JAVA BYTECODE OBFUSCATION

by

Michael R. Batchelder

School of Computer Science
McGill University, Montréal

January 2007

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Programs written for machine execution will always be susceptible to information theft. This information can include trademarked algorithms, data embedded in the program, or even data the program accesses. As technology advances computer scientists are building more and more powerful tools for reverse-engineering such as decompilers.

The Java programming language is particularly open to reverse-engineering attacks because of its well-defined, open, and portable binary format. We examine one area of better-securing the intellectual property of a Java program; obfuscation. Obfuscation of a program involves transforming the code of the program into a more complex, but semantically equivalent representation. This can include the addition of confusing control flow, the removal of certain information embedded in the program which is not explicitly required for execution, or the cloaking of data.

Obfuscation is one of the only techniques available other than cryptological options. While many approaches to obfuscation are ultimately reversible, it nevertheless seriously hinders those attempting to steal information by increasing the computing time and power required by software to reverse-engineer the program and also severely increases the complexity of any source code that is recovered by the reverse-engineering.

In this thesis we present a number of obfuscating transformations implemented within a new automatic tool we name the Java Bytecode Obfuscator (JBCO). We present empirical measures of the performance costs of these transformations in terms of execution speed and program size. Complexity measurements that gauge the effectiveness of the obfuscations are also given. Finally, we review the feasibility of each transformation by looking at source code generated from obfuscated bytecode by various decompilers.

Résumé

Les programmes écrits pour l'exécution dordinateur seront toujours susceptibles au vol d'information. Cette information peut inclure des algorithmes de marque de commerce, des données incluses dans le programme, ou même des données concernant les accès de programme. Suivant les avancées technologiques, les informaticiens construisent des outils de plus en plus puissants pour lingénierie inverse telle que le décompilateur.

Le langage de programmation de Java est particulièrement ouvert aux attaques de lingénierie inverse en raison de son format binaire bien défini, ouvert, et portatif. Nous recherches portent sur un domaine permettant de mieux sécuriser fixer la propriété intellectuelle des programmes en Java ; obscurcissement. L'obscurcissement d'un programme implique de transformer le code du programme en une représentation plus complexe mais sémantiquement équivalente. Ceci peut inclure l'addition de l'écoulement embrouillant de commande, de la supression de certaines informations incluses dans les programmes dont l'exécution n'est pas spécifiquement exigée, ou de la dissimulation des données.

Excepté les techniques cryptologiques, l'obscurcissement est l'une des seules techniques disponibles. Même si beaucoup de stratégies de lobscurissment sont finalement réversibles, il gêne sérieusement ceux qui essayent de voler l'information en augmentant la durée de calcul et la puissance exigées par les logicels dingénierie inverse et augmente considérablement la complexité de n'importe quel code source récupere par cette technique.

Dans cette thèse nous présentons un certain nombre de transformations dobscurcissement mises en application dans un outil automatique que nous appelons le Java Bytecode Obfuscator (JBCO). Nous présentons des mesures empiriques des cots d'exécution de ces transformations en termes de vitesse d'exécution et taille de programme. Des mesures de complexité qui mesurent l'efficacité des obscurcissements sont également indiquées. En conclusion, nous passons en revue la praticabilité de chaque transformation en regardant le code source généré par lobscurcissement de bytecodes par divers décompilateurs.

Acknowledgements

I would like to thank my supervisor Dr. Laurie Hendren for her discussion, ideas, and support throughout the research and writing of this thesis. I would also like to thank fellow graduate student Nomair Naeem as well as the rest of the Sable Research Group.

Finally, I would like to thank my parents for the opportunities they have given me that, had they been unavailable to me, would have prevented this thesis from ever happening.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1
Introduction and Motivation

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.

Historically, well known reverse-engineering cases have centered around military applications. As a specific example, consider the Tupolev Tu-4 Soviet bomber. In 1945, near the end of World War II, the Soviet Union was lacking in strategic bombing capabilities. To remedy this, they captured three United States B-29 Superfortresses flying through their airspace and dismantled them for reverse-engineering purposes. The Tupolev Tu-4 was the result of this effort, as reported by CNN [swr01].

More recently, reverse-engineering has become much more widespread and viable - no doubt through the proliferation of information available on the Internet. Even consumers are using the technique to expand the features and functions of popular electronic devices such as the Apple iPod$^{\text{TM}}$media player (to circumvent digital rights management (DRM) as well as to expand the functionality to include new games and even a book reader) and the Microsoft XBox$^{\text{TM}}$gaming console (DRM circumvention, as well).

It could be argued that software has proven to be among the most susceptible forms to reverse-engineering. With the advancements of computer systems in our everyday lives the examples of the iPod and the Xbox are becoming the norm, not the extreme cases. Because 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 encryption where reverse-engineering is explicitly thwarted.

Nevertheless, software reverse-engineering is a broad field. Within the realm of software there exists both legitimate and illegitimate reasons for reverse-engineering.

A legitimate and legal example of software reverse-engineering would be a situation where the copyright owner of a system has lost documentation or other important information and would like to reproduce it. Since they own the rights to the software (not to be confused with the right to *use* the software) they can do as they will. An illegal example would be a situation where a competitor steals the underlying implementation - the algorithms themselves - and incorporates them in some form in their own product.

There is, of course, a grey area. United States law, for example, stipulates that reverse engineering anything that is patented can be considered breaking the law, as one would expect; however, if the item or software is protected by trade secrets instead of by a patent, then reverse-engineering is lawful given the item itself was obtained through legal means, according to Title 17 of the United States Code [Uni]. A common catalyst for an entity to do this is, ironically enough, to ensure that a competitor's offering does not infringe on any of the entity's own patents.

1.1 The Vulnerability of Java

The Java language was designed with modern computing problems in mind. Specifically, it is compiled into a format called bytecode, organized in *class* files, which is platform independent. This means that the code can easily be executed on any of a number of hardware architectures without the need to re-write or convert the class files. However, it also means that bytecode is an intermediate representation of the program which is not encoded for any specific machine. The downfall of this design is three-fold. Firstly, the compiler cannot optimize the output for any particular hardware. Secondly, the bytecode must be interpreted. Finally, much of the higher-level information contained in source code which is stripped out during the compilation process of a normal language is *not* stripped out of bytecode.

Optimizing compilers that are aware of the architecture for which they are generating code will produce instruction sequences designed to execute quickly on that machine. For example, some instructions may be re-ordered to take advantage of the particular CPU's instruction caching feature.

Interpreted code is executed by an interpreter. In Java's case, this interpreter is part of a Virtual Machine. It is virtual in the sense that it simulates an abstract machine that understands and executes bytecode within a real machine. Because this virtual machine must be implemented on any platform where someone wants to run Java programs, its specification must be fairly simple and open.

Because Java bytecode is encoded in a fairly simple way, is not optimized, and contains much of the higher-level information that its source code did, Java is particularly susceptible to reverse-engineering attacks. Information is power; by retaining information other than that needed by the machine to execute the code, the class file format is less of a black box than other platform-dependant language binaries. Optimized code, while semantically equivalent to its original, can be quite confusing and appear even more complex especially when special architecture instructions are employed. Since bytecode is not optimized, it appears simpler and clearer.

1.2 Protection Schemes

There is more than one way to skin a cat and the same can be said for protecting the intellectual property and secrecy of algorithms contained within software. While some approaches have grown out of ulterior motives, with software protection being a side-effect, others are more intentional and direct.

1.2.1 Server-Side Execution

With the birth of the computer network, a new age was born. Programs and data are no longer tied to a single computer. This situation was furthered by the expansion and world-wide acceptance of the Internet. Software is no longer chained to a specific machine and, in fact, the user may not even be aware of this.

In order to attack or decompile a piece of software, access to the code itself must be obtained. If the code cannot be analyzed then it cannot be reverse-engineered except through blind attacks, a fairly inefficient approach. This fact is the basis for protection through server-side execution.

The server-side execution model (also known as the client-server model) is one in which the application is placed on a server machine. The actions of the software can be controlled remotely by a client machine and services, such as displaying data, can be delivered to the client machine, yet the software is physically stored and executed on the server. In this model, the client software - while itself a program - is never given access to the important server program code.

The major disadvantage of the server-side model is the very nature of its separation between interface and functionality. By creating a divide between the client and the server an inherent weak point is created at the divide as well. Low bandwidth between the client and server could hinder performance and, in the worst case, disconnection altogether would result in an inability to use the software. Even an inconsistent connection could result in the inability to use the software or in the corruption of data being passed over the network.

1.2.2 Encryption

Encryption is another possible form of program protection. In this model the software code is accessible to the attacker, but it is encrypted in some way. In order for the software to run successfully, it must be decrypted at runtime.

Decryption could be performed at the machine level but would require specialized hardware, which is impractical. Even Trusted Computing, backed by such large technology companies as Hewlett-Packard, IBM, Intel, and Sun Microsystems, Inc. has had its fair share of proponents based on practicality issues. Trusted Computing is a technology initiative to ensure that computer hardware can be trusted by its designers to not run unauthorized programs (see the Trusted Computing Group FAQ [Gro]).

Software-level decryption, while also possible, is inherently weak security because the decryption system itself can be reverse-engineered.

1.2.3 Obfuscation

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 primary 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. Another example of obfuscation is the use of opaque predicates and introduction of unnecessary control flow. An opaque predicate is an expression that will always evaluate to the same answer (`true` or `false`) but whose value at runtime, upon static investigation of the code, is not obvious. Such opaque predicates can be used to introduce new if statements or other control flow constructs within a method (essentially adding sections of dead code — instructions which will never be executed because they are along the `false` branch of an opaque if instruction).

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.

While almost all obfuscations are technically reversible the technique as a whole is nevertheless an attractive protection approach for a number of reasons. It can be applied during the software release phase, as part of the packaging process; it need not be tied to the design or development phase whatsoever. Additionally, it does not rely on extra hardware or software to run in the way that other schemes do. Server-side execution obviously requires a network and at least two machines, one server and one client. Encryption requires either hardware or software decryption that is itself susceptible to attack and can slow the program down. Finally, obfuscation is cheap. Once implemented within an automatic tool, obfuscations can be applied to many programs and, indeed, commercial obfuscators exist for reasonable prices.

1.3 Thesis Contributions and Organization

The threat of reverse-engineering and the particular security concerns for Java have been outlined in this first chapter. Potential protection schemes have also been discussed, including code obfuscation which is the topic of this thesis. It is clear that obfuscation, while not fool-proof, is worth further exploration and could provide a potential cover of protection against the thievery of intellectual property or the side-stepping of copyright law.

In this thesis we discuss and develop a number of obfuscating code transformations which we break into three categories.

- Obscuring intent at the operational level. Prime examples are converting multiplications into less obvious bit-shifting operations and identifier renaming.
- Confusing or complicating program structure. This includes using opaque predicates to introduce extra control flow branches in a method as well as higher-level obfuscations which affect the object-oriented design of a program.
- Exploiting the gap between the Java source language and its binary bytecode form. The classic example of this type of obfuscation is the insertion of explicit `goto` instructions at the bytecode level, a construct that is not present in Java source language.

We implement these obfuscations as a module within Soot [VRHS+99] — a Java optimization framework — and develop a user interface for applying them to Java programs. Additionally, each transformation is tested separately to ascertain the runtime performance effect that can be expected from them. Given the empirical results and the observed effectiveness of obscuring intent of each transformation, we derive a combination of obfuscations and how often to apply them as a default to our obfuscator.

The remaining chapters are arranged as follows. Chapter 2 outlines the research that has already been done in the area of obfuscation. Particular care is taken in comparing recent Java bytecode obfuscation techniques.

Chapter 3 describes, in detail, the various features of the Java language and its runtime environment. This includes the bytecode instruction set, the Java Virtual Machine, and the bytecode verifier.

Chapter 4 introduces our approach to obfuscation including the framework that is used. Flow analysis and bytecode instrumentation are addressed. Our experimental framework is then outlined and a number of benchmark programs - our testing suite - are described.

The meat of this work is given in chapters five through seven. Chapter 5 describes the class of obfuscating transformations that we have developed which affect a program at the operational level. The implementation is outlined in detail and examples are given which display each transformation's effect on code by showing before and after (decompiled) source code. Where appropriate, bytecode is given to describe the exact technical changes that are happening "under the hood". At the end of each section experimental results are given which measure the performance degradations one can expect when fully applying each of the transformations.

Chapters 6 and 7 follow the outline of Chapter 5 in detailing the other two classes of obfuscations, confusing program structure and exploiting the gap between the Java language and bytecode, respectively.

In Chapter 8 we discuss the interface of our obfuscator and develop a comprehensive combination of transformations for "the masses" — a default setup — to ensure the obfuscator we develop is easy to use even for those not familiar with the low-level under-pinnings of the Java language. We present experimental results for this default setup that shows the performance effect on each benchmark. Next, we show example decompiled code that expresses the effectiveness of our obfuscations.

Finally, in Chapter 9, we suggest further areas of research on this topic for future work and we summarize our conclusions.

Chapter 2
Related Work

There has been a lot of research into code obfuscation in recent decades. Originally, concerns about code theft drove the majority of advancement but more recently issues of security have been in the limelight. Computing has become increasingly more pervasive in our everyday lives. More and more personal, financial, and social information is stored and processed by computers everyday.

2.1 Viability of Obfuscation as a Protection Tool

Obfuscation is a form of *security through obscurity*. While Eric Raymond is quoted as saying "Any security software design that doesn't assume the enemy possesses the source code is already untrustworthy; therefore, *never trust closed source*," [Ray04] obfuscation does not explicitly break this axiom. When securing code through obfuscation the assumption is that the enemy *can* produce the source code (though not the original) through automatic means such as decompiler, but that the obfuscations performed will render that code unreadable. While there are seemingly few truly irreversible obfuscations, according to Barak, *et al*. [BGI+01] and, in theory, "deobfuscation" under certain general assumptions has been shown to be NP-Easy through the work of Appel [App02], obfuscation is nevertheless a valid and viable solution for general programs. Philosophizing aside, deobfuscation is still not a simple task. Just as optimizing compilers make conservative approximations to avoid undecidable problems, so must decompilers. The result is sub-optimal decompiled code.

Indeed, by releasing our obfuscator as open source we are explicitly assuming that the enemy possesses not only the software code they are attacking but to the source code of the security system that protects it, as well (a tenant of security engineering in general derived from Kerckhoffs' 1883 principle [Ker83a, Ker83b]). Despite this openness, with over fifteen different transformations that can be applied on top of each other and in various orderings, reverse-engineering through automatic means still remains very difficult.

2.2 Low-Level Obfuscation

Early attempts at obfuscation invariably involved machine-level instruction rewriting. Many of these different techniques for obfuscation are considered within the low-level realm by Cohen [Coh93]. They dubbed their techniques "program evolution". This 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. Many of these ideas later became standard and we explore some of them in this thesis.

Much later, a more theoretical approach to obfuscations was presented by Collberg, *et al.* [CTL98]. They outline obfuscations as program transformations and develop a notation for them ($P \rightarrow P'$ where P is the program and P' is the transformed output). They develop terminology to describe an obfuscation in terms of affect and quality:

Potency : the level of obscurity a particular transformation gives.

Resilience : a measure of how well an obfuscation holds up against reverse-engineering attack.

Stealth : how difficult it is to detect whether the transformation has been applied to a program.

Cost : the performance and space (size) penalties incurred by the obfuscation.

They rely on a number of well-known software metrics developed by Chidamber and Kemerer [CK94], Henry and Kafura [HK81], and Munson and Khoshgoftaar [MK93] to measure the potency and they suggest many obfuscations. Some are design-level techniques such as false refactoring (creating dummy parent classes with possible shared variables and dummy methods) and method cloning. Others are data-obfuscations such as array restructuring through splitting or merging.

Later, Collberg and Thomborson [CT02] 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 contribution is in control-flow obfuscations. They make use of opaque predicates (discussed by 1.2.3) 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*. [WHKD00], whereby control and data flow analysis would become co-dependent. This resulted in much higher complexities and lower precision from static analysis tools. On a low-level, this involved using data values within switch statements with potential branching to every basic block of a method. Without any knowledge of which branch targets would be selected in which order, every basic code block of a method could potentially be a predecessor of every other block. While very potent, this approach can have a noticeable negative performance effect and bloat class file sizes as well. We considering a scaled-down version of this type of obfuscation in Section 6.1.

While not Java-specific, a very interesting two-process obfuscation approach which uses inter-process communication (IPC) to communicate between a "control-flow" process and a "computation" process was developed by Ge, *et al*. [GCT05]. Unfortunately, this kind of low-level jury-rigging is not possible in Java. A different multi-process technique for maintaining opaque predicates was presented by Majumdar and Thomborson [MT06] that could certainly be implemented in Java.

Taking a page out of the biologist's book, the idea of diversity as a means towards software protection was introduced by Forrest, *et al*. [FSA97]. They showed that, for example, the randomization of the size of a stack frame could deter a simple buffer overflow attack. Again, this was non-Java specific but interesting. They consider optimizations for parallel processing whereby blocks of instructions that can be run simultaneously are produced from sequential code, confusing the intention of the program. This could potentially be combined with the technique of speculative multi-threading (discussed by Pickett and Verbrugge [PV05]) in an ahead-of-time manner to produce useful Java obfuscation.

2.3 High-Level Obfuscation

The approach by Sakabe, *et al.* [SSM03] was more concentrated towards obfuscating 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. 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 opaque predicate if branching around new object instantiation which confuses the true type of the object and they use exceptions as explicit control flow. Sadly, their empirical results show significantly slower execution speeds — an average slowdown of 30% — and a 300% blowup in class file size.

More high-level obfuscations are presented by Sosonkin, *et al.* [SNM03] which attempt to confuse program structure. 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. By creating new Java interfaces that define the various original classes they are able to achieve type hiding and they claim their approach makes reverse-engineering very uneconomical.

2.4 Summary

At this point a lot of thought has been put into the general area of obfuscation. Some of the obfuscations we detail in the following chapters come from this rich history of research. In most cases very little empirical testing of these ideas seems to have been published, calling into question their usefulness in real world applications, and we wish to show that they have merit. Nevertheless, we have also detailed new and exciting approaches which we were unable to find within the literature, such as those obfuscations which seek to exploit the differences between the Java source language and its compiled bytecode format. These include inserting explicit `goto` instructions, disobeying source code constructor conventions, and taking advantage of the flexible nature of Java exception handling.

Chapter 3
The Java Language

Java is a high-level programming language that falls into the family of third-generation languages that includes C++ [Str97], Smalltalk [GR83], and many others. It has been designed to be easier for a human to understand, including such amenities as named variables within binaries (which aid in debugging), an object-oriented framework with inheritance, and exception handling functionality. While it is fairly similar to C++ and shares much syntax it is nevertheless very different in certain aspects.

Historically, industrial languages such as C were designed to be processed by an automatic tool called a compiler which would then generate machine code - also known as an "executable". In the last two decades, however, interpreted languages – especially Java – have had a major renaissance. Interpreted languages are those which postpone some or all of the "compiling" phase to the time of execution.

Java is not compiled fully and is instead converted into the class file format. This format maintains much of the context and high-level information contained in the source code and is not directly related in any way to a specific hardware architecture or software operating system. Java source that has been compiled into class files can be thought of as being "half-way compiled".

