfuscations 12

Abstract

Software obfuscators are used to transform code so that it becomes more difficult to understand and harder to reverse engineer. Obfuscation of Java programs is particularly important since Java's binary form, Java bytecode, is relatively high-level and susceptible to high-quality decompilation. The objective of our work is to develop and study obfuscation techniques that produce obfuscated bytecode that is very hard to reverse engineer while at the same time not significantly degrading performance.

We present three kinds of obfuscation techniques that: (1) obscure intent at the operational level; (2) obfuscate program structure such as complicating control flow and confusing object-oriented design; and (3) exploit the semantic gap between what is allowed at the Java source level and what is allowed at the bytecode level.

We have implemented all of our techniques as a tool called JBCO (Java Byte Code Obfuscator), which is built on top of the Soot framework. We developed a benchmark suite for evaluating our obfuscations and we examine runtime performance, control flow graph complexity and the negative impact on decompilers. These results show that most of the obfuscations do not degrade performance significantly and many increase complexity, making reverse engineering more difficult. The impact on decompilers was two-fold. For those obfuscations that can be decompiled, readability is greatly reduced. Otherwise, the decompilers fail to produce legal source code or crash completely.

1 Introduction

Reverse engineering is the act of uncovering the underlying design of a product through analysis of its structure, features, functions and operation. Analysis is often performed by taking apart the said product to discover the various pieces or modules that make up its design. Reverse engineering has a long history, including applications in military and pharmacology industries. It could be argued that software has proven to be among the most susceptible forms to reverse-engineering. Since software is an easily and cheaply reproduced product (unlike a military bomber, for example) it must rely on either passive protection such as a patent-law or some form of active protection such as hiding software applications on servers, encryption or obfuscation. This paper presents and studies a wide range of techniques for obfuscating Java bytecode.

Obfuscation is the obscuring of intent in design. With software this means transforming code such that it remains semantically equivalent to the original, but is more esoteric and confusing. A simple example is the renaming of variable and method identifiers. By changing a method from `getName` to a random sequence of characters such as `sdfhjioew`, information about the method is hidden that a reverse-engineer could otherwise have found useful. A more complex example is introducing unnecessary control flow that is hidden using opaque predicates, expressions that will always evaluate to the same answer (true or false) but whose value is not possible to estimate statically. Obfuscation is one of the more promising forms of code protection because it is translucent. It may be obvious to a malicious attacker that a program has been obfuscated but this fact will not necessarily improve their chances at reverse-engineering. Also, obfuscation can severely complicate a program such that even if it is decompilable it is very difficult to understand, making extraction of tangible intellectual property close to impossible, without serious time investment.

Java is particularly vulnerable to reverse engineering because its binary form, bytecode, is relatively high-level and contains considerable information about types, and field and method names. Furthermore, there are often many references in the code to known fields and methods in publicly-available class libraries, including the standard ones provided with a Java implementation. Java decompilers exploit these weaknesses and there has been a long history of decompilers that convert bytecode into quite readable Java source code, particularly when the bytecode is in exactly the format produced by known `javac` compilers [13, 15–17, 19, 22].

Java obfuscators are one way of foiling decompilers. These tools convert Java bytecode into semantically-equivalent bytecode that is difficult to decompile, or even if decompilable is very hard for a programmer or tool to understand. However, a very important factor is that one wants the obfuscations to make reverse-engineering difficult (the most pain), but at the same time not hurt performance of the obfuscated application (the least gain). This tradeoff is not obvious, since the same obfuscations that make it hard for a decompiler may also severely impact the analysts and optimizations in JIT compilers found in modern Java Virtual Machines (JVMs).

The main goal of our work was to develop and implement a collection of obfuscations that would make reverse engineering difficult, while at the same time not affect performance too much. We examine some variations of pre-

viously suggested obfuscations and we also develop some new techniques, most notably techniques that exploit the semantic gap between what can be expressed in Java bytecode and what is allowed in valid Java source.

The remainder of the the paper is organized as follows. In Section 2 we give a short summary of previous work in obfuscation. Section 3 gives a high-level overview of our JBCO obfuscator. Sections 4 through 6 present our obfuscations grouped by type: operator-level obfuscation, modifying program structure and exploiting the semantic gap. At the end of each of those sections we summarize the impact of the obfuscations on three decompilers. Due to space considerations we can only give a brief description of each obfuscation and can't show the full results of the obfuscations. However, some detailed examples and some challenge cases for decompilers can be found at `http://www.sable.mcgill.ca/JBCO`. In Section 7 we introduce a benchmark set and provide a summary of the impact of each obfuscation on runtime performance and control-flow complexity. Finally, Section 8 gives conclusions and future work.

2 Related Work

Obfuscation is a form of *security through obscurity*. While Barak argues that there are seemingly few truly irreversible obfuscations [2] and, in theory, "deobfuscation" under certain general assumptions has been shown by Appel to be NP-Easy [1], obfuscation is nevertheless a valid and viable solution for general programs.

Early attempts at obfuscation invariably involved machine-level instruction rewriting. Cohen used a technique he called "program evolution" to protect operating systems that included the replacement of instructions, or small sequences of instructions, with ones that perform semantically equal functions, instruction reordering, adding or removing arbitrary jumps, and even de-inlining methods [5]. Many of these ideas are now standard.

Much later, a more theoretical approach to obfuscations was presented by Collberg *et al.* [6]. They outline obfuscations as program transformations and develop terminology to describe an obfuscation in terms of performance effect and quality. They rely on a number of well-known software metrics [4, 12, 18] to measure quality. Later, in [7], they reconsider the concepts of lexical obfuscations (name changing) and data transformations (*e.g.*, splitting boolean values into two discrete numerics that are combined only at evaluation time). However, their chief contributions are in control-flow obfuscations. They make use of opaque predicates to introduce dead code, specifically engineering the dead branches to have buggy versions of the live branches.

