Obfuscating Java: the most pain for the least gain*

Michael Batchelder and Laurie Hendren
mbatch@cs.mcgill.ca hendren@cs.mcgill.ca

School of Computer Science, McGill University, Montreal, QC, Canada

Abstract. Bytecode, Java's binary form, is relatively high-level and therefore susceptible to decompilation attacks. An obfuscator transforms code such that it becomes more complex and therefore harder to reverse engineer. We develop bytecode obfuscations that are complex to reverse engineer but also do not significantly degrade performance. We present three kinds of techniques that: (1) obscure intent at the operational level; (2) complicate control flow and object-oriented design (i.e. program structure); and (3) exploit the semantic gap between what is legal in source code and what is legal in bytecode. Obfuscations are applied to a benchmark suite to examine their affect on runtime performance, control flow graph complexity and decompilation. These results show that most of the obfuscations have only minor negative performance impacts and many increase complexity. In almost all cases, tested decompilers fail to produce legal source code or crash completely. Those obfuscations that are decompilable greatly reduce the readability of output source.

1 Introduction

Reverse engineering is the act of uncovering the underlying design of a product through analysis of its structure, features, functions and operation. It has a long history, including applications in military and pharmacology industries, but it could be argued that software has proven to be among the most susceptible to its attacks. Since software is an easily and cheaply reproduced product it must rely on either passive protection such as a patent or some form of active protection such as hiding software on servers, encryption, or obfuscation.

Obfuscation is the obscuring of intent in design. It is one way of foiling decompilers. 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, while it may be obvious to a malicious attacker that a program has been obfuscated, this fact will not necessarily improve their chances at decompilation. Also, obfuscation can severely complicate a program such that even if it is

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

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. There are also 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 are quite a few products that convert bytecode into Java source code that very similar to the original and is quite readable, particularly when the bytecode is in exactly the format produced by known `javac` compilers [12–15, 17, 20].

This paper presents and studies a wide range of techniques for obfuscating Java bytecode. 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 analysis and optimizations in JIT compilers found in modern Java Virtual Machines (JVMs).

This tradeoff is the main goal of our work. We developed and implemented a collection of obfuscations that hinder reverse engineering attempts, while at the same time do not affect performance too much. We examine some variations of previously suggested obfuscations and we also develop some new techniques, most notably those which 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. Section 3 gives a high-level overview of our software obfuscator, the Java Bytecode Obfuscator (JBCO). Sections 4 through 6 present our obfuscations grouped by type: operator-level obfuscation, program structure modification, and semantic gap exploitation. Each section ends with a summary of the impact of the obfuscations on three decompilers. Due to space limitations we briefly describe each obfuscation. However, detailed code examples and 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 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. Transformations included instruction reordering, adding or removing arbitrary jumps, and even de-inlining methods [5].

Much later, a theoretical approach was presented by Collberg *et al*. [6]. They outline obfuscations as program transformations and develop terminology to describe them in terms of performance effect and quality. They rely on a number of well-known software metrics [4, 11, 16] to measure quality. Later, in [7], they reconsider 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 modifications. They make use of opaque predicates to introduce dead code, specifically engineering the dead branches to have buggy versions of the live branches.

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 [18]. 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 method's parameters are hidden. They further obfuscate typing by introducing opaque predicates that branch around new object instantiations which confuses the true type of the object and they use exceptions as explicit control flow. 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 [19]. They suggest the coalescing of multiple class files into one — combining the logic of two or more functionally-separate sections of the program — and its reverse, splitting a single class file into multiples.

The obfuscations presented in this paper build upon both the simple operation-level obfuscations as well as control flow and program structure obfuscations. We have also developed a new set of obfuscations, 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 [21]. Soot is a Java bytecode transformation and annotation framework providing multiple intermediate representations and infrastructure for dataflow analysis and transformations. JBCO uses 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 are implemented in Baf since they require modifying actual bytecode instructions. There are three categories of analyses and transformations:

Information Aggregators: collect data about the program for the transformationsi, such as constant usage and local variable-to-type pairings.