The second half of the compilation process is abstracted into a runtime feature; that is, Java class files are executed by being fed to a virtual machine, which is itself a piece of software. This virtual machine software interprets each bytecode instruction in the class file into a proper sequence of machine instructions. While this sounds like simple translation, it is not exactly that. The virtual machine also runs optimization routines on bytecode fragments that execute frequently (*i.e.*, hot methods) and it verifies the security of the bytecode when it is first loaded.

In addition to the "second-half compilation" part of the virtual machine, it also maintains virtual registers, a program counter, and all of the trappings of a normal hardware machine but emulated in software. The negatives of this approach should be obvious: Java programs have historically been slower in execution speed and larger in memory footprint than their compiled cousins such as programs written in C. Yet the performance gap has been consistently narrowed year after year and the positives to the Java approach are many.

By creating a single trusted and fully tested virtual machine for each possible platform, *any* Java bytecode can be run on any of these platforms. This is the fundamental concept behind Sun Microsystem's slogan for Java: "write once, run anywhere". This means that a developer can write Java code on a personal computer (PC) and reasonably expect it to run properly on any device with a Java virtual machine including cellphones or other embedded devices.

What's most special about Java in relation to other programming languages, however, is that it was designed in the early days of the World Wide Web and, because of this, has some very forward-thinking design approaches unique to the Internet medium. Java lets you write programs called "applets" (a diminutive reference to the word "application" which is often used to describe a program). These applets can be easily transferred over the Internet and executed within a web browser. Technically speaking, the program is run by the virtual machine and the output is embedded in the window of a web browser. However, what is unique to the applet is its limited access to the computer system on which it is being executed. A traditional fully-compiled program, such as those discussed above written in C, has almost limitless access to the computer system it is running on — only recently, in this decade, with the rise of both Internet users and computer viruses have operating system designers begun to truly consider these security implications. It is no longer viable to trust software posted on the Internet and, indeed, virus-detecting software is installed by default on most new consumer machines.

Java solves these security problems by severely limiting what an applet can do. It cannot write data to one's hard disk without permission, it cannot connect to an arbitrary machine on the Internet, and it should not crash your system. Without these assurances, a Java applet could delete important data, maliciously send sensitive data over the Internet, or worse. Additionally, even normal Java applications have limited access through the very design of the virtual machine. They cannot write to arbitrary addresses in memory, for example, and all variables must be defined before they are used. This scheme is sometimes referred to as the "sandbox model" where programs are run within an insulation layer (the virtual machine) and not allowed direct access to hardware. This scheme is built on three main points:

- Java programs are compiled to bytecode, not machine code. Code produced by the Java compiler works on high-level abstractions of data such as object references instead of memory addresses.
- Access is controlled and limited. A program must use APIs (Application Program Interface) known to be safe and trusted to interact with hardware resources.
- Bytecode is run through a "bytecode verifier" (sometimes referred to simply as the verifier) which statically analyses the program to ensure type correctness, and that the first two items in this list are adhered to.

The last item in this list is particularly crucial to the secure design of Java and will be thoroughly discussed later in Section 3.3.

3.1 The Instruction Set

Bytecode is intermediate code. It is more abstract than machine code but less abstract than the source code which humans read and write. Its name is derived from the fact that each opcode can usually be represented by a single byte in length. Each instruction, therefore, has a byte representing the opcode possibly followed by parameters such as a register index, or a value.

When a Java method is executed, a new "frame" in memory is created with space for abstract memory registers and a stack. The maximum space requirements of each method is

```
public static int sum(int iarry[]) {
  int result = 0;

  for (int i = 0; i < iarry.length; i++) {
    result += iarry[i];
  }

  return result;
}
```

Listing 3.1: *Example method* `sum` *in Java source code.*

known, through static analysis at compile time, and encoded in the bytecode. How the stack and registers are physically allocated and accessed is not important - the virtual machine takes care of these details.

To get a flavor of the bytecode instruction set, we will describe the three common instructions `load`, `store`, and `add`. The `load` instruction takes a single integer argument specifying a register index. The instruction obtains the value in the register at that index and places the value on top of the stack. The `store` operation performs the opposite function; it pops a value off the top of the stack and stores it in the register at the index specified by the parameter. Finally, `add` is an example of an instruction with no parameters. It pops the top two values on the stack and adds them together, pushing the result on top of the stack.

In total there are 256 opcodes, although some are reserved. This may seem like a large number but in fact there are specific opcodes for the various forms of each instruction. The `add` instruction, for example, is split into different versions, one each for values that are integers, floats, and doubles. The `load` instruction is likewise split, with the addition of an object version for loading object references. A traditional way for marking the various forms of instructions in code printouts is by adding a single-character prefix — `i` for integer, `f` for float, `a` for object, *etc.*— so the instruction which loads an object reference from the register at index 2 would be printed as “`aload 2`”. A brief fragment of Java source code is shown in Listing 3.1 and its equivalent bytecode (formatted by the `javap` disassembler utility) is shown in Listing 3.2. We will follow this form of representation throughout this thesis.

```
public static int sum(int[]);
    0: iconst_0
    1: istore_1
    2: iconst_0
    3: istore_2
    4: iload_2
    5: aload_0
    6: arraylength
    7: if_icmpge 22
    10: iload_1
    11: aload_0
    12: iload_2
    13: iaload
    14: iadd
    15: istore_1
    16: iinc 2, 1
    19: goto 4
    22: iload_1
    23: ireturn
```

Listing 3.2: *Example method* `sum` *in Bytecode (as output from the* `javap` *disassembler utility).*

3.2 The Virtual Machine

The Java Virtual Machine (JVM or VM for short) by Gosling, *et al.* [GJSB00] is a software emulator of an abstract stack-based machine. Instructions supported by this machine operate on the stack by popping, pushing, or altering the value of the item(s) on the top of the stack. Registers are also available for variables and these are populated by using one of the various "store" instructions to pop a value off the top of the stack and place it in a given register. The "load" instructions retrieve a value from a register and places it on top of the stack. These registers are encapsulated in a frame and are only accessible by the specific method which created them so they are most often referred to as "local" variables (local to the given method only). These registers and the stack are maintained across method calls so that if execution control is passed from one method into another, the original method maintains its variables and can access them when control is returned to it from the second method.

3.2.1 Compilation

The compilation system in the Java world is different from that of statically compiled languages like C. Since the Java compiler produces machine-independent bytecode, there is very little optimization that takes place at compile time. The optimizations that would be done by the compiler in a statically compiled language are instead performed at or during runtime by the VM.

The simplest implementation of a JVM is just a bytecode interpreter. This is because, before any native compilation happens, the JVM simply interprets the bytecode instructions one by one, in sequence.

3.2.2 Just-In-Time compilation

Interpretation is particularly slow, since each and every bytecode instruction must be translated into native code by the JVM each time it is executed. Just-In-Time compiling (JIT) speeds up execution by converting all bytecodes into machine code before that method is first executed. The caveat: it does so lazily. This means that it only compiles a method when the method is actually called to run. The benefit of this design is that it spreads the compilation performance-cost over the life of the running program (or at least until all methods are run once) instead of resolving everything at the beginning, therefore improving start up times. Nevertheless, JIT compilation time can still be significant and aggressive optimization of every part of the code is not possible.

3.2.3 Dynamic compilation

In newer JVMs, such as Sun Microsystem's HotSpot^{TM} [Mic01] virtual machine, the interpreter, a profiling mechanism, and a dynamic compiler are combined together to make up the execution framework. As the name suggests, the HotSpot first runs as an interpreter and only compiles methods which are being frequently executed (as reported by the profiler). The benefit is that time is only expended optimizing and compiling code that is run often — in most cases 90% or more of a program's dynamic execution time is spent on less than 10% of its static code, as shown by Merten, *et al.* [MTG+99].

The HotSpot optimizations are many, and are probably recognizable to anyone who is familiar with optimizing compilers. It performs common subexpression elimination, loop unrolling, data-flow analysis, and more. Perhaps most impressive is its optimization of its optimizations. After interpreting a method a few times, it is then compiled into native code but the JVM still collects profiling for this method. It may then go back and recompile the method later with higher optimizations if the profiling data shows that it is a very hot method.

The newest JVMs use a feature known as on-stack replacement (OSR), sometimes referred to as an "adaptive compilation" technique. This allows newly-compiled code to be activated even when the method is currently being executed.

3.3 The Bytecode Verifier

The Java verifier is a subsystem of the VM which statically evaluates class files when they are loaded, but before they are executed. It is part of what entails the "sandbox" model of Java.

The verifier performs static analysis in order to ensure that the code is well typed and does not contain illegal code. It is able to detect ill-typed operations such as casts from primitives to object references or vice-versa as well is illegal casts from one object type to another. It also ensures proper object-orientedness by checking that private methods are not called outside of a class, that interfaces are correctly implemented, and that code does not attempt to illegally jump outside of a method or otherwise.

The impetus behind the verifier is two-fold. While its chief intent is to act as a security module for the VM, limiting the potential for malicious attacks, it also protects against unintended problems such as encoding errors due to transmission issues or improper method calling that was not malicious in nature. It accomplishes this through program analysis.

There has been research in the past into dynamic checking of code but Cohen [Coh97] has shown this to be expensive and slow. The bytecode verifier instead applies static checks only once, at runtime. It attempts to ensure that these conditions, as outlined by Leroy [Ler01], are met:

Type correctness : arguments of an instruction, whether on the stack or in registers, should always be of the type expected by the instruction.

No stack overflow or underflow : instructions which remove items from the stack should never do so when the stack is empty (or does not contain at least the number of arguments that the instruction will pop off the stack). Likewise, instructions should not attempt to put items on top of the stack when the stack is full (as calculated and declared for each method by the compiler).

Code containment : execution flow of a program should never jump from the inside of one method to the inside of another method. Likewise, code should only jump to an offset which is the beginning of a valid instruction and never into the middle of one.

Register initialization : Within a single method any use of a register must come after the initialization of that register (within the method). That is, there should be at least one store operation to that register before a load operation on that register.

Object initialization : Creation of object instances must always be followed by a call to one of the possible initialization methods for that object (these are the constructors) before it can be used.

Access control : Method calls, field accesses, and class references must always adhere to the Java visibility policies for that method, field, or reference. These policies are encoded in the modifiers (private, protected, public, *etc.*).

Within this thesis each and every obfuscating transformation must follow these guidelines to ensure that code output by the obfuscator does not fail the bytecode verification. Otherwise, transformed code would be useless.

Chapter 4
Obfuscation Approach and Experimental Framework

We developed an automatic tool called JBCO - the Java ByteCode Obfuscator - that is built on top of Soot [VRHS+99] in order to implement the obfuscations outlined in this thesis. Soot is a Java bytecode transformation and annotation framework providing multiple internal representations. It is a framework that has been designed to simplify research and analysis of Java and its class file format to aid in new compiler optimizations and program testing. It includes the ability to instrument code - that is, to modify through adding or removing information, control flow, or program logic.

4.1 Soot: a Java Optimization Framework

During normal execution of Soot, it first transforms Java source code or bytecode into Jimple, a 3-address intermediate form. Jimple is not stack-based, like bytecode, but maintains some low-level constructs such as the explicit goto, unlike Java source code. It also has a static inference engine which can identify variable types in bytecode, explained by Gagnon, *et al.* [GHM00]. Listing 4.1 shows a simple method, `sum`, in Java source code and Listing 4.2 shows the same method in Jimple.

Once in Jimple, each method is run through transformations and analyses which are written for Jimple, possibly modifying or annotating the code. Translation to Baf, a stack-

```
public static int sum(int iarry[]) {
  int result = 0;

  for (int i = 0; i < iarry.length; i++) {
    result += iarry[i];
  }

  return result;
}
```

Listing 4.1: *Example method* `sum` *in its original Java source code form as written by a software programmer.*

```
public static int sum(int[]) {
    int[] r0;
    int i0, i1, $i2, $i3;

    r0 := @parameter0: int[];
    i0 = 0;
    i1 = 0;

  label0:
    $i2 = lengthof r0;
    if i1 >= $i2 goto label1;

    $i3 = r0[i1];
    i0 = i0 + $i3;
    i1 = i1 + 1;
    goto label0;

  label1:
    return i0;
}
```

Listing 4.2: *Example method* `sum` *in Jimple form. Dollar signs indicate local variables which are stack positions only.*

```
public static int sum(int[]) {
    word r0, i0, i1;

    r0 := @parameter0: int[];
    push 0;
    store.i i0;
    push 0;
    store.i i1;

  label0:
    load.i i1;
    load.r r0;
    arraylength;
    ifcmpge.i label1;

    load.i i0;
    load.r r0;
    load.i i1;
    arrayread.i;
    add.i;
    store.i i0;
    inc.i i1 1;
    goto label0;

  label1:
    load.i i0;
    return.i;
}
```

Listing 4.3: *Example method* `sum` *in Baf form. The single character following a dot on the end of an instruction indicates the type of the instruction* (e.g., *i for integer).*

based intermediate representation follows. Transformations and analyses written specifically for Baf are then processed. Instructions in Baf, while similar to bytecode instructions, are statically typed. Making this information available to JBCO allows for more flexibility and simpler implementations (Listing 4.3 shows `sum` in Baf). Finally, Baf is translated into bytecode and written to class files.

4.2 JBCO: The Java ByteCode Obfuscator

JBCO is itself just a number of Jimple and Baf transformations and analyses. There are three categories that each module falls under:

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. Often, they are recalculated each time the code is modified, allowing a transformation to perform multiple modifications to the same piece of code. Without this dynamically-updated information each transformation would have to be run over and over, adding overhead to the obfuscator.

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

4.2.1 Information Aggregators

The simplest of the JBCO tools are those which iterate through the Jimple or Baf code in order to collect information. One such example is the *Constant Collector* which collects references to all constant data embedded in the program, including integers, floats, doubles, and even Strings. Later, a Baf instrumenting transformation will use the information generated by the Constant Collector to obfuscate the program by replacing the constants with static field references.

Other information aggregators found in JBCO are concerned with the details of local variables in the program. Jimple maintains high-level typing information and is able to distinguish between an integer and a boolean (both stored as a 32-bit integer in Java bytecode) but Baf does not; it only stores the size needed for the local (in this case, one 32-bit integer for each boolean). Because of this limitation in Baf, and the transformations we wished to apply at that level, we were required to implement aggregators which would properly map Baf local variables to their Jimple types.

4.2.2 Code Analyses

JBCO code analyzers are mostly implemented as dataflow analyses. Dataflow analysis performs a flow function on each node of a control flow graph, augmenting it with information based on its predecessor and successors, as well as its own code instruction and variable use. The algorithm proceeds to iterate over all nodes until a fixed-point is reached. Two examples follow.

Calculating Stack Height and Types

Many of the obfuscating techniques in the following chapters can only be applied in certain situations. Often, new opaque predicate `if` blocks are added, or random locations within the bytecode are singled out as possible targets for obfuscated control-flow. The obfuscator must know what arguments are on the stack before and after each instruction, including the types, in order to produce verifiable code.

The `StackHeightTypeCalculator` was implemented to derive this information. At any time during the obfuscating process, the number of arguments on the stack (the stack height) and their types can be derived through this analysis. It is a forward flow analysis (*i.e.*, it propagates information in the same direction as execution flows). The information it propagates is, obviously, what is on the stack. As each instruction is processed, depending on its type, arguments are added and/or removed from the stack. When a fixed point is reached, the state of the stack before and after each instruction is known[1]

Object Instantiation to Object Initialization Analysis

In bytecode new objects must first be "allocated" by the `new` instruction and then instantiated by a method call to one of its constructors. The VM's bytecode verifier is particularly picky about this and requires that, at all call sites to an object constructor, it must be statically verifiable that a newly-allocated object reference (of the proper type) is on the stack.

In order to properly implement the obfuscation outlined in Section 6.1, a special analysis needed to be performed during the obfuscation phase in order to ensure that the transfor-

[1]The exact Object types of references on the stack can not always be deduced but a greatest-common-object type can be.

mation does not separate `new` instructions from their matching constructor calls, creating unverifiable code. The Java Virtual Machine Specification by Gosling, *et al.* [GJSB00] states that an uninitialized object must not be on the stack or in a local variable when a backwards branch is executed, nor should there be an uninitialized object in a local variable within a trapped sequence of bytecode[2]. This limitation assures that no object can accidentally or maliciously be initialized more than once within a loop or otherwise, though Coglio [Cog01] argues that the requirements are not necessary to ensure type safety.

This special requirement resulted in the creation of the *New-to-Constructor Analysis*. It is a backwards flow analysis (*i.e.*, it propagates information in the opposite direction of execution flow) and the information it specifies at each node is whether the node may potentially fall between a `new` instruction and its corresponding constructor call. This information is used to ensure that the obfuscation does not add unverifiable control flow branches into these New-to-Constructor ranges.

4.2.3 Instrumenters

Instrumenters are the algorithms within JBCO that actually modify the program code. Also known as transformers, they augment or otherwise convert the code from its simpler origins to a more complex and obfuscated form.

Within the Soot framework there are two different types of transformations: those that operate on the program as a whole and those that operate on one method at a time. High-level program obfuscations - those which change the overall design or object-oriented nature, for example - are whole-program transformations. They require access to and the ability to modify all classes of a program; they are referred to as *Scene Transformers*. Method-level obfuscations are known as *Body Transformers* since they operate on the code bodies of each method.

While each obfuscating transformation implemented for this thesis is discussed in detail in the following three chapters, there is one "helper" instrumenting transformation that ensures that all local variables are defined.

[2]Uninitialized objects on the stack are allowed within exceptional bytecode ranges because if an exception occurs they will be removed from the stack completely

```
double d0, d1, d2, d3, d4, d5, d6, d7;
d0 = 0.7908512635043922;
d1 = 0.8632577915463435;
f0 = 0.27129328F;
d2 = 0.37580820874305876;
f1 = 0.64183664F;
d3 = 0.5397618625049699;
d4 = 0.8257518968619348;
d5 = 0.018002925462166197;
d6 = 0.24160555440326648;
d7 = 0.5634773313795401;
```

Listing 4.4: *"Dummy" values being assigned to local variables at the beginning of a method to assure the bytecode verifier that they are initialized.*

Fixing Undefined Local Variables

Some changes that are made to bytecode by our obfuscations can cause code-motion (movement of a block of code from one spot in a method to a completely different spot). Other transformations, which add opaque branching that never actually occurs, can make it appear as though certain orders of code could be executed in sequence when in fact they never will. This can result in situations where the bytecode verifier computes that a certain local variable might be used before it is initialized or it might seem like a different object type than it really is.

Since the bytecode verifier will not allow this kind of code to execute, it must be eliminated. A special transformation, the *Undefined Local Fixer*, was written to handle these cases. It finds all local variables which might appear undefined at some point in the code. For each one "dummy" initializations are inserted at the beginning of the method. While it only seems like the local might be used before it is assigned, in truth it will always contain a value. Therefore, the "dummy" value inserted to convince the bytecode verifier will always be overwritten before an actual, valid use. Listing 4.4 shows the first few lines of a decompiled method after some obfuscations. As can be seen, there are seemingly nonsensical values being assigned to local variables. These values will never actually be used.

4.3 Experimental Framework

Each transformation described in this thesis (detailed in Chapters 5, 6, 7) was thoroughly tested in multiple time trials in order to ascertain any effect on runtime performance.