A technique for combining the data of a program with its control-flow was developed by Wang *et al.* [24], whereby control and data flow analysis become co-dependent. While not Java-specific, a two-process obfuscation approach which uses inter-process communication to communicate between a "control-flow" process and a "computation" process was presented by Ge. *et al.* [8]. Unfortunately, this kind of low-level jury-rigging is not possible in Java. A different multi-process technique for maintaining opaque predicates presented by Majumdar and Thomborson could certainly be implemented in Java [14].

Sakabe *et al.* concentrate their efforts on the object-oriented nature of Java — the high-level information in a program. Using polymorphism, they invent a unique return type class which encapsulates all return types and then modify every method to return an object of this type [20]. Method parameters are encapsulated in a similar way and method names are cloned across different classes. In this way the true return types of methods and the number and types of a methods parameters are hidden. They further obfuscate typing by introducing an `if` with an opaque predicate to branch around new object instantiations which confuses the true type of the object and they use exceptions as explicit control ow. Unfortunately, their empirical results show significantly slower execution speeds — an average slowdown of 30% — and a 300% blowup in class file size.

Sonsonkin *et al.* present more high-level obfuscations which attempt to confuse program structure [21]. They suggest the coalescing of multiple class files into one — combining the functionality of two or more functionally-separate sections of the program — and its reverse of splitting a single class file into multiple unique units.

The obfuscations presented in this paper build upon both the older and simple operation-level obfuscations as well as control flow and program structure obfuscations. The variations that we have implemented have been chosen to maximize obfuscation, while also minimizing the impact on runtime performance. We have also developed a new set of obfuscations, different from the others, which exploit the semantic gap between Java bytecode and Java source. Many of these were inspired by our experiences in building Java bytecode optimizers and and decompilers. The cases that are difficult for those tools are exactly the cases that should be created by obfuscators.

3 JBCO Structure

JBCO – our Java ByteCode Obfuscator – is built on top of Soot [23]. Soot is a Java bytecode transformation and annotation framework providing multiple intermediate representations and infrastructure for dataflow analysis and transformations. In developing JBCO we use two intermediate representations: Jimple, a typed 3-address intermediate form; and Baf, a typed abstraction of bytecode.

JBCO is a collection of Jimple and Baf transformations and analyses. Whenever possible, we analyze and transform Jimple, since it is at a higher abstraction and easier to work with. However, some low-level obfuscations require modifying actual bytecode instructions and for those we work at the Baf level. There are three categories of analyses and transformations:

Information Aggregators: collect data about the program for other transformations such as identifier names, constant usage, or local variable to type pairings.

Code Analyses: build new forms of information about the code such as control-flow graphs, stack height and type data, or use-def chains. These are used to identify where in the program transformations can be applied. For example, in order to produce verifiable bytecode we must ensure proper matchings between allocations of objects and their initializations.

Instrumenters: actually modify the code, adding obfuscations or shuffling the code to obscure meaning.

JBCO can be used either as a command-line tool or via a graphical user interface.[1] Each obfuscation can be activated independently and depending on the severity of the of the obfuscation desired, a weight of 0-9 can be given where 0 corresponds to no applications of the obfuscations and 9 corresponds to applying it everywhere possible. We also provide a mechanism to allow developers to guide the obfuscations to specific regions of a program by using regular expressions to specify certain classes, fields or methods. This is useful when a developer wants certain parts to be heavily obfuscated or when a specific hot method should not be obfuscated because of performance considerations.

4 Operator-level Obfuscation

Our first group of obfuscations works at the operator level. That is, we convert a local operation into a semantically equivalent computation that is harder for a reverse-engineer to understand. These obfuscations are likely to be decompilable, but the decompiled code is hard to understand. [2]

4.1 Renaming Identifiers: classes, fields and methods (RI[C,M,F])

Perhaps one of the simplest, but also very effective, obfuscations is identifier renaming. Java bytecode retains the names of classes, fields and methods and these names are often very useful for the reverse engineer. For example, a method called `getDate`, with a return type of `Date` is quite explicit. Some of these names cannot be touched because they may be: defined in libraries, referred to via reflection, or as entry points into the application. However, for the remaining cases, the fields, methods and classes can be renamed, as long as this remaming is done consistently through the application.

We have developed two techniques for choosing identifier names. The first is to create random strings using characters that are hard to distinguish visually. Thus we randomly create valid identifiers from the alphabets of {S, 5 $}, {l,1, I} and {_}. In this case decompiled code would include identifiers like I1l, Ill, _____, ______, S5S$ or SS5$. These identifiers will clearly appear mangled to reverse-engineers and although a tool could replace them with less visually confusing identifiers, it is not a straightforward process to create semantically meaningful names for them. Our second technique is to steal names from other parts of the application or standard library. In this case the `getDate` may be renamed `getFirstName`. This switch is not as obvious to a reverse-engineer or tool and conveys incorrect semantic information.

[1] JBCO will soon be released as a new component of Soot.

[2] For each obfuscation, we give the acronym we use for it. This acronym us used both in the experimental results and also as the flag used to enable the obfuscation in JBCO.

4.2 Embedding Constant Values as Fields (ECVF)

Programmers often use constants, particularly string constants, to convey important information. For example, a statement of the form `System.err.println("Illegal argument, value must be positive.");` provides some context to the reverse engineer. The point of the ECVF obfuscation is to move the constant into a static field and then change references to the constant into references to the field. This could lead to something like `System.err.println(___.lI);`, which conveys significantly less meaning. If the static field is initialized only once, an interprocedural constant propagation could sometimes undo this obfuscation. However, if the initialization of the field is further obfuscated through the use of an opaque predicate, this is no longer possible.