Code Analyses: collect information about the code such as control flow graphs, type data, and use-def chains, which help identify where in the program transformations can be applied (e.g. in order to produce verifiable bytecode we must ensure proper matchings between allocations of objects and their initializations).

Instrumenters: are the actual obfuscations, ransforming the code to obscure meaning.

JBCO can be used 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 obfuscation desired, a weight of 0-9 can be given where 0 turns it off completely and 9 corresponds to applying it everywhere possible. We also provide a mechanism to limit the obfuscations to specific regions of a program by using regular expressions to specify certain classes, fields or methods. This is useful if a user 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 should be decompilable, but the decompiled code is expected to be harder to understand.[2][3]

4.1 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(ObjectA.field1);`, which conveys significantly less meaning. An interprocedural constant propagation could potentially 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.

4.2 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

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

[2] Our identifier renamer obfuscation was left out of the paper due to space limits. We developed a unique approach to garbleing names, but the overall technique is quite common.

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

into one variable which has more bits. To provide maximum 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 of a 64-bit long. Each read or write of the original variable must be replaced by packing and unpacking operations in the obfuscated code and this might slow down the application. Thus, it is used sparingly and applied randomly to only some locals. Without further obfuscation of the bitshifting and bitmasking constants used for packing and unpacking, however, a clever decompiler could overcome this technique.

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

Optimizing compilers sometime convert a complex operation such as multiplication or division into a sequence of cheaper ones. 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 extract from C the largest integer i where $i < C$ 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 $-128 \ldots 127$, we can convert the original to $(v << s) + (v * r)$ and the expression $v * r$ can be further decomposed. In order to further obfuscate 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 the original value. We choose a random multiple m, and compute a new but equivalent shift value, $s' = (byte)(s + (m * 32))$.

As an example, an expression of the form $v * 195$ 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, it could stymie the attempt.

4.4 Impact of Operator-level Obfuscations on Decompilers

Although we fully expected all of these simple, operator-level, obfuscations to be decompilable (i.e. correct and compilable source code would be produced, even if less readable than the original), we were surprised to find the results in Table 1. For these and subsequent decompiler tests in this paper, we created some small micro-tests for each obfuscation.[4] A score of *Pass* indicates that the decompiler produced correct Java source that could be recompiled by `javac`, *Fail* indicates that the produced code would not recompile, and *Crash* is the result of a decompiler not terminating normally.

Why do decompilers fail on these simple obfuscations? The three obfuscations unwittingly exploit a semantic gap between bytecode and Java source. Booleans, bytes and

[4] We used micro-tests because some decompilers, most notably pattern-based 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 to measure the impact of the obfuscation itself and not indirect effects due to processing with Soot.

Table 1. Measuring Decompiler Success against Operator-level Obfuscations

Obfuscation	Jad	SourceAgain	Dava
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

chars are expressed as integers in bytecode, whereas in source these are given unique types which must be used consistently and in a manner so as not to lose precision. The decompilers failed to properly type and cast for these computations and produced output that was not recompilable.[5]

5 Obfuscating Program Structure

Program structure can be thought of as the 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 to include two facets: low-level method control flow and high-level object-oriented design. Modern decompilers such as SourceAgain and Dava should be able to handle these techniques, in principle.

5.1 Adding Dead-Code Switch Statements (ADSS)

The switch construct in 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 dead 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 implemented an analysis to find these zero-height locations and we randomly select some as targets for the cases switch. This 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 a single method. 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 patterns not naturally expressed in Java.

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

[5] 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!

- 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 (or no) try blocks as the original O_i.
- Every instruction in a sequence other than the first must have no predecessors that fall *outside* the sequence (*i.e.* no branching into the middle of a sequence).
- Each D_i must share the same stack height and types as the original O_i.
- Each D_i must not have the same offset within the method as *any* instruction O_j.

The algorithm searches for duplicates of length 3 to 20. When a duplicate sequence is found, a new integer is created to act as a control flag. Each duplicate is removed 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 switch 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.