All experiments were run on an AMD Athlon$^{\text{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$^{\text{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. Measurements were taken in both server mode (*i.e.*, with full dynamic JIT compilation) and interpreter mode. Client mode was not included as it is not available in this version of the VM. The graphs display a ratio of obfuscated program runtime over original program runtime. Therefore, a value of 1.0 implies no change in performance.

The Time::HiRes Perl module was used to record the running times of the experiments. Each benchmark was run ten times for each experiment and the highest and lowest time measurements were thrown out. The remaining eight were averaged and this is the number (in seconds) used to calculate percentage speedups/slowdowns shown in the graphs in this thesis.

However, as discussed by Kalibera, *et al.* [KBT05], there is an unavoidable amount of non-determinism in computer systems. The inability to start benchmark runs with the hardware and operating system in the exact same initial state every time leads directly to varying time measurements. Indeed, it was shown that initial state had a large effect on a fast-fourier benchmark despite running on an idle machine with disabled virtual memory allocation with the same files and settings. Furthermore, it has been shown by Gu, *et al.* [GVG06] that side-effects of lower level execution can account for almost 10% of measured performance and that instruction and data caching can be particularly sensitive. In some cases an obfuscation may result in programs showing significantly different execution speeds due to these types of issues which may be completely separate from the particular details of the transformation itself. Earlier work, by Gu, *et al.* [GVG04], shows that even code layout changes as trivial as identifier renaming can result in up to 2.7% of measured machine cycle cost and percentages well into the double-digits for data and instruction cache misses. These numbers will be *directly* affected by obfuscations.

Additionally, languages dynamically compiled during runtime such as Java are, in particular, finicky beasts. The HotSpot$^{\text{TM}}$dynamic compilation functionality is constantly recompiling bytecode into machine code as the benchmarks run. This recompilation can be triggered by the loading of new classes, the exploration of previously unused control flow paths, or for other reasons. This can cause fluctuations in timing and can lead to noisy results.

Precision, however, has been carefully examined. Standard error ($\triangle$) was calculated for all experimental runs. In order to make this meaningful, they were combined in the following manner: where T is the set of eight timings after a given transformation has been applied to a benchmark and O is the set of timings from the benchmark in its original state, and given standard errors $\triangle_T$ and $\triangle_O$[3] the standard error for percentage differences presented in our graphs P (the ratio of transformed benchmark average time to original benchmark average time, or $\frac{T_{avg}}{O_{avg}}$) is $\triangle_P = P\sqrt{(\frac{\triangle_T}{T_{avg}})^2 + (\frac{\triangle_O}{O_{avg}})^2}$.

The largest standard error we saw in our server-mode testing was 2.6% and the majority was well below that. There were 7 degrees of freedom and therefore we used a t-value of 2.37, calculating a 95% confidence interval of $\pm 6.7\%$. Not surprisingly, the interpreter-mode standard error was much lower, at 0.46%, resulting in a 95% confidence interval of $\pm 1.2\%$. Keep this in mind when reviewing the graph data.

Experiments were run separately for every transformation. In each instance the obfuscations were applied in every possible spot. For the `if`-indirection detailed in Section 7.7, for example, each and every `if` instruction in the benchmarks was instrumented with a newly-created trapped `goto` instruction. In this way, these experiments are measuring the true "worst-case" slowdown. After all transformations are detailed and the results are discussed, a combined application of all our obfuscations is derived with respect to performance concerns in Section 8.2.

[3]Standard error for a given set T of eight timings was calculated in the normal way: $\triangle_T = \frac{\sigma_T}{\sqrt{8}}$ where standard deviation for T is $\sigma_T = \sqrt{\frac{1}{8}\sum_{i=1}^{8}(T_i - T_{avg})^2}$.

4.4 The Benchmarks

The benchmarks have been culled 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 the Java source language and compiled with *javac*. The benchmarks represent a wide array of programs each with their own unique coding style, resource usage, and ultimate task. However, none perform heavy input/output - this was required to limit the amount that disk speed issues clouded the results. 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 multi-threaded matrix multiplication algorithm. The third is a multi-threaded 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.

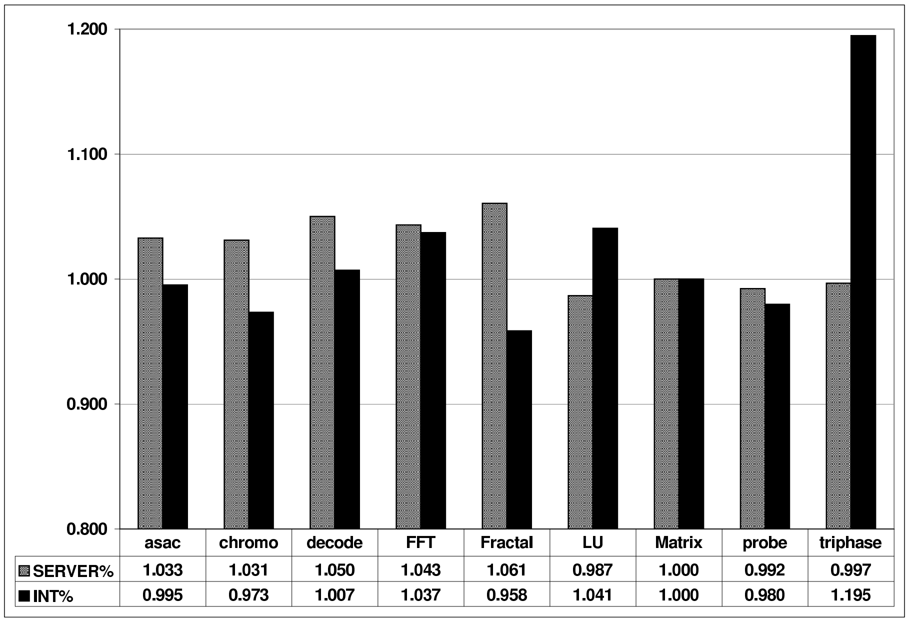
4.4.1 Soot vs. Javac

In order to evaluate the performance effects of each transformation as closely as possible the baseline time measurements were taken from "sootified" benchmark classes. This means their original class files, as produced by `javac`, were run through Soot with the default options enabled. This is necessary because JBCO, regardless of the obfuscating transformations that may be enabled, also runs Soot's default features on the input. Figure 4.1 shows the difference between the original `javac` class files and the "sootified" baseline class files. It's clear that Soot can have an effect on the runtime performance and, in some rare cases, it can be quite glaring (*e.g.*,*Triphase* in interpreted mode shows a ~20% slow down due to a single method but it is unclear what the cause is).

4.4.2 Decompilers

A decompiler is an automatic tool that performs the opposite of a compiler. In this thesis we mean that reverse functionality do be the *translation of executable code back into source code*.

There exists a number of Java decompilers which perform well on bytecode which is specifically produced by Suns Java compiler (`javac`). One of the most popular of these, because of its long history, is `Jad` [Jad]. When given bytecode produced by some version of the `javac` compiler, `Jad` produces excellent output because it is designed to recognize code patterns known to be created by the compiler and it simply recreates the equivalent,



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.033	1.031	1.050	1.043	1.061	0.987	1.000	0.992	0.997
■ INT%	0.995	0.973	1.007	1.037	0.958	1.041	1.000	0.980	1.195

Figure 4.1: *Performance of "Sootified" versus original* `javac` *Benchmarks.*

well-known, source code. If unknown patterns appear in a program, however, `Jad` is almost always unable to fully decompile the method bytecodes in the program. On rare occasions it will crash all together. Because of this it is particularly horrible at decompiling even the most lightly obfuscated software. It is also not able to handle bytecode produced from other sources such as optimizing tools, instrumenters, and third-party compilers generating bytecode from non-Java source languages.

`SourceAgain` [Sou], a more advanced decompiler, also has some tenancies towards code pattern recognition but its website claims it has "insensitivity to the original compiler". It seems to perform some dataflow analysis and definitely performs better than `Jad` in non-Java specific compiler situations.

In contrast to `javac`-specific decompilers, the Soot-based `Dava` [NH06] decompiler was created for the very task of handling arbitrary bytecode. `Dava` does not specifically look for patterns exhibited in `javac` output; it operates on the idea that any valid bytecode should be decompilable. While this requires much more complicated decompilation techniques, it is a more robust approach. The output of `Dava` may not always look as natural but it is more likely to be directly re-compilable and therefore, in some ways, superior.

While older decompilers for Java exist as well, such as Mocha [Moc], we will use only the three that we have outlined here in future chapters in order to give examples of decompiled obfuscations. It will always be noted in the code listings which decompiler was used. Additionally, we will summarize how well each decompiler is able to handle each obfuscation.

Chapter 5
Operator Level Obfuscations

The simplest kind of transformation is one that attempts to conceal information in the same way a magician performs his tricks: by slight of hand. This chapter details transformations which hide information not necessarily by removing it altogether but by putting it in places a reverse-engineer might not think to look.

We call these *operator-level obfuscations* because they do not change the design structure of the program or the control flow of method execution. They simply rework the low-level program logic. These techniques are not built to confuse a decompiler but rather to confuse a human — that is, a reverse-engineer — trying to read the decompiled source. This is the case for the obfuscations in this chapter.

5.1 Renaming Identifiers: Classes, Methods, and Fields (RI$\{$C,M,F$\}$)

The existence of discrete names for program constructs such as classes and methods carries with it a certain vulnerability. Languages compiled to machine code have these identifiers stripped out since they are not essential or even required for the execution of the program. Java maintains the naming scheme even in bytecode in order to be able to report meaningful exceptions for debugging purposes, such as where the exception occurred.

Replacing class, field, and method names in bytecode has the potential to remove a great deal of information. In fact, it is probably one of the most effective obfuscations. Method names, for example, very often carry clear and concise explanations of their function (*e.g.*, `getName`, `getSocketConnection`, *etc.*).

Our renaming transformation changes all class, field, and method names where possible. In order to not disrupt any outside mechanisms such as script files that are used to start the program, the identifier of the Main class (defined as the class with the entry method named `main`) is not changed. Any instances of the `run()` method implemented by a class which extends `java.lang.Thread` likewise are not renamed. Additionally, the program is presumed to *not* use any reflection — a Java framework which allows for loading of classes and calling of method names in a dynamic way.

For those situations where certain constructs cannot or should not be renamed (*e.g.*, in the face of reflection) they can be specified as command line options either explicitly or with regular expressions and those matching constructs will not be renamed (see Section 8.1).

Those identifiers that are renamed are done so with randomly generated sequences or by "stealing" names from other methods or fields within the program.

JBCO uses three sets of "ugly characters" as dictionaries from which to form the randomly generated new names — sets chosen specifically for their hard to read nature, where each character is similar to the others and therefore hard to distinguish among the code. These three sets include:

S, 5, $ — the uppercase letter S, the digit five, and the dollar sign; three characters that are similar in visual appearance.

l, 1, I — the lowercase letter L, the digit one, and the uppercase letter i. These three characters can be very hard to tell apart, depending on the choice of font.

_ — the underscore. This is a useful single-character set because it is particularly hard to distinguish between a variable named with five underscores versus a variable named with six underscores, especially when the font used to display the code renders consecutive underscores as a single line, as is often the case.

The algorithm creates new names randomly. If, after ten attempts, it does not find a name which is unique of the given length it increase the length by one and resets, attempting another ten tries. This repeats until the algorithm is successful.

Stolen names are sometimes used instead in order to confuse the reverse-engineer. Using a method name of "createFile" for a method that deletes files, for example, could be very confusing.

Listing 5.1 shows a method in one of the benchmarks after renaming has been performed. While the actual operations may be obvious and library method names and fields could not be renamed, it is much harder to follow the semantics due to the difficulty in clearly identifying the different objects, variables, and methods from one another. Note that `Dava` has chosen names for local variables, in the absence of a local variable table in the class file, that help clarify their type.

5.1.1 Performance Results (RI{C,M,F})

The renaming of identifiers in Java will, in most cases, result in shorter average identifier lengths. Human programmers tend to give meaningful names to methods and fields. These human-created names will, on average, be longer than the short names generated by this transformation. Because of this, we should not expect to see any slowdown whatsoever due to this obfuscation. In fact, if a lower average is obtained, there could be room for an improvement in runtime performance. Indeed, this is what we observe in the server mode for almost all benchmarks, as seen in Figure 5.1. Note that not all benchmarks are represented in this graph because not all of them had a significant number of classes, methods, and fields to rename.

The reason for the observed slowdowns in interpreted mode for all benchmarks is unclear but well within the range of standard error. A possible explanation could be slow method name resolution due to key collisions in a hashtable — the renaming transformation does use names which are often similar and have a high likelihood of sharing a prefix. Nevertheless, without a detailed study of the VM implementation this is pure conjecture.

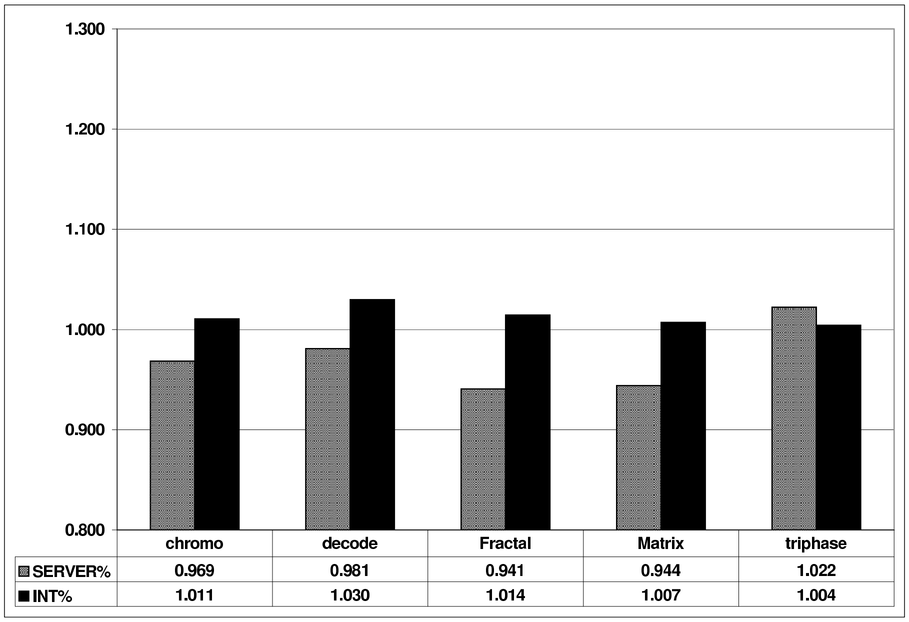
```
void ___(Graphics r1, float f0, float f1, float f2, float f3, float f4, int i0) {
  lII = f0;
  lll = f1;
  lIl = f3;
  lIII();
  float f6 = lII;
  float f5 = lll;
  int i3 = i0 + −1;

  if (i3 >= 3) {
    r1.setColor(Color.white);
    ___(r1, f2, (int)f0, ll1 − (int)f1, (int)f6, ll1 − (int)f5);
  } else {
    r1.setColor(Color.green);
    ___(r1, f2, (int)f0, ll1 − (int)f1, (int)f6, ll1 − (int)f5);
  }

  if (i3 > 0) {
    $5$ = this.____(f0, f1, f6, f5);
    l11I(Ill1);
    _____(r1, lII, lll, $5S ∗ f2, Il1 ∗ f3, Ill1, i3);
    $5$ = ____(f0, f1, f6, f5);
    l11I((− (SSS)));
    _____(r1, f6, f5, $5S ∗ f2, Il1 ∗ f3, SSS, i3);
  }
}
```

Listing 5.1: *Example method after renaming of classes and fields and methods. Decompiled by* `Dava`.

5.2 Embedding Constant Values as Fields (ECVF)

Within the Java source language constant pieces of data — that is, values that are hard coded into the program — are sometimes "in place". They appear directly within the code where they are used. Because of this it is quite clear when and where this data is being used. Consider Listing 5.2 where the use of the constant error string "Illegal Depth Number! …" is embedded directly in the code, and therefore its use is clear.

While the values appear directly within the source code, they are actually stored separately in bytecode. Each class file in a program has constant data used by that class file



	chromo	decode	Fractal	Matrix	triphase
▨ SERVER%	0.969	0.981	0.941	0.944	1.022
■ INT%	1.011	1.030	1.014	1.007	1.004

Figure 5.1: *Effects of renaming class, field, and method identifiers.*

stored in a constant pool, separate from the method codes. These constants are accessed by their index within the pool through the `ldc` instruction. These values are the truly embedded constants. Other values, such as class fields which are not final but might nevertheless maintain a constant value throughout the life of a program (which are assigned a value in the class initializer) are *not* considered embedded because they are accessed through the field and not directly by index. While these field uses may also increase the readability of code due to their meaningful names, such as `INTEGERMIN` or `MAXBUFSIZE`, they can not be obfuscated with this approach. See Section 5.1 for a possible technique in clouding the meaning of these fields.

The approach of our obfuscation of constant data is to conceal these values in some way so that the actual data does not appear directly where it is used in the program source if it is decompiled. A very simple and cheap way to do this is store the constants in static class fields. In this case, all instances of the constants within the code can be replaced by references to the static fields by a quick sweep through all method code.

In its basic form this technique is not very resilient. Inter-procedural copy propagation

```java
try {
  level = Integer.parseInt(arg);
} catch (NumberFormatException exc) {
  System.out.println("Illegal Depth Number! (try -help)\n" +
    "could not parse: " + arg);
  System.exit(1);
}
```

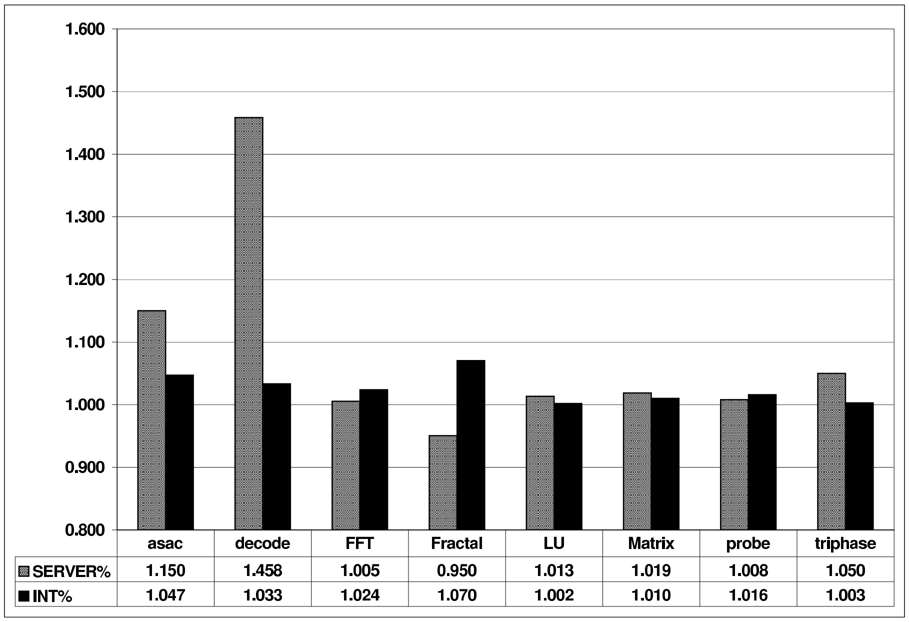
Listing 5.2: *A Java source code snipper clearly showing embedded constant data.*

could quickly reverse the obfuscation. Nevertheless, an opaque predicate and dead re-assignment code for each constant could easily subjugate this shortcoming.