4.3 Packing Local Variables into Bitfields (PLVB)

In order to introduce a level of obfuscation on local variables with primitive types (boolean, char, byte integer), it is possible to combine some variables and pack them into one variable which has more bits (a long, for example). The simplest solution would be to pack variables starting from the least-significant or most-significant bit. However, to provide further confusion we randomly choose a range of bits to use for each local variable. For example, an integer variable may get packed into bits 9 through 43. Since each read or write of the original variable must be replaced by a packing and unpacking operation in the obfuscated code, this can potentially slow down the application. Thus, this obfuscation is used sparingly and applied randomly to only some of the locals. Without further obfuscation of the constants used for packing and unpacking, this kind of obfuscation could be undone by a clever decompiler that was aware of this technique.

4.4 Converting Arithmetic Expressions to Bit-Shifting Operations (CAE2BO)

Optimizing compilers sometimes convert a complex operation such as multiplication or division into a sequence of cheaper operations. This same trick can be used to obfuscate the code. In particular, we look for instances of expressions in the form of $v * C$ (a similar technique is used for v/C), where v is a variable and C is a constant. We then extract from C the largest integer value i which is less than C and is also a power of 2, $i = 2^s$, where $s = floor(log_2(v))$. We then compute the remainder, $r = v - i$. If s is in the range of $-128 \ldots 127$, then we can convert the original computation as $(v << s) + (v * r)$ and the expression $v * r$ can be further decomposed. In order to further obfuscate the computation we don't use the shift value s directly, but rather find an equivalent value s'. To do this we take advantage of the fact that shifting a 32-bit word by 32 (or a multiple of 32) always returns it to its original state. Thus we choose a random multiple m, and compute a new shift value, $s' = (byte)(s + (m * 32))$, which computes an equivalent shift value in the correct range $(-128 \ldots 127)$.

As an example, an expression of the form $v * 195$ in the original program would be converted first to $(v << 7) + (v << 6) + (v << 1) + v$ and then the three shift values would be further obfuscated to something like $(v << 39) + (v << 38) + (v << -95) + v$.

A decompiler that is aware of this calculation could potentially reverse it, but if one or more of the constants were hidden with an opaque predicate, this would further hamper decompilation.

4.5 Impact of Operator-level Obfuscations on Decompilers

Although we fully expected all of these simple, operator-level, obfuscations to be decompilable (i.e. a decompiler should produce correct compilable Java code, even though this code would be confusing to a reverse engineer), we were quite surprised to find the results in Table I. For these and subsequent decompiler tests in this paper, we created some small micro-tests for each obfuscation.[3] A score of *Pass* indicates that the decompiler produced correct Java source that could be recompiled by `javac`, whereas *Fail* indicates that the decompiler produced code that cannot be recompiled, and *Crash* means that the decompiler did not terminate normally.

[3]The reason that we used micro-tests is that some decompilers, most notably pattern-based decompilers like Jad, are very sensitive to whether the bytecode looks exactly like it came from a `javac` compiler or not. Since all of our tests have been run through Soot, which even without obfuscations is sometimes enough to confuse decompilers, we wanted to ensure that our tests were small enough so that we could measure the impact of the obfuscation itself and not indirect effects due to processing with Soot.

Table I: Measuring Decompiler Success against Operator-level Obfuscations

Obfuscation	Jad	SourceAgain	Dava
Renaming Identifers: classes, fields and methods	Fail	Pass	Pass
Embedding Constant Values as Fields	Fail	Fail	Fail
Packing Local Variables into Bitfields	Fail	Fail	Fail
Converting Arithmetic Expressions to Bit-Shifting Ops	Fail	Fail	Pass

Why do decompilers fail on these simple obfuscations? Jad is unable to correctly process our renamed identifiers containing the character $. The other three obfuscations unwittingly exploit a semantic gap between bytecode and Java source. At the bytecode level, booleans, bytes and chars are all expressed as integers, whereas in Java these are given different types which must be used consistently and in a manner so as not to lose precision. The decompilers failed to find consistent typing and casting for these computations and thus produced Java source that would not compile.[4]

5 Obfuscating Program Structure

Program structure can be thought of as framework. In a building this would be the supporting beams, the floors, and the ceiling. It would not be the walls or the carpeting. We define structure in this chapter to include two facets: low-level method control flow and high-level object-oriented design. These obfuscations should be decompilable by modern decompilers such as SourceAgain and Dava.

5.1 Adding Dead-Code Switch Statements (ADSS)

The switch construct in Java bytecode offers a useful control flow obfuscation tool. It is the only organic way (other than the try-catch structure) to manufacture a control flow graph that has a node whose successor count is greater than two. This can severely increase the complexity of a method.

This obfuscation adds edges to the control flow graph by inserting a switch. To ensure that the switch itself is never executed it is wrapped in an opaque predicate. All bytecode instructions with a stack height of zero are potentially safe jump targets for cases in the switch. We have defined an analysis to compute these zero-height locations and we select a random set of them as targets for the cases in the dead switch. This obfuscation increases the connectedness and overall complexity of a method. A decompiler cannot remove the dead switch because it cannot statically determine the value of the opaque predicate.

5.2 Finding and Reusing Duplicate Sequences (RDS)

Because of the nature of bytecode, there is often a fair amount of duplication. Even within one method a sequence of instructions might appear a number of times. By finding these clones and replacing them with a single switched instance we can potentially reduce the size of the method while also confusing the control flow, creating control flow that is not naturally expressed in Java.

We determine when a duplicate sequence D is a clone of the original sequence O using the following checks and analyses:

- D must be of the same length as O and for each index i, instruction D_i must equal O_i.
- Each D_i must be protected by the same try blocks as the original O_i. If the original is not protected at all, neither can the duplicate.
- Every instruction in a sequence other than the first must have no predecessors that fall *outside* the sequence (*i.e.* there should be no branching into the middle of a sequence).