5.3 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 cannot be renamed because the obfuscator should not change library code[6]. Therefore, the next best option is to hide library method calls. We do this by indirecting library calls through intermediate methods that have nonsensical identifiers.

Each program method is checked for library calls. A new method M is then created for each library method L referenced in the program. M is modified to invoke L. 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 hides the library method calls by indirecting them through a different "physical" part of the program.

5.4 Building Library Buffer Classes (BLBC)

Having a class that extends a library class directly can also lend a certain amount of clarity to a program. Parent class methods that are over-ridden in the child are more obvious as well. Experienced Java programmers are able to quickly grasp design intent from this information.

This obfuscation attempts to cloud this particular design structure of Java. For each class C, which directly extends a library class L, we create a new buffer class B. It is

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

inserted as a child of L and a parent of C. Since no part of the program itself ever uses B directly, methods over-ridden in C can be defined as nonsense methods in B, further adding confusion. This complicates and confuses the design of the program by adding extra layers. Ultimately, it spreads the single-intent class structure over multiple files making it difficult for a reverse engineer to understand.

5.5 The impact of program structure obfuscations on decompilers

The results are shown in Table 2. Jad fairs badly when decompiling 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 has difficulty with the heavier control flow obfuscations. RDS causes it to crash completely.

Table 2. 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
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 BAPIBM might throw exceptions the new methods that call them are required to as well.

6 Exploiting the Design Gap

Certain gaps between what is representable in Java source code and what is representable in bytecode exist. The classic example is the goto instructioni that has no direct counterpart in source[7].

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, the result is incorrect decompiled code or no decompiler output whatsoever. Sometimes a decompiler crashes altogether.

6.1 Converting Branches to `jsr` Instructions (CB2JI)

The `jsr` bytecode[8], 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

[7] 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

[8] 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.

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 very difficult to handle when dealing with typing issues because each subroutine can be called from multiple places, requiring that type information be merged which gives 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 are each prepended by a `pop` 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 (RLAII)

Patterns in bytecode produced by `javac` can be examined for areas of possible obfuscation. This simple obfuscation looks for situations where a local variable is used directly following both paths of an `if`. That is, along both branches the first instruction loads the variable on to the stack. This is a somewhat common occurance.

The 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 change, any decompiler relying on pattern matching (such as Jad) will become very confused.

6.3 Disobeying Constructor Conventions (DCC)

The Java language specification [8] 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.

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 constructors with no valid source code representation. This obfuscation randomly chooses among four different approaches 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` before the super call and creates a new try block 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` directly before the super constructor call: The `jsr`'s target is the `pop`, which removes the subsequent return address that is pushed on the stack by the `jsr`. 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. A `push null` is inserted, followed by a `throw`, along the false branch of the `if`. A 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` (`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. The Java construct allows only well-nested and structured uses, but the bytecode implementation is at a lower 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 matching 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. They 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. An example of this is the switch construct. In source, the switch 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 *must* reproduce the trap in the output, 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` always produces predictable try blocks it is possible to abuse them 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 is difficult for decompilers to synthesize equivalent source code.

6.7 `Goto` Instruction Augmentation (GIA)

Explicit `goto` statements are not allowed in Java source.[9] One must use abrupt statements instead. However, the `goto` exists in bytecode. It is possible to insert an explicit `goto` in bytecode. While reversible using control flow graph analysis, some simple decompilers will still struggle with this.

Our obfuscation 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. One is made the first instruction in the method and points to the start of P_1. The other is appended to P_1 and targets P_2. The new layout is now: $\{$ `goto` P_1, P_2, P_1, `goto` $P_2\}$. A try block is then created, spaning from the end of P_2 to the beginning of P_1, thereby "gluing" the two together. This makes it difficult to shuffle them 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 3 shows that both Jad and SourceAgain fail all tests and Dava is only successful once. Jad generates source with much bytecode left in it, making it difficult to identify anything specific as the cause. SourceAgain was unable to analyze the scope of local variables. It would declare a local within a nested block even when the parent block used that local. Both SourceAgain and Dava had difficulties marking methods which might throw exceptions. They also could not recognize the super constructor method calls in DCC