5.2.1 Performance Results (ECVF)

The benefit of constant data appearing within source code is obvious. The programmer is not required to look up the value of a constant field or otherwise know anything about the design of a program. An integer variable within the program, for example, which is multiplied by the constant integer two is simply that (`x * 2`). Certainly, some data should and will be assigned to static final constant fields by the programmer but many simple uses of primitive values in the program do not warrant creating an entire new field. This transformation removes a level of accessibility for those pieces of data left in code of the program and makes it more difficult to understand the intent of the source code when decompiled.

Not surprisingly, negative effects caused by this transformation are limited to those programs which rely heavily on constant data (see Figure 5.2). Both `asac` and `decode` see significant slowdowns in server mode, whereas all other benchmarks do not show much change. This makes sense; `decode` performs the highest number of field accesses per second of running time — an order of magnitude greater than `asac`, which is itself an order of magnitude greater than the next highest `triphase`. It is likely that interpreted mode is so slow in general that the added costs of the field accesses are not noticeable.



	asac	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.150	1.458	1.005	0.950	1.013	1.019	1.008	1.050
■ INT%	1.047	1.033	1.024	1.070	1.002	1.010	1.016	1.003

Figure 5.2: *Effects of embedding constant program data into static fields.*

5.3 Packing Local Variables into Bitfields (PLVB)

The idea of data obfuscation is not a new one. Collberg and Thomborson [CT02] suggested that a boolean variable could be split into two integers i_1 and i_2. The boolean value could then be resolved by performing an exclusive or on the integers. Through this logic, the boolean value itself is hidden.

Arrays, Strings, and other data structures are also available for data obfuscation. However, there will always be a penalty incurred by such transformations when the data is reconstituted. Additionally, since it is fully-formed at some point in the execution of the program it is therefore vulnerable to reverse-engineering attacks through the use of debuggers. The overall usefulness of these approaches as a general obfuscation scheme is therefore minimal.

Nevertheless, we have attempted to reach a compromise. This transformation coalesces local primitive variables into a fewer number of bit fields. In reality, these bit fields are Java primitive long types — 64-bit data blocks — which are used to represent multiple booleans, integers, characters, or bytes.

The simplest approach would be to collect all the primitive locals of a method and calculate the number of bits required to represent them all. However, to further obfuscate things, we choose random bit ranges within the long primitives for packing. For example, an integer primitive normally represented by 32 bits might get packed into a long between its 9th and 43rd bits, exclusively.

Every `load` and `store` instruction of the packed locals in the original method are replaced with loads and stores of the long primitive they are packed into, accompanied by the proper bit-shifting and bit-masking operations in order to isolate and restore the value of the local. Listings 5.3 and 5.4 present a method before and after local packing.

```
static void FillPowerMatrix(Digit matrix[][], Digit x[]) {
  int n = matrix[0].length;

  for (int i = 0; i < n; i++) {
    matrix[i][0] = new Digit(1);

    for (int j = 1; j < n; j++) {
      matrix[i][j] = matrix[i][j−1].mult(x[i]);
    }
  }
}
```

Listing 5.3: *Example `FillPowerMatrix` method before local variables are packed into bitfields.*

Because this transformation will affect performance due to increased value manipulation and because the placement of locals within the long primitives is random, not all locals will be packed.

5.3.1 Performance Results (PLVB)

Any encapsulation of data should be expected to slow performance since any access to the data must entail some unpacking and storing must entail some packing. In the case of locals packing as we have described, this is mostly true (see Figure 5.3). In some benchmarks the effect can be quite crippling, as in `asac` and `triphase`. Strangely, a third

```
static void FillPowerMatrix(Digit[][] r0, Digit[] r1) {
  long l0;
  int i2, i3;

  for (i2 = r0[0].length, i3 = 0; i3 < i2; i3++) {
    r0[i3][0] = new Digit(1);

    for (l0 = (long) 1 & 4294967295L ^ l0 & −4294967296L;
           (int) (l0 & 4294967295L) < i2;
             l0 = (long) (1 + (int) (l0 & 4294967295L)) ^ l0 & −4294967296L) {
      r0[i3][(int) (l0 & 4294967295L)] =
                  r0[i3][(int) (l0 & 4294967295L) − 1].mult(r1[i3]);
    }
  }
}
```

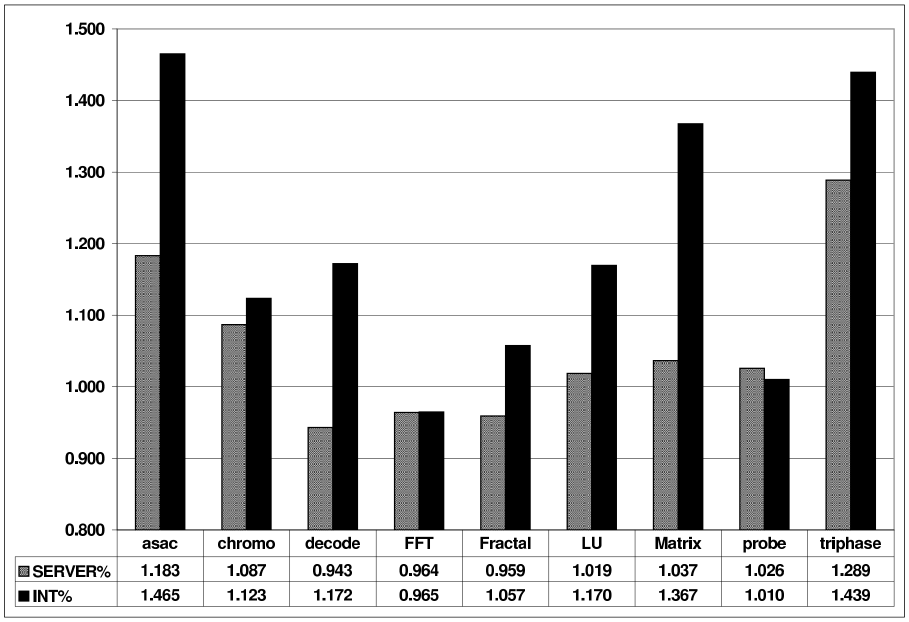
Listing 5.4: *Example `FillPowerMatrix` method after local variables are packed into bitfields.*

of the benchmarks actually show improvements in server mode. This is possibly due to a reduction of register usage in certain methods, limiting the amount of memory swapping that is required. One method with 20 local booleans, for example, could have all of those variables packed into one long by this transformation. This would result in the use of two 32-bit registers instead of 20.

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

There are often more ways than one to express a calculation and to the human eye, some are more complex than others. In fact, optimizing compilers sometimes convert a complex operation such as multiplication or division, which may be more readable, into a sequence of cheaper operations, which are more confusing. With this arithmetic transformation, the goal is to use the same trick as the optimization mentioned above by obscuring multiplications and divisions into less obvious bit-shifting instructions.

In particular, we look for instances of expressions in the form of $v*C$ (a similar tech-



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.183	1.087	0.943	0.964	0.959	1.019	1.037	1.026	1.289
■ INT%	1.465	1.123	1.172	0.965	1.057	1.170	1.367	1.010	1.439

Figure 5.3: *Effects of packing local data into bitfields.*

nique 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.

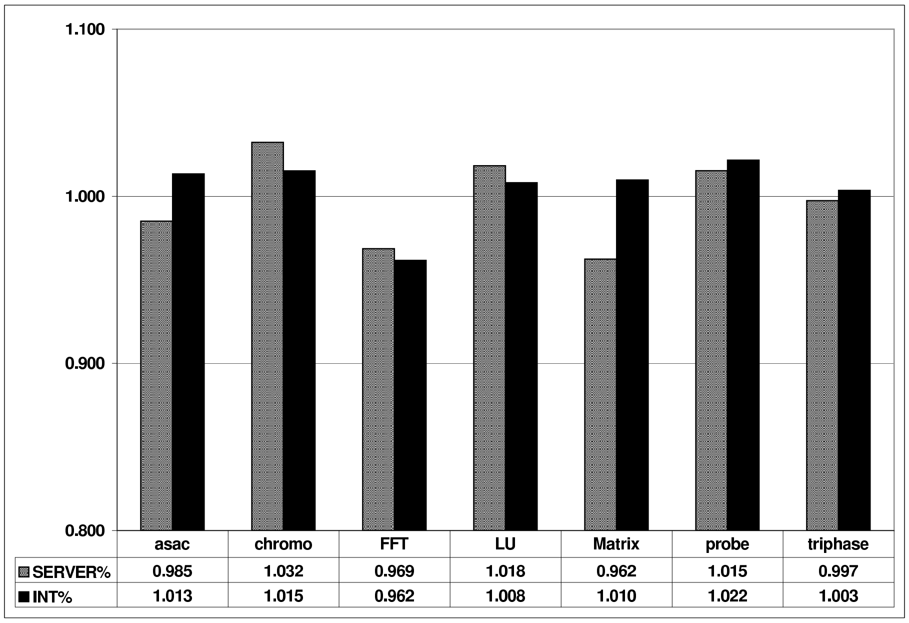
In order to further obfuscate the computation we don't use the shift value s directly, but rather find an equivalent value s'. To do this we take advantage of the fact that shifting a 32-bit word by 32 (or a multiple of 32) always returns it to its original state. Thus we choose a random multiple m, 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 hiding the constants with opaque predicates could hamper decompilation.

5.4.1 Performance Results (CAE2BO)

This transformation is increasing the overall amount of calculations a program must perform. Nevertheless, it does not seem to make a performance footprint at all. The largest variations from the original program, shown in Figure 5.4, were the `FFT` and `Matrix` benchmarks but they were *faster*, not slower. It would appear that the VM, or modern hardware in general, is particularly good at bit-shifting operations compared to multiplication, which explains why this approach is also used by optimizing compilers. The rest of the numbers are well within the standard error range.



	asac	chromo	FFT	LU	Matrix	probe	triphase
▨ SERVER%	0.985	1.032	0.969	1.018	0.962	1.015	0.997
■ INT%	1.013	1.015	0.962	1.008	1.010	1.022	1.003

Figure 5.4: *Effects of transforming arithmetic expressions into bit-shifting operations.*

Chapter 6
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 in this chapter to include two facets: the high-level object-oriented design and the low-level control flow.

Each transformation presented in this chapter operates on either the high-level design (moving methods, creating new classes) or on the low-level design (confusing the control flow of a method). A round house built to appear square can be very confusing indeed.

6.1 Adding Dead-Code Switch Statements (ADSS)

The switch construct in Java bytecode offers a compelling and useful control flow obfuscating tool. A switch with a few targets is the only organic way, other than traps, to manufacture a control flow graph which has a node whose successor count is greater than two.

The goal of this transformation is to add additional edges to the control flow graph in order to severely complicate flow analysis. To do this, a stack height calculator determines those units in the graph which have a stack height of zero. All of these points are now viable destinations for a `tableswitch`. Using opaque predicates to jump around the `tableswitch`, the switch is instrumented with as many as ten possible cases. Each case is randomly chosen out of the list of zero-height units.

The end result is an additional incoming edge for every unit chosen as a switch destination and an additional outgoing edge from the `tableswitch` for each case. The only additional nodes are the `tableswitch` itself and the opaque predicate `if` statement. This, effectively, increases connectedness of the graph and increases overall complexity.

The result of this transformation is usually quite messy as can be seen when comparing original code (Listing 6.1) against decompiler output from obfuscated code (see Listing 6.2). There is no easy way to properly handle the control flow other than to heavily nest the switch statement within labelled blocks, allowing the different cases to jump outward to their proper destinations. While it is commendable that `SourceAgain` manages to re-create a close approximation of what the bytecode is doing, it is still not recompilable and fairly unreadable. `Dava` is also unable to reproduce proper source code and in some cases crashes while trying to resolve cycles in the control flow graph.

```
if (writeImage != null) {
  try {
    File file = new File("out");
    ImageIO.write(writeImage, "png", file);
  } catch (Exception e) {
    System.exit(1);
  }
}
System.exit(0);
```

Listing 6.1: *A Java source code snippet* before *obfuscation with a dead-code switch.*

6.1.1 Performance Results (ADSS)

The idea of adding switch statements into a method is quite useful. Even relying on randomness to choose the branch targets is usually enough to increase the complexity of the control glow graph. This results in very unnatural looking decompiled code or, often times, completely un-decompilable code. This, in theory, is a fairly cheap obfuscation as well since the switch itself is never executed — it is essentially dead code. However, there is the cost of the opaque predicate. This extra branching logic needs to be inserted to skip over

```
if(obj != null) {
  try {
    ImageIO.write((RenderedImage) obj, "png", new File("out"));
  } catch(Exception exception2) {
    ++i;
    obj = exception2;
    i += 2;
    System.exit(1);
  }
}

label_167: {
  while(lI1.booleanValue() == ___) {
    switch (i) {
      default:
        break;
      case 3:
        break label_167;
      case 1:
        ++i;
        obj = exception2;
        i += 2;
        System.exit(1);
        continue;
      case 2:
        i += 2;
        System.exit(1);
        continue;
    }
  }
  System.exit(0);
}
```

Listing 6.2: *A Java source code snippet* after *obfuscation with a dead-code switch. Decompiled by* `SourceAgain` *— it is not correct source code.*

the switch.

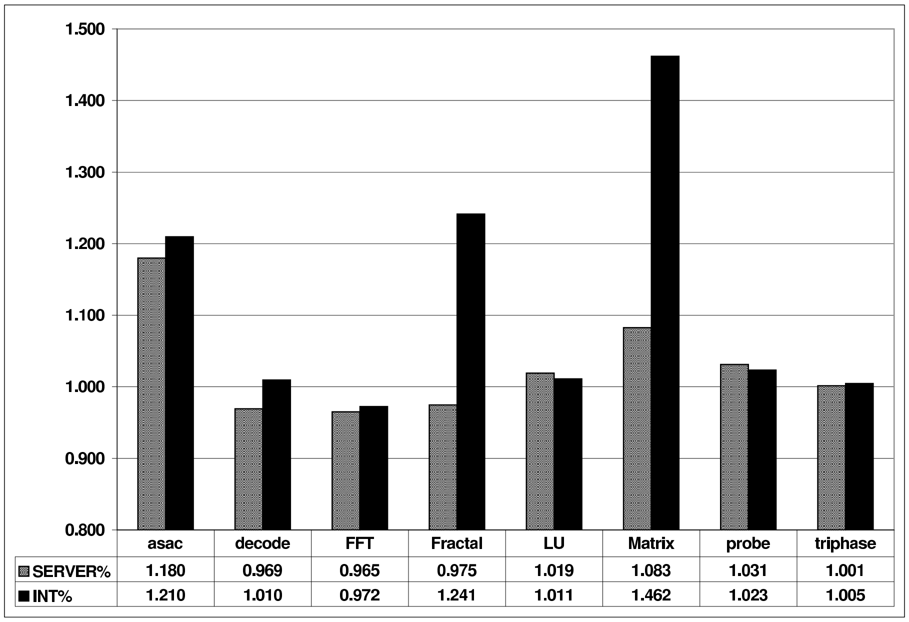
In general, the obfuscation does not hinder performance. In fact, a number of the benchmarks show improvements in server mode, as can be seen in Figure 6.1. `asac` is particularly worse off in both modes but this is due to the fact that one particular method in the program, a method called to swap two numbers in the sorting algorithm, is called over 225 million times in a single five-second run of the program. This method was instrumented with a switch by the transformation and the opaque predicate installed results in the addition of one method call (booleenValue()), one field access, and one `if` instruction. A profile of the running program shows a sharp increase in the amount of time spent in swap versus any other method. This illustrates the usefulness of selective obfuscation, which will be discussed later in a later chapter.

A similar situation occurred in `Matrix`, only worse. An even smaller method (used to set elements in a matrix) was augmented with a switch statement that had six branch targets; highly unusual for a method of such a small size. Normally the transformation would not find so many target candidates in bytecode that small. In this situation, the method jumped from 14 bytes to 97 bytes long and a number of "junk" additions were made to the ultimately unread switch control local. These random increments are intended to make it look as though the switch is actually used in a normal way. Luckily, the VM JIT compiler is able to mostly optimize away this performance issue although the interpreter is not nearly as lucky — the method in question accounted for 33% of the program runtime in interpreted versus 3% in server mode.

6.2 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 duplicates and replacing them with a single switched instance we can potentially reduce the size of the method and can definitely confuse the control flow.

A complex analysis could be written to handle many duplicates of a sequence, including those that straddle try block boundaries. However, this would require much more complicated analysis as well as the addition of more control flow logic. We chose a simple



	asac	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.180	0.969	0.965	0.975	1.019	1.083	1.031	1.001
■ INT%	1.210	1.010	0.972	1.241	1.011	1.462	1.023	1.005

Figure 6.1: *Effects of adding dead-code switch instructions.*

conservative approach instead. Every sequence of 20 instructions or less within a method is collected and then the method is checked for duplicates of these sequences. There are a number of rules which define whether a sequence is a proper duplicate or not.

- The sequence must be of the same length as the original and any bytecode instruction in the duplicate sequence b_d at index i must equal the original bytecode b_o at index i.
- The duplicate instructions and their parameters must match the original sequence - that is to say an istore is only a proper duplicate instruction if it shares the same register parameter with the original istore and an if instruction is only a proper duplicate if it shares the same jump destination as the original.
- The duplicate instruction b_d at index i must be protected by exactly the same try blocks as the original instruction b_o at index i. If the original is not protected at all, neither can the duplicate be protected. This simplifies the analysis a great deal.
- Every instruction in a sequence other than the first must have no predecessor instructions that fall *outside* the sequence. This prevents the analysis from having to

verify that the original and duplicate sequences share the same predecessors, further simplifying the problem.

- Each instruction in the duplicate sequence b_d at index i must share the same stack as its counterpart in the original sequence b_o at index i. This ensures that the verifier will not complain and that each instruction in the method will always have the same initial stack height with the same initial stack type ordering.

- No instruction within the duplicate sequence can overlap with the instructions in the original sequence as per their layout within the method.

The algorithm performs rounds, starting with the longest sequences first. Thus, in the first round the algorithm collects candidate sequences of length 20 and then attempts to find duplicates of those candidate sequences. If any are found, the algorithm then creates a new method local of integer type which will act as a control flow flag. The duplicate sequences are removed and replaced with an initialization of the control flow flag to a unique number (usually 1, 2, 3, and so on, as the duplicate sequences are removed) followed by a `goto` instruction directed at the first instruction in the original sequence. When all duplicates have been removed, the original sequence is prepended with two instructions which store 0 to the control flow flag and appended with a load of the control flow flag followed by a `tableswitch` instruction. The default action is to fall through to the next instruction (the original successor of the original sequence). A case for each duplicate sequence is added to the table which results in a jump to the original successor instruction for that sequence.

After the first round, the sequence length is decremented and the second round is run. This round of candidate selection and duplicate detection is performed on the *new* version of the method which might have been changed by the previous round. This is repeated until the sequence length is less than 3. No doubt, sequences of length two are heavily repeated throughout Java bytecode and replacing these would only increase the overall method size and decrease the overall performance. For each duplicate sequence there are three instructions added: the unique integer constant push onto the stack, the store of that constant to the control flow flag, and the `goto` instruction which jumps to the top of the original sequence. For each original sequence, a `push/store` instruction pair is prepended to it and a `tableswitch` is added to the end of it. The `tableswitch` size, in the class file, is dependant on the number of cases in the instruction. The more duplicate sequences, the larger the space required to store it. Overall, this is three extra instructions for each duplicate, plus one extra case in the tableswitch and two extra instructions for each original sequence plus one `tableswitch` instruction: 3 + 4*duplicates. Clearly, in order for this transformation to reduce the class size, the sequence length has to be at least seven if there is a single duplicate, at least 6 if there are two duplicates, at least 5 for three, *etc.* An example can be seen in listings 6.3 and 6.4, which show a method's code before and after obfuscation, respectively.