[4]Clearly our research group would like to fix Soot/Dava to properly handle this variation of the typing problem - it is quite interesting to have one subgroup building a decompiler, while at the same time another subgroup is trying to break it!

- Each D_i must share the same stack height and basic types as the original O_i.
- Each D_i must not have the same offset within the method as *any* instruction O_j.

When a duplicate sequence is found, a new integer which acts as a control flow flag is created. The duplicates are removed completely and replaced with an assignment of the flag to a unique id followed by a goto directed at the first instruction in the original sequence. The original sequence is prepended with instructions which store 0 to the flag (the "first" unique id) and appended with a switch. The default jump falls through to the next instruction (the successor of the original sequence). A jump to the successor of each duplicate sequence is added to the switch based on its flag id. For each method we search for duplicates of length 3 through 20.

5.3 Replacing `if` Instructions with Try-Catch Blocks (RIITCB)

The try-catch construct in the Java language can be used to create control flow, either through an explicit throw statement or by inserting a statement that is known to create an exception. This obfuscation exploits a well known fact of Java: invoking an instance method on a `null` object will always result in a NullPointerException. This obfuscation searches for `ifnull` tests and replaces these with an unneeded and harmless instance method call on the reference which is wrapped in a try block. The method call will raise an exception when the reference is null, thus the target of the original `ifnull` branch is used as the target for the handler part of the try. The only thing remaining to do is to prepend the handler with a pop instruction in order to remove the NullPointerException reference that will be placed on the stack. This is very similar to the approach by Sakabe *et al.* in [20], although they relied on creating special exception objects which added unnecessary overhead.

5.4 Building API Buffer Methods (BAPIBM)

A lot of information is inherent in Java programs because of the widespread use of the Java libraries. These libraries have clear and well-defined documentation. The very existence of library objects and method calls can give shape and meaning to a method based entirely on how they are being used. The method calls that direct execution into the native Java libraries — the design of which is known as an Application Programming Interface (API) — cannot be renamed because the obfuscator should not change library code[5]. Therefore, the next best option is to hide the names of the library methods. The approach we take in this obfuscation is to indirect all library calls through intermediate methods with nonsensical identifiers.

Each method of each class is checked for library calls. A new method M is then created for every library method L referenced in the program. The method M is instrumented to invoke the library method L. The new method M is placed in a randomly chosen class in order to cause "class-coagulation", an increase in class interdependence. Therefore, this obfuscation is two-fold. It confuses the object-oriented design of the program and also hides the library method calls by moving them to a completely different "physical" part of the program.

5.5 Building Library Buffer Classes (BLBC)

In addition to library method calls, having a class that extends a library class directly can lend a certain amount of clarity to a program. Parent class methods that are over-ridden in the child are more obvious as well. Any experienced Java programmer will quickly grasp a large amount of design intent from this information.

This obfuscation attempts to cloud this particular design structure of Java. For each class which directly extends a library class we create a new buffer class. The buffer class is inserted as a child of the library class and a parent of the application class. Since no part of the program itself ever uses the buffer class directly, methods over-ridden in the child class can be defined as nonsense methods in the buffer class, further adding confusion. This serves as a way to complicate and confuse the design of the program by adding extra layers and, ultimately, it spreads the single-intent class structure over multiple files making it difficult for a reverse-engineer.

[5]While it is not completely impossible, it is not reasonable. Obfuscating library code would mean that those modified libraries would have to be distributed with the program as well, causing both licensing issues and an unreasonable increase in the program's distribution size

5.6 The impact of program structure obfuscations on decompilers

As in the previous section we developed micro-tests for each obfuscation to be as fair as possible to the decompilers. The results are shown in Table II. Jad fairs badly when trying to decompile our structure obfuscations, most likely due to its lack of control flow analysis. It resorts to leaving pure bytecode in its output where it is unable to produce correct source. More surprisingly, SourceAgain also have difficulty with the heavier control flow obfuscations. In fact RDS causes it to crash completely.

Table II: Measuring Decompiler Success against Structure Obfuscations

Obfuscation	Jad	SourceAgain	Dava
Adding Dead-Code Switch Statements	Fail	Fail	Pass
Finding and Reusing Duplicate Sequences	Fail	Crash	Pass
Replacing if Instructions with Try-Catch Blocks	Fail	Pass	Pass
Building API Buffer Methods	Fail	Fail	Fail
Building Library Buffer Classes	Fail	Pass	Pass

None of the decompilers were able to properly mark which methods might throw exceptions, which is a requirement of Java source. Because some methods indirected by the BAPIBM obfuscation might throw exceptions the new methods that call them are required to as well.

6 Exploiting the Design Gap

Inherent in the design of the Java language are certain gaps between what is representable in Java source code and what is representable in bytecode. The classic example is the goto bytecode instruction which has no direct counterpart in source[6].

The obfuscations detailed in this section were designed to exploit these bytecode-to-source gaps. Smart decompilers can sometimes transform the obfuscated bytecode into a semantically equivalent form of source code yet it is usually unreadable. Often, however, these obfuscations can result in a situation where decompilers either produce incorrect code or do not produce any code whatsoever. On occassion the decompilers crash altogether.

6.1 Converting Branches to `jsr` Instructions (CB2JI)

The `jsr` bytecode[7], short for Java subroutine, is analogous to the `goto` other than the fact that it pushes a return address on the stack. Normally, the return address is stored to a register after a `jsr` jump and when the subroutine is complete the `ret` bytecode is used to return.

The `jsr` - `ret` construct is particularly difficult to handle when dealing with typing issues because each subroutine can be called from multiple places, requiring that type information be merged and therefore a more conservative estimate. Also, decompilers will usually expect to find a specific `ret` for every `jsr`.