[9] Studies have shown this to be a good design decision [3].

either, leaving the bytecode name `<init>` which is not legal. Dava crashed on DCC due to its inability to handle explicitly null exceptions.[10]

Table 3. 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

7 Empirical Evaluation

An important aspect of our work is the evaluation of the impact of obfuscations on performance. To test this we have gathered a set of computation-extensive benchmarks. They represent a wide array of programs each with their own unique coding style, resource usage, and ultimate task. Below is a list of brief descriptions of the programs.

Asac: compares the performance of the Bubble Sort, Selection Sort, and Quick Sort algorithms. It creates a thread for each algorithm.
Chromo: runs a genetic algorithm; a technique using randomization instead of a deterministic search strategy. It instantiates many chromosome objects and performs many 64-bit array comparisions for each generation it simulates.
Decode: implements Shamir's Secret Sharing algorithm for decoding encrypted messages.
FFT: performs fast fourier transformations on double precision data.
Fractal: generates a fractal image. It performs many trigonometric functions and is deeply recursive.
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 of pi.
Triphase: contains three programs: (1) a Linpack linear system solver performing heavy floating-point math; (2) a multithreaded matrix multiplier; and (3) a multithreaded Sieve prime-finder algorithm.

7.1 Impact of Obfuscations on Performance

Figure 1(a) summarizes the ratio of the execution times of the obfuscated benchmark to the original benchmark.[11] A ratio of 1 indicates no effect on performance, a ratio of

[10] 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.

[11] 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.

less than 1 indicates that the obfuscated benchmark was faster, and a ratio greater than 1 indicates that it was slower.[12] Each bar corresponds to one obfuscation, the diamond on the bar indicates the average over all the benchmarks. The bars show the range of ratios with the bottom of the bar indicating the benchmark with the lowest ratio and the top of the bar corresponds to the benchmarks with the highest ratio.

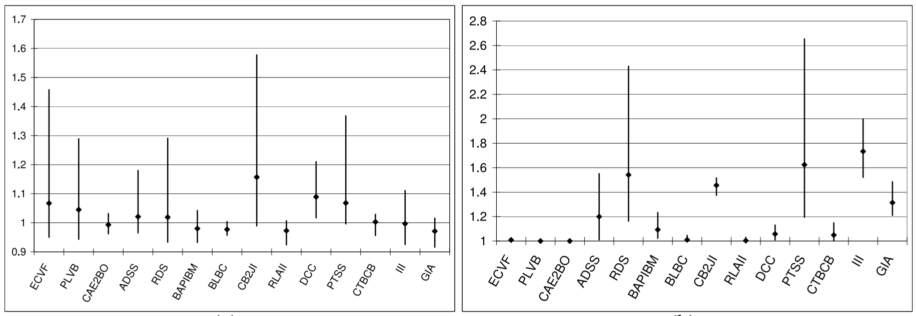

Fig. 1. Comparing obfuscated programs to their original forms: (a) Performance Ratio — (average execution time of obfuscated program)/(average execution time of original program); (b) Complexity Ratio — (sum of edges and nodes in obfuscated CFG)/(sum of edges and nodes in original CFG).

All experiments were run on an AMD Athlon™64 X2 3800+ machine with 4GB of RAM running Ubuntu 6.06 Linux. Sun Microsystem's Java HotSpot™64-Bit Server VM (build 1.5.0 06 b05) was used with the initial and maximum Java heap sizes set to 128MB and 1024MB, respectively.

As shown by recent empirical studies by Gu *et al.* [9, 10], 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. Notable performance differences are those less than .95 or greater than 1.05.

Average performance of the obfuscated code is very reasonable and quite a few are, in fact, faster. The most expensive is CB2JI, which converts branches to jsr instructions, with an average slowdown of 1.16 and a maximum slowdown of almost 1.6.[13] Only 6 obfuscations lead to a maximum slowdown > 1.2. These should be used carefully, avoiding hot methods if possible.