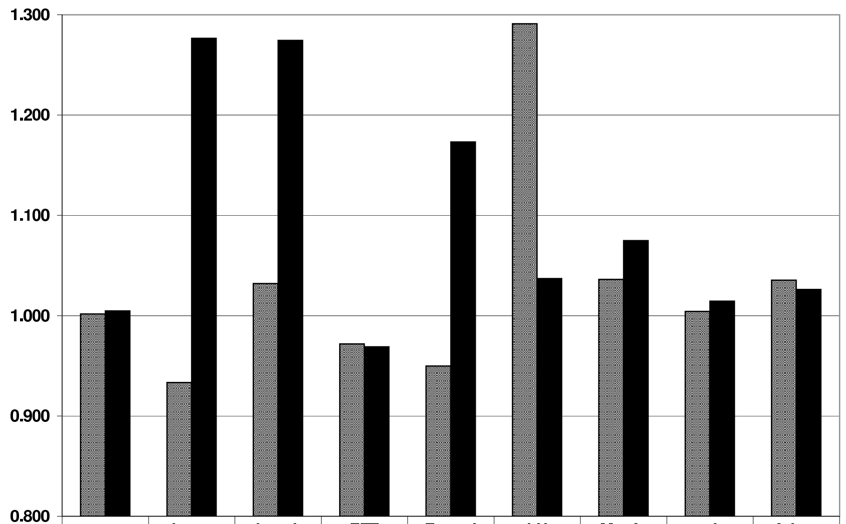
```
protected static void bitreverse(double data[]) {
  int n=data.length/2;
  int nm1 = n−1;
  int i=0;
  int j=0;

  for (; i < nm1; i++) {
    int ii = i << 1;
    int jj = j << 1;
    int k = n >> 1;

    if (i < j) {
      double tmp_real = data[ii];
      double tmp_imag = data[ii+1];
      data[ii] = data[jj];
      data[ii+1] = data[jj+1];
      data[jj] = tmp_real;
      data[jj+1] = tmp_imag;
    }
    while (k <= j) {
      j −= k;
      k >>= 1;
    }

    j += k;
  }
}
```

Listing 6.3: *Example method* `bitreverse` before *duplicate sequences have been resolved into one.*

Empirical testing has found that it is possible to occasionally find duplicate sequences of high magnitude. Lengths of 17 and 19 have been seen and almost all benchmarks tested had some duplicates of length 8 or more. Nevertheless, the majority of duplicates found are in the 3 to 5 length range. While these are not productive in reducing the class file sizes they are very useful in confusing control flow. Additionally, performance testing finds that in many cases this transformation can *decrease* the running times, suggesting this is a possible area of study for compiler optimization.

6.2.1 Performance Results (RDS)

The effects of duplicate sequence reuse can be fairly significant as seen in Figure 6.2. In some cases performance times are improved in server mode, yet a number of benchmarks show slowdowns in interpreted mode. This is likely due to positive JIT optimizations or improved instruction caching. The transformation is usually enlarging methods and effecting the amount of control flow overhead so it is unsurprising to see measurable slowdowns in interpreted mode.



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.002	0.933	1.032	0.972	0.950	1.291	1.036	1.004	1.035
■ INT%	1.005	1.277	1.274	0.969	1.173	1.037	1.075	1.014	1.026

Figure 6.2: *Effects of duplicate sequence reuse.*

```
protected static void bitreverse(double[] r0) {
  // variable declarations and initializers removed.
  label_2: while (i2 < i1) {
             i4 = i2 << 1;
             i5 = i3 << 1;
             $i9 = i0;
             i6 = 0;
             while (true) {
               i7 = $i9 >> 1;
               label_1: switch (i6) {
                          case 1: break label_1;
                          default: if (i2 >= i3) break label_1;
                            $d1 = r0[i4];
                            i11 = 0;
                            label_0: while (true) {
                                       $i12 = i4 + 1;
                                       switch (i11) {
                                         case 1:
                                           r0[$i12] = r0[i5 + 1];
                                           r0[i5] = $d1;
                                           r0[i5 + 1] = d0;
                                           break label_1;
                                         default:
                                           d0 = r0[$i12];
                                           r0[i4] = r0[i5];
                                           i11 = 1;
                                           continue label_0;
                                       } // end switch (i11)
                            } // end label_0 block and while loop
               } // end label_1 block and switch (i6)
               if (i7 > i3) {
                 i3 = i3 + i7;
                 i2++;
                 continue label_2;
               }
               i3 = i3 − i7;
               $i9 = i7;
               i6 = 1;
             }
  } // end label_2 block and while loop
}
```

Listing 6.4: *Example method* `bitreverse` after *duplicate sequences have been resolved into one. Decompiled by* `Dava`*. The code is semantically equivalent but much more difficult to read. Local variable declarations were removed for space considerations.*

The `LU` benchmark is the only one to show a marked slowdown in server mode — almost 30%. The slowdown is almost entirely encapsulated in a single method, `factor`, which is the same method that is isolated in Section 7.2.1 as the cause of a JIT slowdown (which is likely due to its complex nested structure). In this situation, the JIT is simply unable to significantly optimize the method.

Indeed, individual profiling of the `factor` method shows a ∼26% slowdown after transformation. Since this method accounts for well over 90% of the programs normal runtime, it explains the performance degradation.[1]

6.3 Replacing if Instructions with Try-Catch Blocks (RIITCB)

The try-catch construct in the Java language can be considered implicit potential control flow. In almost all normal uses of a try block, there is a possibility of an exception occurring but it is both not intended and hopefully does not happen. Proper programming style would decree that code should do its best to *prevent* exceptions even if it does handle them.

Nevertheless exception throwing can be explicit. The `throw` instruction allows this or, alternately, one can write code that is guaranteed to cause an exception. By catching these exceptions one can create a new (though unintended by the language designers) use of the try-catch — normal program control branching.

This transformation exploits two well known facts of Java. First, invoking an instance method on a `null` object will always result in a NullPointerException being thrown. Second, two `null` objects will always be considered equal by the `ifacmpeq` instruction and a non-null object will always be considered *not equal* to a `null` object.

Each `ifnull` and `ifnonnull` instruction in a method is considered for this transformation. One of two different scenarios are applied in order to convert this single instruction into a more obfuscated try-catch block such that the original if-branching is performed instead by the exception catching mechanism. This is very similar to the approach by Sakabe, *et al.* [SSM03], although they relied on creating special exception objects which added un-

[1]This is an example where profiling can help to fine tune obfuscation settings described in 8.1.

necessary overhead. This transformation, in throwing exceptions, takes advantage of Java's ability to throw `null` exceptions. Normally, when an exception is thrown an instance of an Exception object must be created and its reference must be pushed onto the stack, causing some performance overhead. Throwing a `null` exception side-steps this object instantiation but still allows for the exceptional control flow associated with the try-catch.

Every object in Java is a child of, or is itself, a `java.lang.Object`. This ensures that certain methods defined in that class will always be invokable, no matter the actual class of an object. Whether those methods will virtually resolve to `java.lang.Object` or some other overloaded version is not important.

The `ifnull` instruction being transformed is removed and replaced with a call to `toString` or a `ifacmpeq` instruction comparing the original object to a null reference. Listings 6.6 and 6.7 show decompiled examples for `SourceAgain` and `Dava`, respectively. The original source code is shown in Listing 6.5.

```java
public static void main(String argv[]) {
  if (argv == null) {
    System.out.println("Something is really wrong");
    return;
  }

  for (int i = 0; i < argv.length; i++) {
    if (argv[i] != null)
      System.out.println(argv[i]);
  }
}
```

Listing 6.5: *Example Java source method* `main` before `ifnull` *instructions are transformed into a try-catch blocks.*

In the case of the `toString` replacement, the method call is wrapped in a Try block which handles NullPointerExceptions and if thrown, redirects control flow to the original `if` instructions target. If the object is indeed null it will be unable to resolve the method call and, of course, a NullPointerException will be raised. The original target is prepended with a pop in order to discard the exception that is pushed on the stack by the exception handling mechanism.

```
public static void main(String as[]) {
  int i;

  try {
    as.equals(null);
  } catch(NullPointerException unused2) {
    System.out.println("Something is really wrong");
    return;
  }

  for(i = 0; i < as.length; ++i) {
    try {
      as[i].toString();
    } catch(NullPointerException unused3) { }
    // location of "not null" branch of the code
  }
}
```

Listing 6.6: *Example Java source method* `main` after `ifnull` *instructions are transformed into a trapped* `toString`/`equals` *calls. Decompiled by* `SourceAgain` *— it is missing the code following the second* `if` *and is therefore incorrect code.*

If the `ifcmpeq` option is chosen, a null object is pushed onto the stack in order to compare it with the original object on the stack. If they are equal, control flow jumps to an explicit `throw null` instruction, which is trapped by a try block. This exceptional control flow behaves as the previous example does, popping the null exception off the stack and proceeding to the original target.

6.3.1 Performance Results (RIITCB)

Unfortunately, the benchmark suite selected for use in this thesis had very few instances of `ifnull` and `ifnonnull` instructions. Only two benchmarks were modified by this transformation and the effect was very minimal. In both cases the specific code that was changed was not “hot” - that is, it was not a heavily executed area within the program. Therefore the true performance penalties incurred by this transformation are not known. However, it is likely that there would be a significant slowdown if “hot” code was affected due to the addition of an exception table lookup. Indeed, the research by Sakabe, *et*

```
public static void main(String $no[]) {
  Object $n0 = null;
  int i0;
  String $r4;

  try {
    r0.equals($n0);
  } catch (NullPointerException $r2) {
    System.out.println("Something is really wrong");
    return;
  }

  for (i0 = 0; i0 < r0.length; i0++) {
    $r4 = r0[i0];
    label_0: {
      try {
        $r4.toString();
      } catch (NullPointerException $r8) {
        break label_0;
      }

      System.out.println(r0[i0]);
    } //end label_0:
  }
}
```

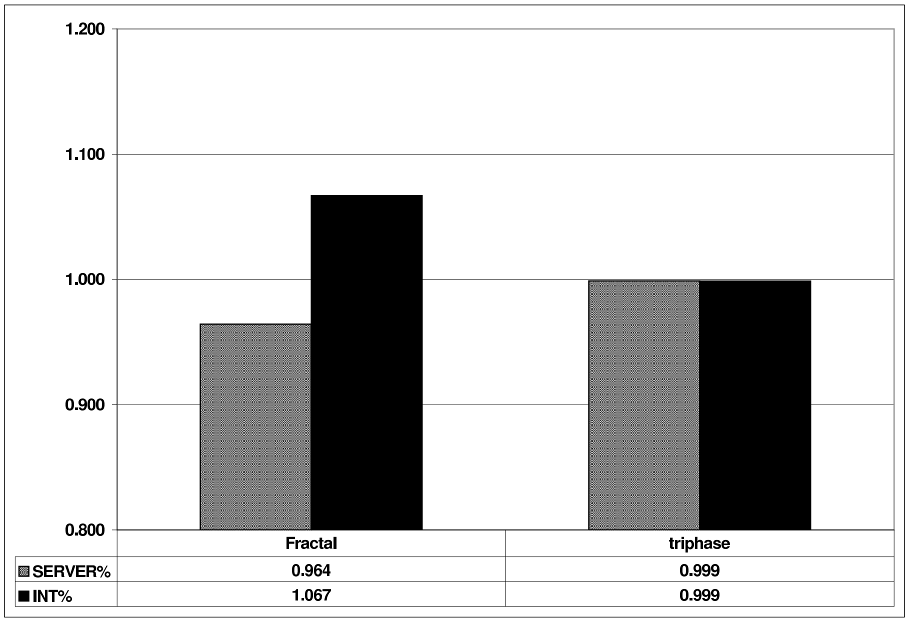
Listing 6.7: *Example Java source method* `main` after `ifnull` *instructions are transformed into a trapped* `toString`/`equals` *calls. Decompiled by* `Dava` *— it is correct code.*

al. [SSM03] suggests this to be true.

6.4 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, while not exposed themselves, 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 standard Java libraries — the design of which is known as an Application Programming Interface (API) — cannot be renamed



	Fractal	triphase
▨ SERVER%	0.964	0.999
■ INT%	1.067	0.999

Figure 6.3: *Effects of replacing* `if(non)null` *instructions with Try-Catch constructs.*

because the obfuscator should not change library code[2]. Therefore, the next best option is to hide the names of the library methods. The approach we take in this transformation is by indirecting all library method calls through intermediate methods with nonsensical identifiers.

The code for 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 then 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. If two program classes, `A` and `B` do not reference each other or the fields of the other then they are independent. If *A* invokes a library call somewhere within its method code and that call is obfuscated by this transformation through the addition of a new method *M* in class *B*, then *A* and *B* become interdependent, or "coagulated" (see Figure 6.4). Therefore, this obfuscation is two-fold. It confuses the overall design of the program and also hides

[2]While it is not completely impossible, it is not reasonable. Firstly, obfuscating library code is sometimes illegal. Second, it would mean that those libraries would have to be distributed with the program, as well, causing an astronomical blowup in the program's distribution size

the library method calls by moving them to a completely different “physical” part of the program.

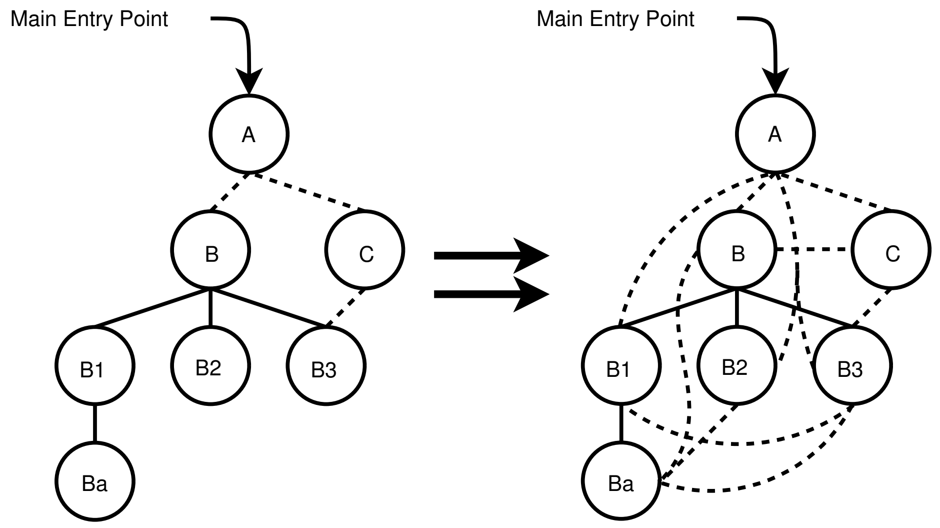


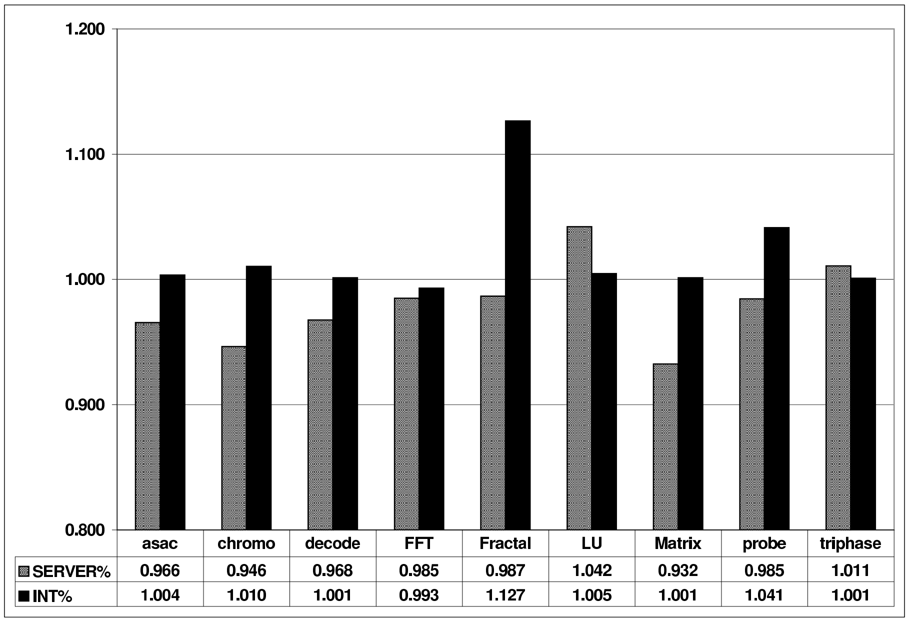
Figure 6.4: *Effects of “class-coagulation” on program structure. Solid lines represent parent-child class links, dotted lines represent other class dependencies.*

6.4.1 Performance Results (BAPIBM)

While this transformation has a potential to increase the overall size of a program it should not affect performance very much. Certainly, a level of indirection is being added, incurring an extra method invocation for every library method call. However, in most cases the JIT will inline these methods since they are very short. If any noticeable performance slowdowns occur they should be expected to manifest themselves much more sharply in interpreted mode.

In Figure 6.5, Fractal shows this interpreted slowdown, but it is also the heaviest benchmark in terms of dynamic library API usage. In total it makes over 82 million of these (now indirected) calls, or roughly 5.8 million per second. `LU` is a distant second at 2.8 million calls per second and `probe` is third at 1.3 million.

As to the slowdown of LU in server mode, it appears to be due to the fact that the benchmark has an extremely short running time - roughly 1 second - and there the JIT does



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	0.966	0.946	0.968	0.985	0.987	1.042	0.932	0.985	1.011
■ INT%	1.004	1.010	1.001	0.993	1.127	1.005	1.001	1.041	1.001

Figure 6.5: *Effects of adding library API buffer methods.*

not have time to fully inline the indirected library calls. Using the “-Xcomp” flag with the VM to force full compilation of all methods results in almost identical running times between the original and obfuscated versions.

Many of the obfuscated benchmarks were faster in server mode. This is possibly due to the heuristics used for inlining in the JIT. The small single-statement buffer methods might be flagged by the JIT for more aggressive inlining.

6.5 Building Library Buffer Classes (BLBC)

In addition to library method calls, having a class that extends a library class (as opposed to another application class) can lend a certain amount of clarity to a program. Parent class methods that are overridden in the child are more obvious as well. Take, for example, a class called `IDObject` which extends `java.lang.Object` and adds an integer instance field called `ID`. Let’s say the `toString` method is overridden and made to prepend the `ID` to the object’s string representation. Any normal Java programmer will understand very quickly the overall purpose of the class: to have an Object which is marked with an ID that is always included in print outs.

In order to cloud this particular design structure of Java we have implemented a transformation that creates buffer classes between any application classes that extend a library class and its parent library class. The purpose should be clear. In the above example, after obfuscation, the `IDObject` class would extend a newly created buffer class which we will call `IDBuffer`. This parent class would implement its own nonsense `toString` method. Coupled with the obfuscation from the previous section that buffers library method calls, 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 that a reverse-engineer must look at.