This obfuscation replaces `if` and `goto` targets with `jsr` instructions. The old jump targets have `pop` instructions inserted before them in order to throw away the return address which is pushed onto the stack. If the jump target's predecessor in the instruction sequence falls through then a `goto` is inserted after it which jumps directly to the old target (stepping over the `pop`).

6.2 Reordering `load` Instructions Above `if` Instructions (RLAIl)

In some cases patterns in bytecode produced by `javac` can be examined to identify areas of possible obfuscation. This simple obfuscation looks for situations where a local variable is used directly following both paths of an `if`

[6]Abrupt jumps in source must be performed through the `break` or `continue` statements which force a certain level of structure since they must always be directly associated with well-defined statement blocks

[7]The `jsr` was originally introduced to handle finally blocks — sections of code that are ensured to run after a try block whether an exception is thrown or not. It is a historical anomaly that is no longer used by modern `javac` compilers.

branch. That is, along both branches the first instruction loads the variable on to the stack. This is a somewhat common occurance — consider code that follows the pattern `if x then i=...else i=...`.

This obfuscation then moves the `load` instruction above the `if`, removing its clones along both branches. While a modern decompiler like Dava which is based on a 3-address intermediate representation will be able to overcome this obfuscation with little problem, any decompiler relying on pattern matching (such as Jad) will become very confused.

6.3 Disobeying Constructor Conventions (DCC)

The Java language specification [9] stipulates that class constructors – those methods used to instantiate a new object of that class type – must always call either an alternate constructor of the same class or their parent class' constructor as the *first directive*. In the event that neither is specified in source `javac` explicitly adds a call to the parent at the beginning of the method in the compiled bytecode.

While this super call, as a rule, must be the first statement in the Java *source* it is, in fact, not required to be the first within the bytecode. By exploiting this fact it is possible to create constructor methods whose bytecode representation cannot be converted into legal source. This obfuscation randomly chooses among four different approaches to transforming constructors in order to confuse decompilers:

Wrapping the super call within a try block: This ensures that any decompiled source will be *required* to wrap the call in a try as well to conform to the rules of Java. To properly allow the exception to propagate, the handler unit — a `throw` instruction — is appended to the end of the method.

Taking advantage of classes which are children of java.lang.Throwable: This approach inserts a `throw` instruction before the super call and creates a new try block in the method that traps just the new `throw`. The handler unit is designated to be the super call itself. This takes advantage of the fact that the class is throwable and can be pushed onto the stack through the throw mechanism instead of the standard load.

Inserting a `jsr` jump and a `pop` instruction directly before the super constructor call: The `jsr`'s target is the `pop` instruction, which removes the subsequent return address that is pushed on the stack as a result of the `jsr` instruction. This confuses the majority of decompilers which have problems dealing with `jsr` instructions.

Adding new instructions before the super call: This approach inserts a `dup` followed by an `ifnull` before the super call. The `ifnull` target is the super call. The `if` branch instruction will always be `false` since the object it is comparing is the object being instantiated in the current constructor. Two new instructions are inserted along the false branch of the `if`: a `push null` followed by a `throw`. A new try block is created spanning from the `ifnull` up to the super call. The catch block is appended to the end of the method as a sequence of `pop, load o, goto sc`, where `o` is the object being instantiated and `sc` is the super call. This confuses decompilers because it is more difficult to deduce which local will be on the stack when the super call site is reached.

6.4 Partially Trapping Switch Statements (PTSS)

There is a big gap between high-level structured use of try-catch blocks in Java source and their low-level byte implementation. Whereas the Java construct allows only well-nested and structured uses, the bytecode implementation is a much lower trap abstraction. A bytecode trap specifies a bytecode range $a \ldots b$, a handler unit h, and an exception type E. If an exception T is raised within the method at bytecode c then the JVM searches for a trap in the list which matches either the type of T or a parent type of T whose bytecode range $a \ldots b$ contains c. If a trap is found then the stack is emptied, T is pushed on top, and the program counter is set to the handler h.

There are no rules that enforce nesting of these ranges and at the bytecode level these may overlap or even share code with handler code.

Thus, one way of confusing decompilers is to trap sequential sections of bytecode that are not necessarily sequential in Java source code. The perfect example of this is the switch construct. In source code, the switch statement encapsulates different blocks of code as *targets* of the switch. However, in bytecode there is nothing explicitly tying the `switch` instruction to the different code blocks (*i.e.* there is no explicit encapsulation).

If the `switch` is placed within a trap range along with only *part* of the code blocks which are associated as its targets then there will be no way for an automatic decompiler to output semantically equivalent code that looks anything like the original source. It simply must reproduce the trap in the source code in some form, potentially by duplicating code.

This transformation is conservatively limited to those switch constructs which are *not* already trapped, which alleviates some analysis work. This implies that the `switch` instruction itself and any additional instructions that are selected for trapping were not previously trapped in any way.

6.5 Combining Try Blocks with their Catch Blocks (CTBCB)

Java source code can only represent try-catch blocks in one way: with a try block directly followed by one or more catch blocks associated with it. In bytecode, however, try blocks can protect the same code that is used to handle the exceptions it throws or one of its catch blocks can appear "above" it in the instruction sequence.

This obfuscation combines a try-catch block such that both the beginning of the try block and the beginning of the catch block are the same instruction. This is accomplished by prepending the first unit of the try block with an `if` that branches to either the try code or the catch code based on an integer control flow flag. Once the try section has been officially entered, the flag is set to indicate that any execution of the `if` in the future should direct control to the catch section. The integer flag is reset to its original value when the try section is completed.