In some cases the obfuscations actually seem to slightly improve peformance. The RLAII obfuscation that moves loads above ifs is one such case.[14]

---

[12] The execution time is computed by timing 10 runs, dropping the slowest and fastest and averaging the remaining 8. The largest standard error we saw was 2.6% and most measurements were well below that.

[13] The maximum slowdown was in the LU benchmark and we found th entire slowdown was caused by one deeply nested loop which had very complex control flow after obfuscation. The JIT compiler struggled to analyze this, causing a 5-fold slowdown in compilation time.

[14] This makes 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

Figure 1(b) shows the increase in code complexity due to obfuscations (the pain). We have opted for a simple measure of complexity based on the total number of nodes and edges in the control flow graphs of the program. Each node is a basic block and each edge is a control flow edge. Obfuscations which change the structure of the code may introduce new edges and/or redirect existing edges to split basic blocks. Figure 1(b) displays the ratio of the sum of nodes and edges of the obfuscated code over the sum of the original. This count captures the impact of control flow obfuscations well.[15]

Some structure obfuscations show a signficant increase in complexity.[16]

As we have shown in Table 3, the third group of obfuscations are those that are most effective in breaking decompilers. Some of these also show significant increases in 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 Conclusions and Future Work

Fourteen obfuscations have been presented. The intent was to hinder reverse engineering while maintaining performance. The operator-level techniques are intended to make the code less readable. We didn't expect these to break decompilers, yet several decompilers failed to properly type the obfuscated code. The structure obfuscations were meant to confuse control flow and object-oriented design. The decompilers also had trouble with some of these techniques, although they should in principle be decompilable. These failures were mostly due to obfuscations creating unstructured control flow which is more difficult to handle than structured control flow. The gap obfuscations were new techniques and were aimed at exploiting the differences between bytecode and Java source. These were very successful in increasing the complexity of the code and breaking the decompilers.

The effect on performance varied. The average performance ratio of obfuscated/original ranged from .96 to 1.16. The maximum ratio reached almost 1.6 but only 6 of 14 obfuscations were over 1.2. These 6 should not be used heavily in hot methods of a program. More detailed analysis of specific instances showed that performance slowdowns were often due to the increased time needed by the JIT compilers to analyze the complex control flow created by our modifications. Hence the obfuscations are not just more difficult for reverse engineers to understand, they also cause problems for tools like compilers and decompilers.

We presented obfuscations we developed and this paper has shown how they work individually. The next step is to develop techniques to automatically determine optimized obfuscation sites and how to best select a combination of obfuscations so that

[15] As expected, the operation-level obfuscations have no impact on control flow complexity. Complexity for these obfuscations is better demonstrated by an increase in the number of operations. We have collected these kinds of metrics, which do demonstrate an increase.

[16] The two obfuscations that confuse the object-oriented design, BAPIBM and BLBC, do not increase complexity, but would affect other metrics which measure coupling.

the best overall protection is acheived. We have also started to develop metrics to quantify the effect of obfuscators and decompilers.

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. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.
9. 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.
10. 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.
11. S. Henry and K. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.
12. Jad - the fast Java Decompiler. Available on: `http://www.kpdus.com/jad.html`.
13. 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.
14. J. Miecznikowski and L. Hendren. Decompiling Java using staged encapsulation. In *Proceedings of the Working Conference on Reverse Engineering*, pages 368–374, October 2001.
15. Mocha, the Java Decompiler. Available on: `http://www.brouhaha.com/~eric/computers/mocha.html`.
16. J. C. Munson and T. M. Khoshgoftaar. Measurement of data structure complexity. *J. Syst. Softw.*, 20(3):217–225, 1993.
17. N. A. Naeem and L. Hendren. Programmer-friendly decompiled Java. In *Proceedings of the 14th IEEE International Conference on Program Comprhension*, 2006.
18. 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.
19. 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.
20. Source Again - A Java Decompiler. Available on: `http://www.ahpah.com/`.
21. 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.