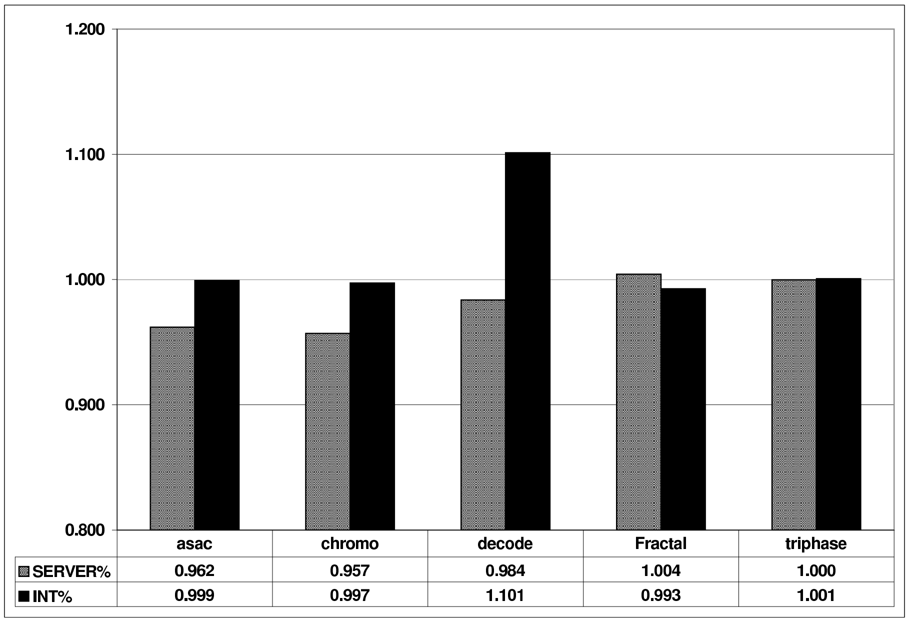
6.5.1 Performance Results (BLBC)

The object-oriented nature of Java allows for some higher-level abstraction and this can be used to further obfuscate code. Because the framework of a program is human-designed, it will most likely hold a certain amount of information about the software's general purpose, yet it is rare that this framework is necessary to derive a solid and effective compiled program. This transformation is intended to fleece a certain amount of object-oriented design by moving overridden methods into an intermediate class which acts as a buffer between the library and application levels. Large applications could see a significant number of classes created by this transformation but, in general, it is an order of magnitude smaller than the number of classes in the application as a whole.

There will be some performance overhead incurred by this transformation since there will be more classes and more method calls which will result in multiple virtual lookups but overall, little slowdown should be expected.

The only notable performance slowdown in the benchmarks is with the `decode` benchmark. This slowdown can be attributed to the extra work necessary to call the intermediate `S$5$5`'s constructor - the added level of object abstraction created by the transform between the `Digit` class and the `java.lang.Object` class. `decode` is a decryption program and creates many of these objects - over 30 million in total in its short-run of $\sim$1.8

seconds. In interpreted mode where object creation is especially noticeable as a bottleneck in the JVM, the slowdown is as much as ∼10%. All other benchmarks are the same, or faster, than their originals[3], as shown in Figure 6.6.



	asac	chromo	decode	Fractal	triphase
▨ SERVER%	0.962	0.957	0.984	1.004	1.000
■ INT%	0.999	0.997	1.101	0.993	1.001

Figure 6.6: *Effects of adding intermediate buffer classes.*

[3]Only benchmarks which were affected by this transformation are shown in Figure 6.6

Chapter 7
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`. Java source code does not have a `goto` statement. Abrupt jumps must be performed through the `break` or `continue` statements which force a certain level of structure on the programmer. The rawest form of abrupt statement possible in source code (shown in Listing 7.1) is a labelled `break` out of a block of code. The very existence of the code block — a limitation of the language — incites structure.

```
label0: {

  while (true) {
    String line = readLine();

    if (line == null)
      break label0; // breaks to end of label0
    else if (isEOF(line))
      break; // breaks out of while loop
  } // end of while loop

  System.out.println("End of file reached");
} // end of label0
```

Listing 7.1: *A Java source code snippet displaying use of a labelled break construct.*

The obfuscations detailed in this chapter attempt to exploit these gaps between the two representations of source and bytecode. Often, this can result in a situation where decompilers either produce incorrect code or do not produce any code whatsoever. Occasionally, the decompiler crashes.

7.1 Exception Handling

The Java language has a special exception handling framework that makes for easy and robust application development. However, due to the way this framework is defined it is possible to take advantage of its structure for obfuscating purposes.

Specifically, *traps* are used to specify sections of bytecode that may throw an exception. In the event this section of code produces an exception, execution control is passed to a specified handler instruction within the same method. Because of this layout, it is implicitly assumed that any instruction within the trap may throw an exception and therefore a proper control flow graph must take these extra branching possibilities into account. This heavily increases complexity with potentially little performance loss.

Normal Java source code, when compiled with `javac`, will produce very predictable bytecode. Catch blocks will always follow their corresponding try blocks within the bytecode and try blocks will not overlap each other. However, it is possible to construct artificial traps within the bytecode such that it is very difficult to properly represent the meaning in source by breaking away from these predictable patterns.

The last four obfuscations detailed in this chapter take advantage of the architecture of Java traps.

7.2 Converting Branches to `jsr` Instructions (CB2JI)

The Java bytecode `jsr` instruction is a historical anomaly. It was originally introduced to handle finally blocks. Finally blocks are sections of code that are guaranteed to run after a try block, whether an exception is thrown within the try block or not.

Nevertheless, the `jsr` is analogous to the `goto` instruction other than the fact that it pushes a return address on the stack. Normally, a `jsr` is used to jump to a subroutine.

Part of the jump process is to place a return address on the stack which is the location that execution should "return" to after the subroutine is run. The return address is normally stored to a register after the jump and when the subroutine is complete the `ret` instruction is used to jump back to the return address.

In this transformation we take advantage of the `jsr`'s ability to jump to different locations by replacing many `if` targets and `goto` instructions within the code with `jsr` instructions. The old jump targets have `pop` instructions inserted before them in order to discard 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 original target (side-stepping the `pop`) or an additional `jsr` is inserted after it which jumps to the `pop` (the very next instruction). The transformation changes almost all control flow to be `jsr` based. Listings 7.2 and 7.3 show a before and after example in bytecode for clarification.

```
public static int sum(int[]);
   0: iconst_0
   1: istore_1
   2: iconst_0
   3: istore_2
   4: iload_2
   5: aload_0
   6: arraylength
   7: if_icmpge 22
   10: iload_1
   11: aload_0
   12: iload_2
   13: iaload
   14: iadd
   15: istore_1
   16: iinc 2, 1
   19: goto 4
   22: iload_1
   23: ireturn
```

Listing 7.2: *Example method* `sum` *in bytecode form* before *obfuscation with* `jsr` *instructions.*

```
public static int sum(int[]);
   0: iconst_0
   1: istore_1
   2: iconst_0
   3: istore_2
   4: goto 8 // side−step return address pop when falling−through
   7: pop // if coming from jsr at #23, pop return address
   8: iload_2
   9: aload_0
   10: arraylength
   11: if_icmpge 26 // jump to jsr at #26
   14: iload_1
   15: aload_0
   16: iload_2
   17: iaload
   18: iadd
   19: istore_1
   20: iinc 2, 1
   23: jsr 7
   26: jsr 29
   29: pop // pop return address off from jsr at #26
   30: iload_1
   31: ireturn
```

Listing 7.3: *Example method `sum` in bytecode form* after *obfuscation with `jsr` instructions.*

The `jsr-ret` construct is particularly difficult to handle when dealing with verification issues (as well as flow analysis) and, in fact, decompilers will usually expect to find a specific `ret` for every `jsr`. If no `ret` instructions are found, it can seriously confuse decompilers. Listing 7.4 presents a method before obfuscation. Listing 7.5 shows the same method after obfuscation and decompilation by `SourceAgain`. While `SourceAgain` is fairly adept at handling these `jsr-ret` issues it still produces heavily nested code with many repeated sequences. This repeating of code is due to the “inlining” of what it perceives to be subroutines but is really normal control flow.

```
public static void main(String args[]) {
  int level = 23;

  if (args.length > 1 && args[1] != null && args[1].equalsIgnoreCase("-i"))
    drawToImage = true;

  if (args.length > 0) {
    String arg = args[0];

    if (arg == null || arg.toLowerCase().indexOf("help") >= 0) {
      printHelp();
      System.exit(1);
    } else {
      try {
        level = Integer.parseInt(arg);
      } catch (NumberFormatException exc) {
        System.out.println("Illegal Depth Number!");
        System.exit(1);
      }
    }
  } else {
      drawToImage = true;
  }
  setLevel(level);
  new Fractal();
}
```

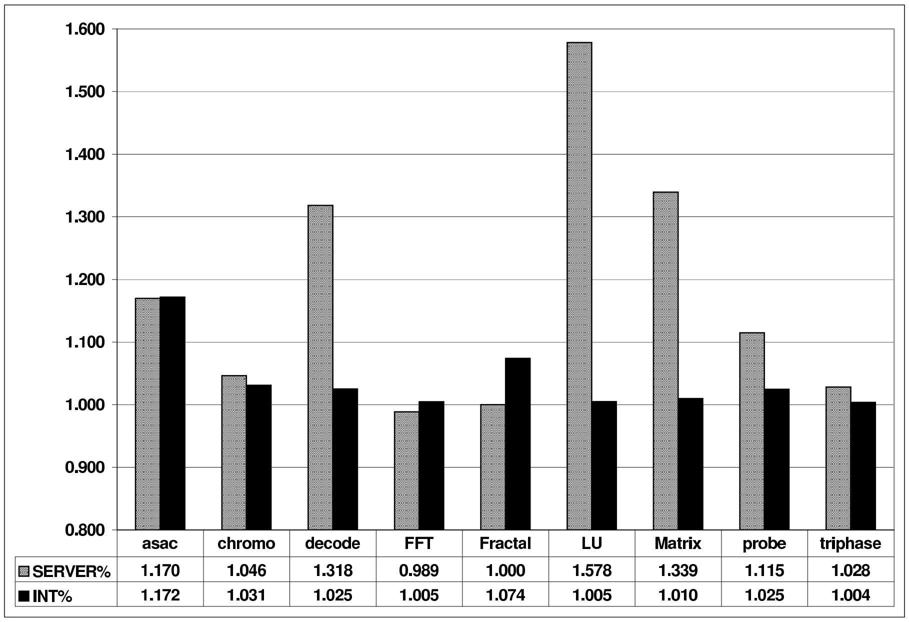
Listing 7.4: *Example method* `main` before *obfuscation with* `jsr` *instructions.*

7.2.1 Performance Results (CB2JI)

The transformation should be expected to slow performance down to some degree. As stated in Chapter 3 subroutines are complex and, when used in an irregular way, will likely cause slow downs in the verifier and in JIT compilation. Certainly, this is apparent in Figure 7.1 where `Decode`, `LU`, and `Matrix` are particularly slow in server mode. This makes sense. The entire reasoning behind this obfuscation is that the `jsr-ret` subroutine framework is inherently difficult for flow analysis and most decompilers will except to instances of `jsr` instructions with no matching `ret`. Proper typing and call graph cycle-finding under this kind of environment is hard.

```
public static void main(String[] as) {
  int i = 23;

  if(as.length > 1 && as[1] != null && as[1].equalsIgnoreCase("-i"))
    drawToImage = true;

  if(as.length > 0) {
    String s = as[0];

    if(s == null || s.toLowerCase().indexOf("help") >= 0) {
      printHelp();
      System.exit(1);
      setLevel(i);
      new Fractal();
    } else {
      try {
        i = Integer.parseInt(s);
      } catch( NumberFormatException numberformatexception1 ) {
        System.out.println("Illegal Depth Number!");
        System.exit(1);
      }
      setLevel(i);
      new Fractal();
    }
  } else {
    drawToImage = true;
    setLevel(i);
    new Fractal();
  }
  return;
  setLevel(i);
  new Fractal();
}
```

Listing 7.5: *Example method* `main` after *obfuscation with* `jsr` *instructions. Decompiled by* `SourceAgain` *— trailing statements after* `return` *make this illegal code.*



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.170	1.046	1.318	0.989	1.000	1.578	1.339	1.115	1.028
■ INT%	1.172	1.031	1.025	1.005	1.074	1.005	1.010	1.025	1.004

Figure 7.1: *Effects of replacing `goto` instructions with `jsr` instructions.*

Using the HotSpot `-XX:+CITime` option shows that `LU` is performing JIT compilation at approximately 10∼20% of the speed seen when running other benchmarks. Some brief exploration into the code yields a particularly dense method, `factor`, which has heavily nested `for` loops. Excluding this single method from compilation yields a server-mode runtime almost exactly that of the interpreted mode speed and also returns the JIT compilation speed in bytes per second to normal. This one method is responsible for slowing the entire performance down with complex control flow.[1]

7.3 Reordering `loads` Above `if` Instructions (RLAII)

This simple transformation looks for situations where a local variable is used directly following both paths of an `if` branch. This is a somewhat common occurrence — consider code that follows the pattern `if x then i++ else i--`. Along both branches the

[1]This exploration serves as a good way to track down these types of methods in order to limit the obfuscation performed on them through the weighting mechanism described in Section 8.1.

first course of action will be for the local variable `i` to be loaded onto the stack (as shown in bytecode form in Listing 7.6 where `i` is in register 1 and `x` is in register 2). This obfuscation moves the `load` of `i` above the `if`, removing the two copies of `load`, one each along the `true` and `false` branches. This is shown in bytecode form in Listing 7.7.

```
4: iload_2
5: ifeq 16
8: iload_1
9: iconst_1
11: iadd
12: istore_1
13: goto 21
16: iload_1
17: iconst_1
19: isub
20: istore_1
```

Listing 7.6: *Example bytecode snippet* before *reordering* `load` *instructions above* `if` *instructions.*

```
0: iload_2
1: iload_1
2: swap
3: ifeq 13
6: iconst_1
8: iadd
9: istore_0
10: goto 17
13: iconst_1
15: isub
16: istore_0
```

Listing 7.7: *Example bytecode snippet* after *reordering* `load` *instructions above* `if` *instructions.*

While a modern decompiler based on control-flow graph analysis should be able to overcome this obfuscation with little problem, any decompiler relying on pattern matching (such as the venerable `Jad`) has the potential to become very confused. Consider the small function `sum` presented in its original source code form in Listing 7.8. The main branch in the method is the head of the `for` loop. During normal execution of the loop, the branch instruction follows on to the aggregated addition statement. In the case where `i > iarry.length`, that is the loop condition fails, the branch instruction will jump to the return statement. In both statements, the local variable `result` is used immediately. Listing 7.9 shows the decompiled `SourceAgain` code for the same method after it has been obfuscated by this transformation. Notice that the method is both very garbled and *not at all* semantically equivalent to the original. This is surprising given the graph-based techniques `SourceAgain` claims to employ. `Dava` produces correct output, shown in Listing 7.10.

```
public static int sum(int iarry[]) {
  int result = 0;

  for (int i = 0; i < iarry.length; i++) {
    result += iarry[i];
  }

  return result;
}
```

Listing 7.8: *Example method* `sum` before *reordering* `load` *instructions above* `if` *instructions.*

7.3.1 Performance Results (RLAII)

This transformation, which moves load instructions above if branches, does not cause very drastic changes to the code. Very little performance effect should be expected from this modification. It is possible that specific instruction caching optimizations in the VM implementation take advantage of this code pattern to improve data lookup. It is not surprising to see speedups in some cases.

Indeed, the tests show (see Figure 7.2) very little change in running speed at all. Small speedups were seen in four of the benchmarks in server mode and it is quite possible that the effect of this transformation is similar to the effect of an early load optimization suggested in the previous paragraph.

```
public static int sum(int ai[]) {
  int i = 0;
  int j = 0;
  do {
    if(ai.length >= i)
      break;
    i = j + ai[j];
    j++;
  } while(true);
  return;
}
```

Listing 7.9: *Example method* `sum` after *reordering* `load` *instructions above* `if` *instructions. Decompiled by* `SourceAgain` *— it is not semantically equivalent to the original.*

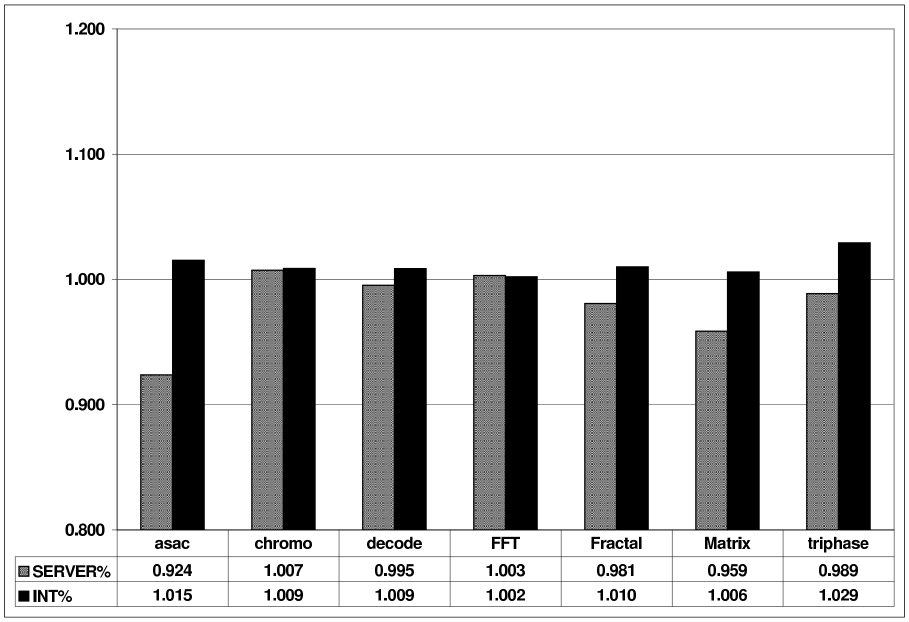
```
public static int sum(int[] r0) {
  int i0, i1;
  i0 = 0;

  for (i1 = 0; i1 < r0.length; i1++) {
    i0 = i0 + r0[i1];
  }

  return i0;
}
```

Listing 7.10: *Example method* `sum` after *reordering* `load` *instructions above* `if` *instructions. Decompiled by* `Dava` *— it is correct and almost identical to the original code.*

7.4 Disobeying Constructor Conventions (DCC)

The Java language specification by Gosling, *et al*. 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 statement.

> "If a constructor body does not begin with an explicit constructor invocation and the constructor being declared is not part of the primordial class Object,



	asac	chromo	decode	FFT	Fractal	Matrix	triphase
▨ SERVER%	0.924	1.007	0.995	1.003	0.981	0.959	0.989
■ INT%	1.015	1.009	1.009	1.002	1.010	1.006	1.029

Figure 7.2: *Effects of moving `load` instruction above `if` branch instructions.*

> then the constructor body is implicitly assumed by the compiler to begin with a superclass constructor invocation "super();", an invocation of the constructor of its direct superclass that takes no arguments." Section 2.12 [GJSB00]

In the event that no super constructor is called the Java compiler explicitly adds the call at the top of the method in the compiled output.