6.6 Indirecting `if` Instructions (III)

While `javac` will always produce very predictable try blocks it is possible to abuse these constructs in other ways. This obfuscation takes advantage of this by indirecting `if` branching through `goto` instructions which are within a special try block. Normally, modern compilers would remove the `goto` and modify the `if` to jump directly to its final target. However, since a try block protects all these gotos it is not valid to remove them unless the code can be statically shown to never raise an exception. Since there is no explicit `goto` allowed in Java source, it becomes very difficult for a decompiler to synthesize equivalent source code.

6.7 `Goto` Instruction Augmentation (GIA)

Explicit `goto` statements are not allowed in Java source. Studies have shown this to be a good design decision [3]. In source, you must use abrupt statements instead. Java bytecode *does* have a `goto` instruction because it is necessary for simulating higher-level constructs such as loops. Therefore it is possible to insert an explicit `goto` within the bytecode. While it is very easily reversed using control flow graph analysis it can still cause many simple decompilers to fail.

Our obfuscation takes a simple approach. It randomly splits a method into two sequential parts: The first, containing the start of the method, P_1 and a second, containing the end of the method, P_2. It then reorders these two parts and inserts two `goto` instructions. The first `goto` is inserted as the first instruction in the method and points to the start of P_1. The second is inserted at the end of P_1 and targets P_2. The final method now looks like { `goto` P_1, P_2, P_1, `goto` P_2}.

As an added step, a try block is manufactured to span from the end of P_2 to the beginning of P_1, thereby "gluing" the two sections together. This makes it difficult for a decompiler to shuffle the instructions back to their original order.

6.8 The impact of exploiting the semantic gap on decompilers

All of the decompilers have difficulty with the obfuscations from this section. Table III shows that Dava was only successful with one out of seven. Jad and SourceAgain failed all tests. In all cases Jad generates source with many bytecode instructions left in it and therefore it is difficult to identify anything specific as the cause. In most of the cases SourceAgain was unable to realize where certain local variables were used. It would declare a local variable within a nested block even when the parent block used that variable, for example. Both SourceAgain and Dava had difficulties

marking methods which might throw exceptions. They could not properly name the super constructor method calls in DCC either, leaving the bytecode name `<init>` which is not a legal Java identifier.

Table III: Measuring Decompiler Success against Semantic Gap Obfuscations

Obfuscation	Jad	SourceAgain	Dava
Converting Branches to jsr Instructions	Fail	Fail	Crash
Reordering loads Above if Instructions	Fail	Fail	Pass
Disobeying Constructor Conventions	Fail	Fail	Crash
Partially Trapping Switch Statements	Fail	Fail	Fail
Combining Try Blocks with their Catch Blocks	Fail	Fail	Fail
Indirecting if Instructions	Fail	Fail	Fail
Goto Instruction Augmentation	Fail	Fail	Fail

Dava also crashed on the DCC obfuscation due to its inability to handle the throwing of explicitly null exceptions. Soot is unable to read in classfiles that include `jsr` instructions with no matching `ret`. This is not a limitation of Dava itself but we marked it as having crashed on the CB2JI obfuscation because of this.

7 Empirical Evaluation

Since an important aspect of our work is the evaluation of the impact of obfuscations on runtime performance, we have gathered a set of benchmarks from a graduate-level compiler optimizations course where students were required to develop interesting and computation-intensive programs for comparing the performance of various Java Virtual Machines. Each one was written in Java and compiled with *javac*. The benchmarks represent a wide array of programs each with their own unique coding style, resource usage, and ultimate task. Below is a list of each benchmark with a brief description of its key features.

Asac: is a multi-threaded sorter which compares the performance of the Bubble Sort, Selection Sort, and Quick Sort algorithms. It uses reflection to access each sorting algorithm class by name and creates a new thread for each one. In the experiments, the benchmark sorts a randomly generated array of 30,000 integers.

Chromo: implements a genetic algorithm, an optimization technique that uses randomization instead of a deterministic search strategy. It generates a random population of chromosomes. With mutations and crossovers it tries to achieve the best chromosome over successive generations. It instantiates many chromosome objects and, for each generation, evaluates over 5,000 of these 64-bit array chromosomes.

Decode: implements an algorithm for decoding encrypted messages using Shamir's Secret Sharing scheme.

FFT: performs fast fourier transformations on complex double precision data.

Fractal: generates a tree-like (as in leaves) fractal image. It calls java.lang.Math trigonometric methods heavily and is deeply recursive in nature.

LU: implements Lower/Upper Triangular Decomposition for matrix factorization.

Matrix: performs the inversion function on matrices.

Probe: uses the Poisson distribution to compute a theoretical approximation to pi for a given alpha.

Triphase: performs three separate numerically-intensive programs. The first is linpack linear system solver that performs heavy double precision floating-point arithmetic. The second is a heavily multithreaded matrix multiplication algorithm. The third is a multithreaded variant of the Sieve prime-finder algorithm. In total, 1,730 java threads are created during the execution of this program with as a many as 130 of them alive at once.

7.1 Impact of Obfuscations on Performance

Figure 1(a) summarizes the ratio of the execution time of obfuscated benchmark to the execution time of original benchmark.[8] A ratio of 1 indicates that the obfuscation had no effect on performance, a ratio of less than 1 indicates that the obfuscated benchmark was faster, and a ratio greater than 1 indicates that the obfuscated benchmark was slower.[9] Each bar corresponds to one obfuscation, the diamond on the bar corresponds to the average ratio over all the benchmarks. The bars show the range of ratios with the bottom of the bar corresponds to the benchmark with the lowest ratio and the top of the bar corresponds to the benchmarks with the highest ratio.