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 transformation does exactly this.[2] It randomly chooses among four different approaches to transforming constructors in order to confuse decompilers. Of the four techniques described below, all of them consider only simple super constructors with zero pa-

[2]Notice in Listings 7.12 and 7.13 that because the "super" call is not where the decompiler expects it to be, it is unable to label it as such and instead the actual underlying method name, `<init>`, appears. This is true for all four approaches. Listing 7.14 attempts to call a constructor based on the local name for the object reference — this is slightly different because it is `SourceAgain` output; `Dava` was unable to decompile it.

rameters. They could easily be extended to include all super constructors.

Wrap in Try: The first approach simply wraps the super constructor method call within a try block. This ensures that any decompiled source will be *required* to wrap the super constructor call in a try block to conform to the exception handling rules of Java. To properly allow the exception to propagate, the handler unit — a `throw` instruction — is appended to the end of the method. However, this conversion immediately renders the source uncompilable. Listing 7.11 shows the original source of a simple object constructor. Listing 7.12 presents the same constructor *after* transformation (note that this is illegal source code and will not compile).

```
ObjectA() {
  super();
}
```

Listing 7.11: *Example object constructor method in Java source code form.*

```
ObjectA() {
  try {
    this.<init>();
  } catch (IndexOutOfBoundsException $r2) {
    throw $r2;
  }
}
```

Listing 7.12: *Example object constructor* after *wrap in try transformation. Decompiled by* `Dava` *— it is not legal source code.*

Throw Throwable: The second approach takes advantage of classes which are children of `java.lang.Throwable`. These are classes which can be thrown by the exception handling mechanism of Java. It inserts a `throw` instruction before the super constructor call and creates a new try block in the method that traps just the new `throw` instruction. The handler unit is designated to be the super constructor. This takes advantage of the fact that the class itself is throwable and can be pushed onto the stack through the throw mechanism instead of the standard push mechanism. The new try block can trap exceptions of the parent type, since the super constructor is a method from the parent type. This will further confuse the decompilers, as seen in Listing 7.13.

```
ObjectA() {
  try {
    throw this;
  } catch (Throwable $r2) {
    $r2.<init>();
    return;
  }
}
```

Listing 7.13: *Example object constructor after throwing itself as a throwable approach. ObjectA extends the* `java.lang.Throwable` *object. Decompiled by* `Dava` *— it is not legal source code.*

Indirect through jsr: The third approach simply inserts a `jsr` jump instruction 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.

Trap Ifnull Indirection: The final approach is the most complicated. Directly before the super constructor call new instructions are added: a `dup` followed by an `ifnull`. The `ifnull` target is the super constructor call itself. This `if` branch instruction will clearly always be `false` since the object it is comparing is the object being instantiated in the current constructor. Two new instructions are inserted directly after the `if` (the `false` branch): a `push null` followed by a `throw`. A new trap is created spanning from the `ifnull` instruction up to the super constructor 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 constructor call.

```
ObjectA() {
  ObjectA tz1 = this;

  try {
    if(tz1 != null)
      throw tz1;
  } catch(IndexOutOfBoundsException unused2) {
    tz1 = this;
  }
  tz1();
}
```

Listing 7.14: *Example object constructor after adding trapped `ifnull` indirection. Decompiled by `SourceAgain` — it is not legal source code.*

This final approach confuses decompilers because it is more difficult to deduce which local will be on the stack when the super constructor call site is reached. While `Dava` crashes trying to decompile this approach, `SourceAgain` produces the improper output in Listing 7.14.

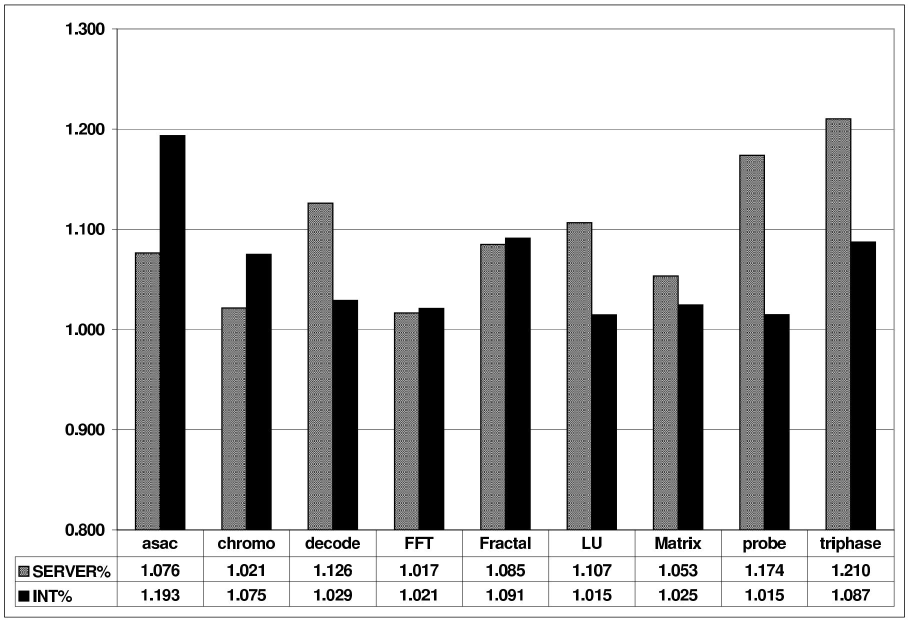
7.4.1 Performance Results (DCC)

Having discovered the cost of object creation and the significant performance penalties that can be incurred when tampering with that process (see Section 6.5.1) it is not surprising to see a similar, albeit more universal, slowdown due to this constructor transformation.

In fact, the performance penalties seen here in Figure 7.3 are fairly large and it is clear that this transformation should be used sparingly, if at all. One should be especially careful in transforming programs tasked with large amounts of object creation.

7.5 Partially Trapping Switch Statements (PTSS)

Another way to exploit the obfuscating potential of the try-catch block 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



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.076	1.021	1.126	1.017	1.085	1.107	1.053	1.174	1.210
■ INT%	1.193	1.075	1.029	1.021	1.091	1.015	1.025	1.015	1.087

Figure 7.3: *Effects of confusing constructor method code.*

different blocks of code. Which block gets executed is controlled by an integer value. However, in bytecode there is nothing explicitly tying the `switch` instruction to the different code blocks — that is, there is no explicit encapsulation.

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

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.

The exception which is trapped by the new try-catch block is randomly choosen from the set of RuntimeExceptions.

Listing 7.15 gives a brief method containing a `switch` statement. Figures 7.4 and 7.5 show the control flow graph for the method before and after this transformation has been applied, respectively. Finally, Listing 7.16 displays the source code produced by Dava after the transformation; it is clearly much more difficult to ascertain the general purpose of the method given the extra nesting levels and additional abrupt breaks.

```
public static int calc(int oper, int val, int total) {
  switch (oper) {
    case 0:
      total += val;
      break;
    case 1:
      total -= val;
      break;
    case 2:
      total *= val;
      break;
    default: System.out.println("Invalid Oper!");
  }
  return total;
}
```

Listing 7.15: *Example method* `calc` before *its switch is partially trapped in a try-catch block.*

7.5.1 Performance Results (PTSS)

This transformation was run *after* applying the transformation from Section 6.1 which adds dead-code switch statements. This was to ensure that every benchmark had switch statements available and that this transformation would actually have an effect. This is not the most thorough test since most of the switches being agumented are never actually executed as branching statements; they are wrapped in opaque predicates. Nevertheless, the try blocks which are added by this transformation are only added with the intent to confuse decompilers and should not effect execution speed significantly unless exceptions are actually thrown and exception table lookups occur.

In general, there is no change in the speed of execution, however there are exceptions. `chromo` is slower in interpreted mode while `LU` and `Matrix` are slower in server mode. `triphase` is the worst with over 30% slowdown in both modes. Running the Java pro-

```
public static int calc(int i0, int i1, int i2) {
  label_4: {
    label_3: {
      label_2: {
        label_1: {
          label_0: {
            try {
              switch (i0) {
                case 0:
                  i2 = i2 + i1;
                  break label_0;
                case 1:
                  break label_1;
                case 2:
                  break label_2;
                default:
                  break label_3;
              }
            } catch (NullPointerException $r5) {
              throw $r5;
            }
            break label_4;
          } // end of label_0

        } // end of label_1

        i2 = i2 − i1;
        break label_4;
      } // end of label_2

      i2 = i2 ∗ i1;
      break label_4;
    } // end of label_3

    System.out.println("Invalid Oper!");

  } // end of label_4
  return i2;
}
```

Listing 7.16: *Example method* `calc` after *its switch is partially trapped in a try-catch block. Decompiled by* `Dava` *— it is semantically equivalent to the original but much more difficult to decipher.*

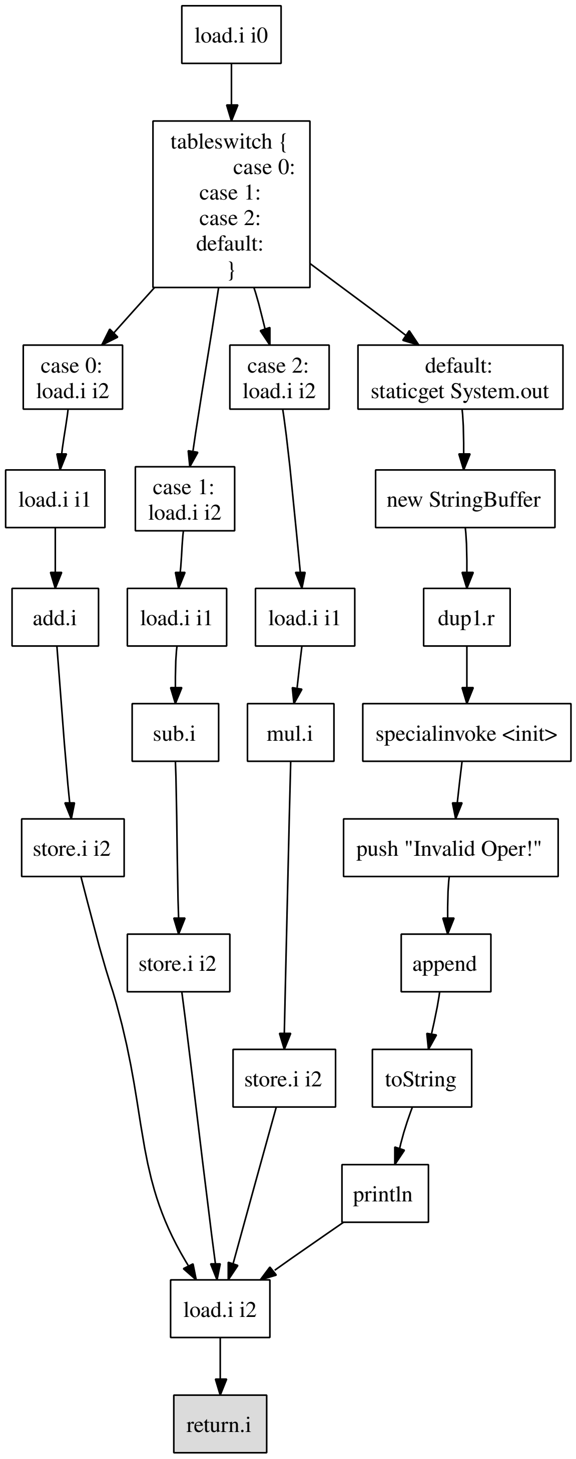


Figure 7.4: *Control-flow graph of the* `calc` *method* before *partially trapping its switch statement.*

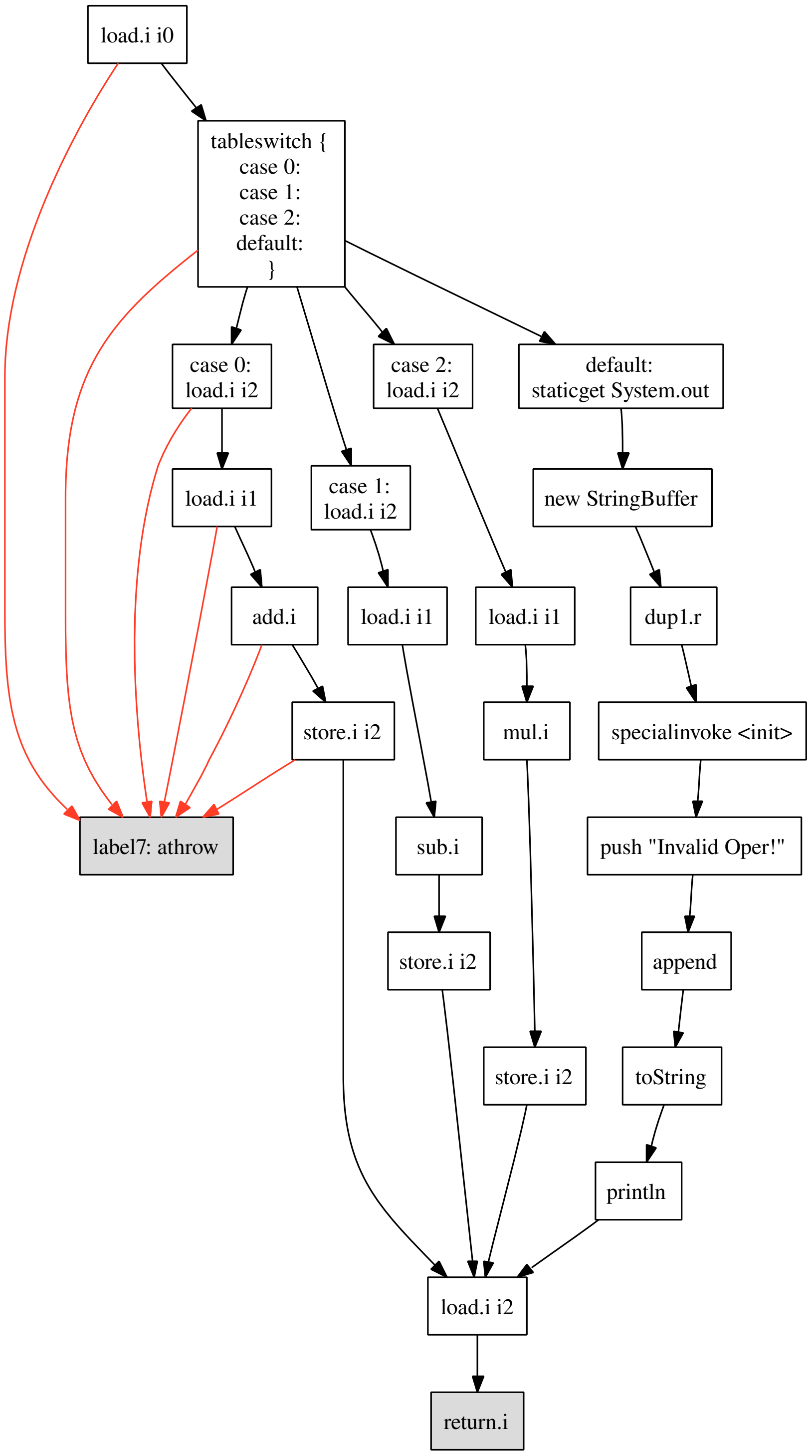
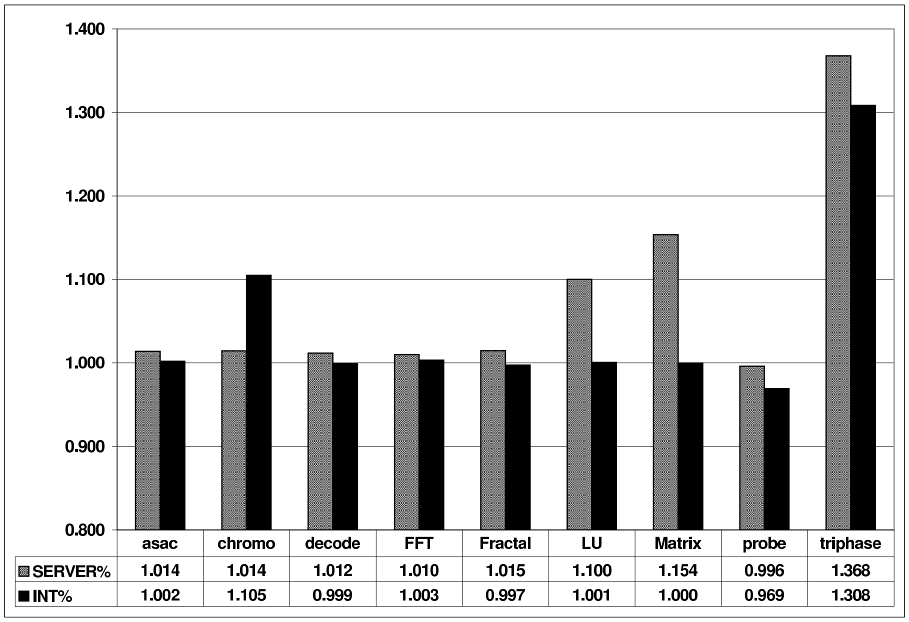


Figure 7.5: *Control-flow graph of the* `calc` *method* after *partially trapping its switch statement. Exceptional edges are shown in red.*



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.014	1.014	1.012	1.010	1.015	1.100	1.154	0.996	1.368
■ INT%	1.002	1.105	0.999	1.003	0.997	1.001	1.000	0.969	1.308

Figure 7.6: *Effects of partially wrapping switch statements in try blocks.*

filer on the obfuscated versions of `triphase` shows that roughly 60% of the running time is spent in a single method, `daxpy`. The original version does not register this method in the profiler at all. Using the weighting mechanism in JBCO to exclude `daxpy` from this obfuscation we find that only 35 switch instructions are obfuscated instead of the initial 38. Furthermore, the $\sim$30% slowdowns in server and interpreted mode disappeared completely.

7.6 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, this structure is absent; 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 is because try blocks only list an offset in the code to jump to in the case of an exception.

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 which 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 set back to its original value when the try section is completed. A small bytecode example is shown in Listing 7.17

```
Exception trapped from 2 through 16

1: goto 70 // goto the control flag initializer.
2: iload_1 // load control flag
3: ifne 39 // if control flag is not zero, jump to the catch block
4: pop // if control flag is zero, pop null and continue with try block
5: iinc 1, 1 // once in try block, mark control flag
13: new #36; //class x
14: dup
15: invokespecial #18; //Method "<init>":()V
16: return
```

Listing 7.17: *Example try block which has been combined with its catch block. Register 1 contains the control flag.*

Listing 7.18 shows a method before applying this transformation. Listing 7.19 shows the same method after obfuscation. The code is clearly more obscure.

7.6.1 Performance Results (CTBCB)

A certain amount of overhead is incurred by this transformation in the form of a control local lookup and branch instruction. Because this is performed everytime a try block or its equivalent catch block is entered, this could lead to significant performance degradation if the code is in a nested loop. Luckily, it is generally known to be bad coding style to nest try-catch blocks in loops and so this should rarely be an issue.

The numbers in Figure 7.7 certainly suggest that there should be very little concern for performance changes due to this transformation. Among the significant obfuscations, this

```
public static void main (String args[]) {
  x i;

  try {
    try {
      i = new x();
      i.foo();
    } catch (x exc) {
      System.out.println("inner exc");
      if (exc.eventime == 0)
        throw exc;
    }
  } catch (x exc1) {
    System.out.println("outer exc");
  }
}
```

Listing 7.18: *Example method* `main` before *try blocks are combined with their catch blocks.*

one seems to have almost no effect whatsoever on the timings of the benchmarks.