All experiments were run on an AMD Athlon^{TM}64 X2 Dual Core Processor 3800+ machine with 4 gigabytes of RAM running Ubuntu 6.06 Dapper Drake Linux. The machine was unloaded and running no extraneous processes at the time each experiment was performed. Sun Microsystem's Java HotSpot^{TM}64-Bit Server VM (build 1.5.0 06 b05) was used in all experiments with the initial and maximum Java heap sizes set to 128 and 1024 megabytes, respectively.

As shown by recent empirical studies by Gu *et al.* [10, 11], small variations in code layout can lead to relatively large performance differences in Java (on the order of 5-10%). Thus, we can expect some performance differences between the original and obfuscated code just because the obfuscated code leads to different code layouts. Thus, the significant performance differences are really those less than .95 or greater than 1.05.

We can see that average performance of the obfuscated code is very reasonable, with quite a few below 1. The most expensive obfuscation appears to CB2JI, which converts branches to jsr instructions, with an average slowdown of 1.16 and a maximum slowdown of almost 1.6. This maximum slowdown was in the LU benchmark and in further investigations we found that almost the entire cause for the decreased performance was a slowdown in one deeply nested loop which now had very complex control flow. The JIT compiler struggled to analyze this nested loop and caused a 5-fold slowdown in compilation time. There are 6 obfuscations that lead to a maximum slowdown greater than 1.2. These obfuscations should be used carefully, avoiding hot methods whenever possible.

In some cases the obfuscations actually seem to slightly improve peformance. The RLAII obfuscation that moves loads above ifs is one such case. This does make sense since it is moving a load that is known to be needed on both branches earlier in the computation.

7.2 Impact of obfuscations on control-flow complexity

Whereas Figure 1(a) shows how much the obfuscated application slowed down (the gain), Figure 1(b) shows the the increase in code complexity due to obfuscations (the pain). We have opted for a reasonably simple measure of complexity based on counting the number of nodes and edges in the control flow graph of the program, where the nodes in the control flow graph are basic blocks and the edges are control flow edges. Obfuscations which change the structure of the code may introduce new control flow edges and/or redirect control flow edges to split basic blocks. The numbers reported in Figure 1(b) show the ratio of the sum of the number of nodes and edges of the obfuscated code over the sum of nodes and edges of the original. This count captures the impact of control flow obfuscations well, but does not measure the impact of simpler obfuscations such as identifier renaming.

As expected, the operation-level obfuscations have no impact on control-flow complexity. The increase in complexity for these obfuscations is better demonstrated by an increase in identifier complexity and an increase in the number of operations.[10]

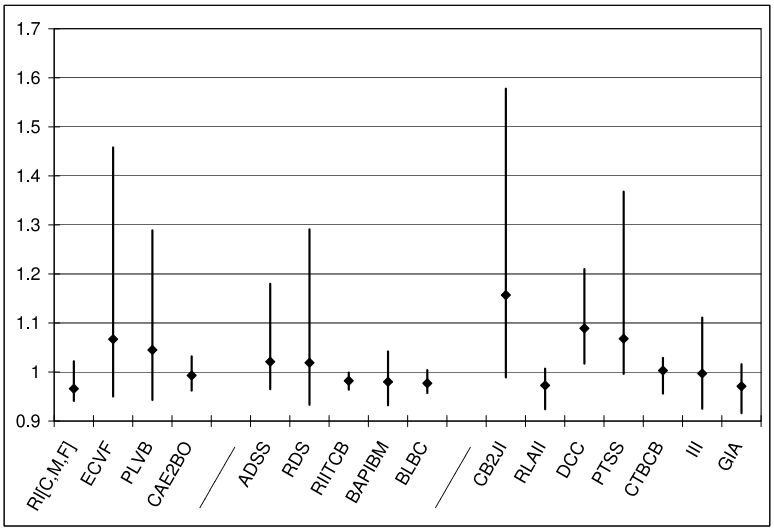
The structure obfuscations do show a signficant increase in control-flow complexity. The two obfuscations that confuse the object-oriented design, Building API Buffer Methods (BAPIBM) and Building Library Buffer Classes (BLBC), do not increase control-flow complexity, but would affect other metrics which measure coupling.

As we have shown in Table III, the third group of obfuscations are those that are most effective in breaking decompilers. Some of these also show some significant increases in control-flow complexity. Based on our experiences with Dava, which can partially handle many of these cases, we expect that a complete decompilation will lead to source code with a lot of code duplication and heavy use of labeled blocks.

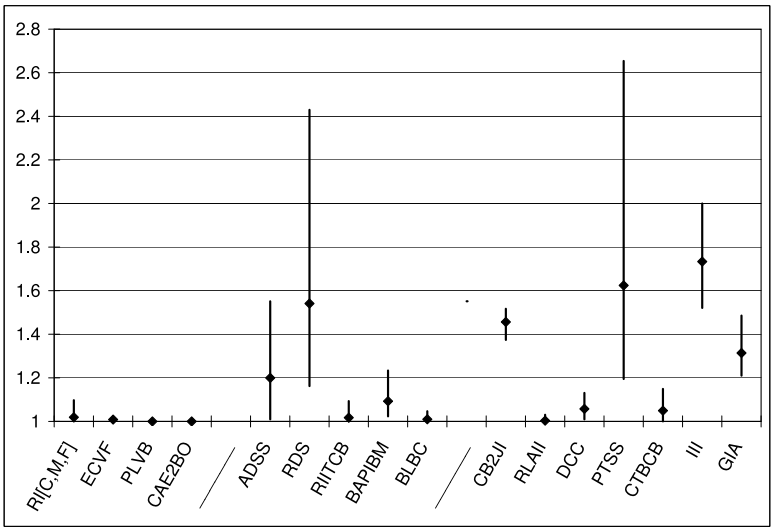
[8] To time the original benchmark, we first processed it via Soot with no obfuscations turned on. This is to factor out any differences due to Soot processing.

[9] The execution time is computed by timing 10 runs, dropping the slowest and fastest and averaging the remaining 8 runs. The largest standard error we observed over these 8 runs was 2.6% and the majority of the measurements had a standard error well below that.

[10] We have collected these kinds of metrics, which do demonstrate an increase.



(a) Performance Ratio — (average execution time of obfuscated program)/(average execution time of original program). High and low bars are given.



(b) Complexity Ratio — (sum of edges and nodes in obfuscated CFG)/ (sum of edges and nodes in original CFG). High and low bars are given.

Figure 1: Performance and Complexity Ratios comparing obfuscated programs to their original forms.

8 Conclusions and Future Work

This paper has presented a collection of obfuscation techniques for Java bytecode. The intent was to make the bytecode harder to reverse-engineer without imposing significant performance penalties. We presented three groups of obfuscations. The first group were relatively simple and focused on operator-level obfuscations which are intended to make the code harder to reverse-engineer. Although we didn't expect these to break decompilers, several decompilers failed to properly type the obfuscated code. The second group of techniques were aimed at confusing both the control flow within methods and the object-oriented design. The decompilers also had trouble with some of these techniques, although they should in principle be decompilable, even though the decompiled code is much more complex than the original. These decompiler failures were mostly due to the obfuscations creating unstructured control flow which is much more difficult to decompile than structured control flow. The third group of obfuscations were mostly new techniques and were aimed at exploiting the gap between the high-level structure of Java source and the lower-level rules for bytecode. These obfuscations were very successful in both increasing the complexity of the code and breaking the decompilers.

The effect on performance of the obfuscations varied, the average performance ratio of obfuscated/original ranged from .96 to 1.16, which is very reasonable. The maximum ratio reached almost 1.6, with 6 of 16 obfuscations having a maximum ratio of over 1.2. This demonstrates that in some cases the obfuscations should not be used everywhere in the program, particularly in hot methods. More detailed analysis of specific instances showed that some performance slowdowns were due to the increased time needed by the JIT compilers to deal with the analysis of the complex control flow created by the obfuscations. Hence the obfuscations are not just more difficult for reverse engineers to understand, they also cause problems for tools like compilers and decompilers.

All of our obfuscations have been implemented in the JBCO tool, which is built on top of Soot. We are quite pleased with the wide variety of obfuscations we developed and this paper has shown how they work individually. We feel that the next interesting problem is to develop new techniques to automatically determine where each obfuscation should be applied and how to best select a combination of obfuscations so that one can achieve the best overall effect without applying all obfuscations at all points. We have also started to develop a wide variety of metrics to quantify the effect of obfuscators and decompilers.

Acknowledgments

This work was supported, in part, by NSERC and FQRNT

References

[1] A. W. Appel. Deobfuscation is in NP, Aug. 21 2002.

[2] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139:1–??, 2001.

[3] B. A. Benander, N. Gorla, and A. C. Benander. An empirical study of the use of the goto statement. *J. Syst. Softw.*, 11(3):217–223, 1990.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[5] F. B. Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12(6):565–584, 1993.

[6] C. Collberg, C. Thomborson, and D. Low. Breaking abstractions and unstructuring data structures. In *ICCL '98: Proceedings of the 1998 International Conference on Computer Languages*, page 28, Washington, DC, USA, 1998. IEEE Computer Society.

[7] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, volume 28, pages 735–746, Aug. 2002.

[8] J. Ge, S. Chaudhuri, and A. Tyagi. Control flow based obfuscation. In *DRM '05: Proceedings of the 5th ACM workshop on Digital rights management*, pages 83–92, New York, NY, USA, 2005. ACM Press.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[10] D. Gu, C. Verbrugge, and E. Gagnon. Code layout as a source of noise in JVM performance. In *Component And Middleware Performance workshop, OOPSLA 2004*, 2004.

[11] D. Gu, C. Verbrugge, and E. M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 111–121. ACM Press, 2006.

[12] S. Henry and K. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[13] Jad - the fast JAva Decompiler. Available on: `http://www.kpdus.com/jad.html`.

[14] A. Majumdar and C. Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In V. Estivill-Castro and G. Dobbie, editors, *Twenty-Ninth Australasian Computer Science Conference (ACSC 2006)*, volume 48 of *CRPIT*, pages 187–196, Hobart, Australia, 2006. ACS.

[15] J. Miecnikowski and L. J. Hendren. Decompiling Java bytecode: problems, traps and pitfalls. In R. N. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 111–127. Springer Verlag, 2002.

[16] J. Miecznikowski and L. Hendren. Decompiling Java using staged encapsulation. In *Proceedings of the Working Conference on Reverse Engineering*, pages 368–374, October 2001.

[17] Mocha, the Java Decompiler. Available on: `http://www.brouhaha.com/~eric/computers/mocha.html`.

[18] J. C. Munson and T. M. Khoshgoftaar. Measurement of data structure complexity. *J. Syst. Softw.*, 20(3):217–225, 1993.

[19] N. A. Naeem and L. Hendren. Programmer-friendly decompiled Java. In *Proceedings of the 14th IEEE International Conference on Program Comprhension*, 2006.

[20] Y. Sakabe, M. Soshi, and A. Miyaji. Java obfuscation with a theoretical basis for building secure mobile agents. In *Communications and Multimedia Security*, pages 89–103, 2003.

[21] M. Sosonkin, G. Naumovich, and N. Memon. Obfuscation of design intent in object-oriented applications. In *DRM '03: Proceedings of the 3rd ACM workshop on Digital rights management*, pages 142–153, New York, NY, USA, 2003. ACM Press.

[22] Source Again - A Java Decompiler. Available on: `http://www.ahpah.com/`.

[23] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[24] C. Wang, J. Hill, J. Knight, and J. Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, Dec. 2000.