7.7 Indirecting `if` Instructions (III)

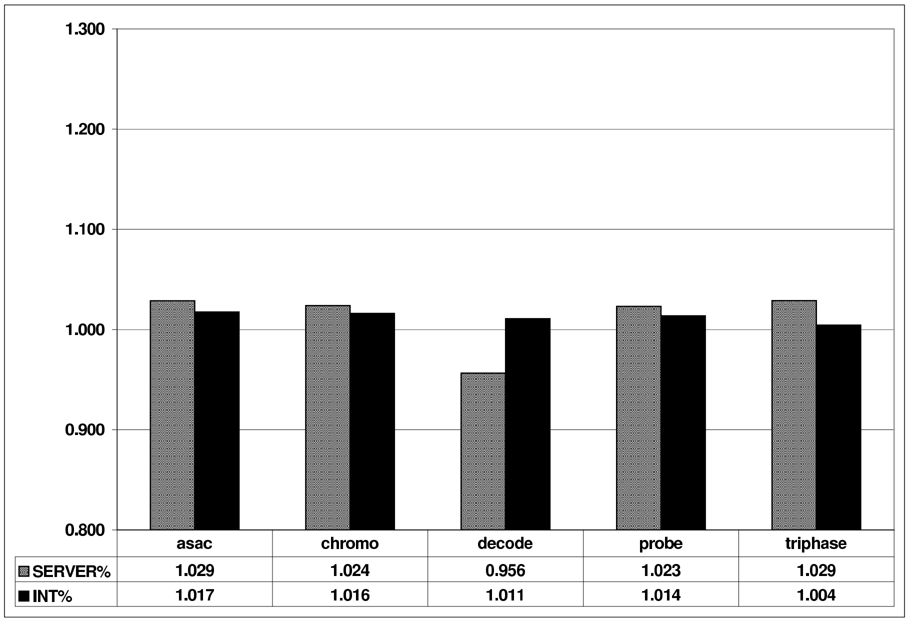
One very basic example of bytecode that is not directly translatable to source code is the `goto` instruction. Explicit gotos (*i.e.*, those which are not part of loop directives) are not allowed in Java source. Studies show that the goto is largely misused and removing it altogether simplifies the language — specifically the research by Benander, *et al*. [BGB90]. Nevertheless, Java maintains the ability to break out of nested loops through the use of the `continue` and `break` statements.

Despite the lack of a goto directive in the Java source language, Java bytecode does maintain a `goto` instruction. This is necessary since higher-level source code constructs such as while loops are implemented within bytecode with `if` and `goto` instructions. Therefore it is possible to insert explicit `goto` instructions within the bytecode.

This `if`-indirection transformation attempts to take advantage of the trap mechanism while also exploiting the bytecode `goto`. It identifies a number of `if` instructions within

```
public static void main(String[] r0) {
  boolean z0 = false;
  int i0 = 0;

  label_5:
  while (true) {
    label_4: {
      label_3: {
        label_0: {
          if (i0 != 0)
            break label_0;
          i0++;
          z0 = false;
          break label_3;
        } //end label_0:
        try {
          System.out.println("inner exc");
          if ($r1.eventime != 0)
            break label_4;
          throw $r1;
        } catch (x $r2) { }
      } //end label_3:
      label_1:
      while (true) {
        label_2: {
          try {
            if ( ! (z0)) {
              z0 = true;
              (new x()).foo();
              break label_2;
            }
          } catch (x $r2) {
            continue label_1;
          } catch (x $r1) {
            continue label_5;
          }
          System.out.println("outer exc");
          break label_4;
        } //end label_2:
        break label_4;
      }
    } //end label_4:
    return;
  }
}
```

Listing 7.19: *Example method* `main` after *try blocks are combined with their catch blocks. Decompiled by* `Dava`*. The code is not correct.*



	asac	chromo	decode	probe	triphase
▨ SERVER%	1.029	1.024	0.956	1.023	1.029
■ INT%	1.017	1.016	1.011	1.014	1.004

Figure 7.7: *Effects of combining try blocks with their catch blocks.*

each method in order to *indirect* their branching. The word indirect is used here to describe a situation where an `if` instruction jumps to a `goto` instruction, which then jumps to the final target. The `goto` could, of course, be removed and the `if` instruction could be modified to jump to the final target. However, this transformation wraps all these indirections — a block of `goto` instructions — within a trap. It is not valid to remove traps in bytecode unless it can be statically shown that an exception will never be raised while executing code within the trap. Since there is no explicit `goto` allowed in Java source, it becomes very difficult for a decompiler to synthesize equivalent source code. An example of decompiled code generated by SourceAgain is shown in Listing 7.20. It is the same `sum` method originally shown in Listing 4.1, where only one `if` is available for obfuscation. `Dava` crashes when trying to handle bytecode that throws an explicitly `null` exception.

The transformation is implemented as follows. First, each `if` instruction is identified and its target is recorded. A new `goto` pointing to the target is then appended to the bytecode and the `if` instruction is reassigned to branch to the new `goto`. This added level of indirection is wrapped in a try block. In the case where the original `if` instruction was

```
public static int sum(int[] ai) {
  int i = 0;
  int j = 0;

  while(j < ai.length) {
    Object tobj;
    i += ai[j];
    ++j;
    if( (byte) 654154038 % 9 == 0 )
      continue;
    tobj = null;
    return texception1;
  }
  try {
  } catch(Exception texception1) {
    throw texception1;
  }
  return i;
}
```

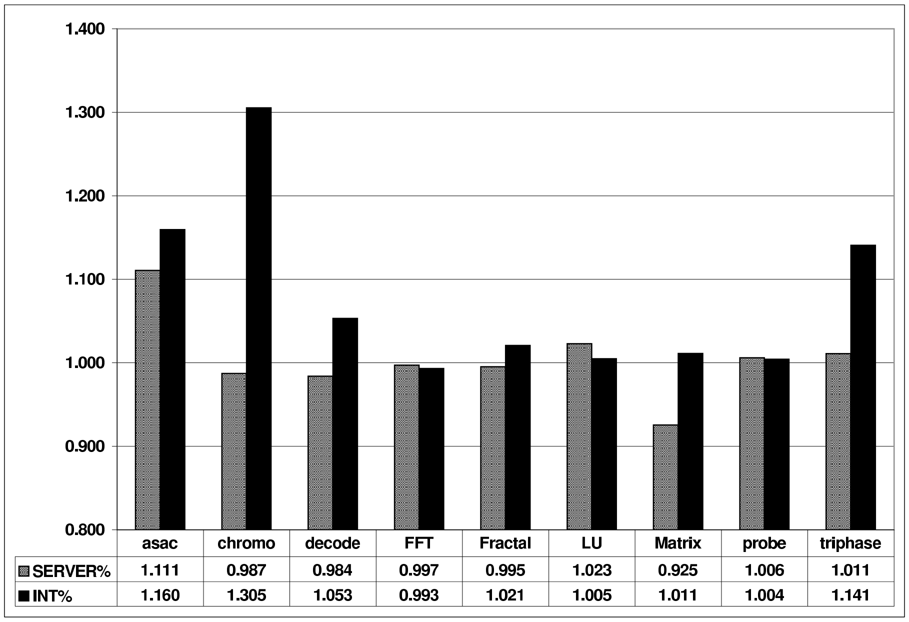
Listing 7.20: *Example method* `sum` after *its* `if` *instructions have been indirected through trapped* `goto` *instructions. Decompiled by* `SourceAgain` *— this is improper code and will not recompile.*

trapped, the same exception is trapped in the new `goto` and the same handler instruction is used. Otherwise, a random exception is assigned to be caught at the new `goto` and the handler instruction just re-throws the exception upward.

Furthermore, if a handler instruction is generated by this transformation — that is, at least one indirected `if` is *not* trapped — then it is used for every untrapped `if` that is indirected. In order to confuse the reasoning behind the existence of the handler, it is inserted at a randomly chosen spot within the method (although the spot must have a stack height of zero, as calculated by the `StackHeightTypeCalculator` described in Section 4.2.2) and an opaque predicate is inserted before the handler which will always branch over it.

7.7.1 Performance Results (III)

This control flow indirection transformation is powerful in that it generally does not have a negative effect on performance. As in earlier transformations, there are a few key exceptions as can be seen in Figure 7.8. As usual, the slowdowns almost always occur due to a single method that is particularly complicated. In these situations the JIT is just not able to optimize the modified method as much as its original version.



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.111	0.987	0.984	0.997	0.995	1.023	0.925	1.006	1.011
■ INT%	1.160	1.305	1.053	0.993	1.021	1.005	1.011	1.004	1.141

Figure 7.8: *Effects of indirecting `if` instructions through trapped `goto` instructions.*

In the case of `asac` a single sorting method accounts for the entire slowdown in both the server and interpreter modes. In other cases, such as `chromo`, `decode`, `Matrix`, and `triphase`, the JIT is able to successfully optimize away any complicated structure introduced by the transformation.

7.8 `Goto` Instruction Augmentation (GIA)

The approach of this transformation uses the same `goto`-exploit from the previous section to create a very simple yet useful obfuscation. 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 sequence now looks like:

$$\{\,\texttt{goto}\, P_1, P_2, P_1, \texttt{goto}\, P_2\}$$

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.

While this transformation is very easily reversed when used without the try block, it nevertheless stops many simple decompilers from properly decompiling the entire method. `Jad`, for example, outputs bytecode level instructions in those places where it cannot properly deduce a source code equivalent. With the try block things become much more difficult and even `SourceAgain` fails to properly decompile this obfuscation. See Listing 7.21 and 7.22 for a before and after example using `Dava`.

```java
private static double[][] RandomMatrix(int M, int N, Random R) {
  double A[][] = new double[M][N];

  for (int i=0; i<N; i++) {
    for (int j=0; j<N; j++)
      A[i][j] = R.nextDouble();
  }

  return A;
}
```

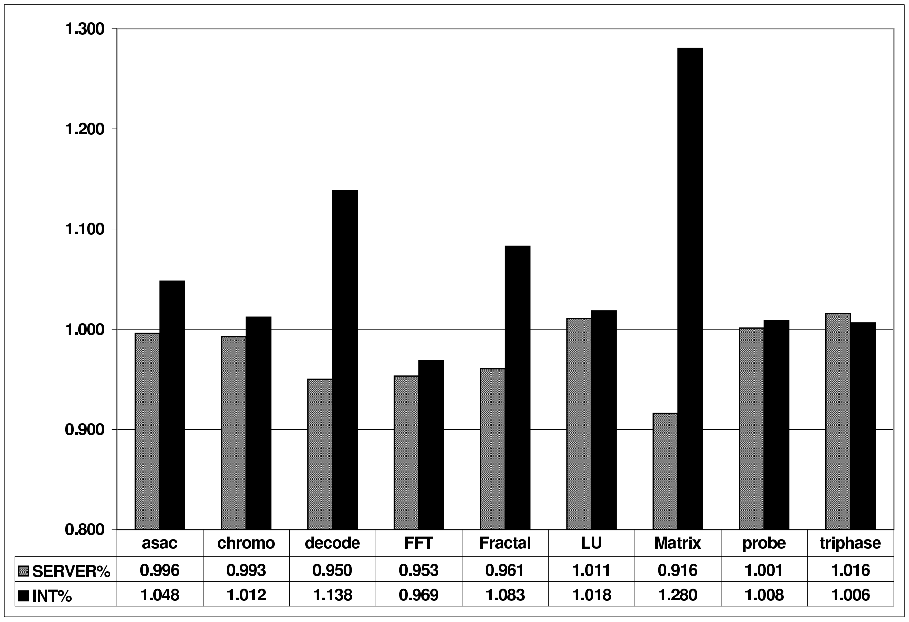
Listing 7.21: *Example method* `RandomMatrix` before *augmentation with new trapped* `goto` *instructions.*

```
private static double[][] RandomMatrix(int i0, int i1, Random r0) {
  double[] r1;
  int i2, i3;
  double d0;
  double[][] r2;

  try {
    r2 = new double[i0][i1];
    i3 = 0;
  } catch (Throwable $r3) {
    throw $r3;
  }

  while (i3 < i1) {
    for (i2 = 0; i2 < i1; i2++) {
      r1 = r2[i3];
      d0 = r0.nextDouble();
      r1[i2] = d0;
    }

    i3++;
  }

  try {
    return r2;
  } catch (Throwable $r3) {
    throw $r3;
  }
}
```

Listing 7.22: *Example method* `RandomMatrix` after *augmentation with new trapped* `goto` *instructions. Decompiled by* `Dava`*.*

7.8.1 Performance Results (GIA)

The explicit goto instrumentation is a direct attack on a known weakness of Java decompilers. Because of its simple nature it effects runtime performance very little. Only in particularly small transformed methods do we sometimes see a measurable slowdown in interpreted mode (see Figure 7.9). As the method is called many times over, which is often the case with small methods, the interpreter must continue to follow the exact instructions step by step. In server mode the JIT is able to optimize out the unnecessary and pointless goto jump as do some graph-based decompiler analyses (such as in `Dava`).



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	0.996	0.993	0.950	0.953	0.961	1.011	0.916	1.001	1.016
■ INT%	1.048	1.012	1.138	0.969	1.083	1.018	1.280	1.008	1.006

Figure 7.9: *Effects of augmenting methods with explicit `goto` instructions.*

In the case of `LU`, where we see a speedup in server mode, we found just such a case. Two small methods used to set and get matrix elements were affected by the transformation. In the original benchmark the code did not take up enough execution time to get inlined by the JIT. However, in the transformed code, these methods are given two optimization passes by the JIT and are inlined, resulting in improved running time. Turning off all inlining results in almost identical speed between the two benchmark versions.

Chapter 8
Introducing a Solid, Stable, and Effective Third-Generation Obfuscator

In chapters five through seven we have developed a number of obfuscations, many of which are unique and exploit the gap between Java source code and its binary bytecode representation. To make these transformations useful they need to be combined within an automatic tool that has an intuitive and simple interface. This is JBCO. We call it a *third-generation* obfuscator because it builds upon a lot of the early work in obfuscation (renaming identifiers, confusing control flow), but also incorporates later object-oriented obfuscations as well as new and original ideas which have been specifically designed to cause existing decompilers to produce illegal output or make them crash altogether.

In the following sections we will introduce the interface, derive three default combinations (one each tuned for speed, size, and protection) and present experimental performance graphs for those combinations.

8.1 JBCO User Interface

JBCO was designed as a command-line tool and therefore it is very versatile and can easily be incorporated into a build process. Each transformation available in JBCO can be activated independently and, depending on the severity of obfuscation desired, can be weighted independently within a range of 0–9. If set to 0, the transformation will not be applied anywhere and if set to 9 it will be applied everywhere possible. The argument format for specifying a transformation is:

`-t:`*W*`:`*SP*`.jbco_`*PN*

Where *W* is the weight, *SP* is the `Soot` phase the transformation belongs to (`wjtp` for whole program Jimple transformation pack, `jtp` for normal Jimple transformation pack, and `bb` for Baf body pack), and *PN* is the phase name, such as `ADSS` for the obfuscation described in Section 6.1 that adds dead-code switch statements to method bodies. All available transformations and extra options can be listed using the command-line option `-jbco:help` (see Listing 8.1).

Special weighting can be given to a certain method or a group of methods whose names match a regular expression. The argument format for specifying a special case such as this is:

`-it:`*W*`:`*SP*`.jbco_`*PN*`:`*E*

Where *E* is the name of the method to assign the special weight *W* to or, alternately, a regular expression to match method names to. *E* should be prepended with an asterix (*) to signify that it is a regular expression. The following, for example, would match all methods starting with the term "get" and give them a weight of 5 when adding dead-code switches:

```
-it:5:jtp.jbco_adss:*get*
```

When giving specific arguments for obfuscations which operate on the class or field level (such as the class renamer or field renamer, respectively) then the argument *E* will, of course, match on class names or field names instead of method names.

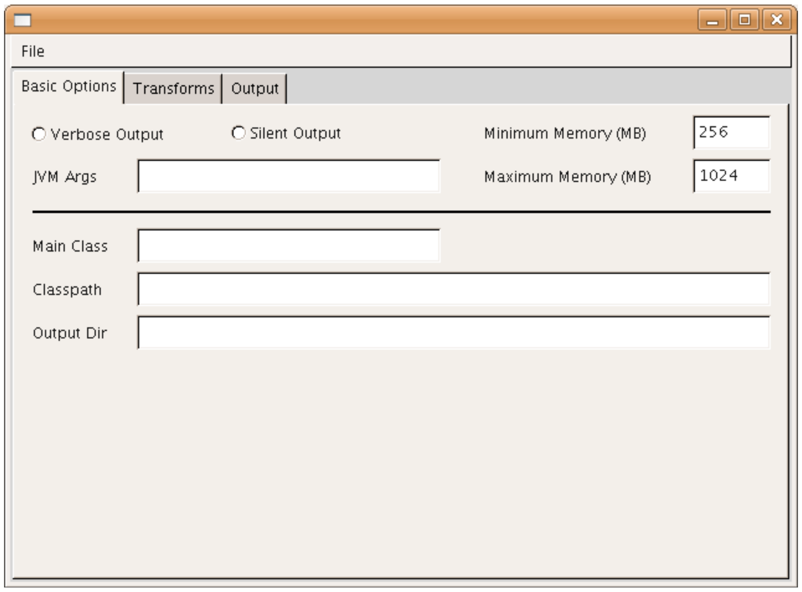
A graphical user interface (GUI) has also been developed for JBCO. It is a simple front end that allows a user to specify the same options discussed above but in a graphical format.

The first panel of the GUI, seen in Figure 8.1, allows the user to specify the classpath JBCO will use, along with memory requirements, the output directory, and any other arguments for the JVM and not JBCO itself.

The second panel of the GUI, shown in Figure 8.2, lists all the possible obfuscations in a list. Each one is fully enabled with a weight of 9 by default. However, the `File` menu has

```
General Options:
          −jbco:help − print this help message.
          −jbco:verbose − print extra information during obfuscation.
          −jbco:silent − turn off all output, including summary information.
          −jbco:metrics − calculate total vertices and edges;
                              calculate avg. and highest graph degrees.
          −jbco:debug − turn on extra debugging like
                              stack height and type verifier.

Transformations ( −t:[W:]<name>[:pattern] )
          W − specify obfuscation weight (0−9)
          <name> − name of obfuscation (from list below)
          pattern − limit to method names matched by pattern
                              prepend * to pattern if a regex

          wjtp.jbco_cr − Rename Classes
          wjtp.jbco_mr − Rename Methods
          wjtp.jbco_fr − Rename Fields
          wjtp.jbco_blbc − Build API Buffer Methods
          wjtp.jbco_bapibm − Build Library Buffer Classes
          jtp.jbco_gia − Goto Instruction Augmentation
          jtp.jbco_adss − Add Dead Switch Statements
          jtp.jbco_cae2bo − Convert Arith. Expr. To Bitshifting Ops
          bb.jbco_cb2ji − Convert Branches to JSR Instructions
          bb.jbco_dcc − Disobey Constructor Conventions
          bb.jbco_rds − Reuse Duplicate Sequences
          bb.jbco_riitcb − Replace If(Non)Nulls with Try−Catch
          bb.jbco_iii − Indirect If Instructions
          bb.jbco_plvb − Pack Locals into Bitfields
          bb.jbco_rlaii − Reorder Loads Above Ifs
          bb.jbco_ctbcb − Combine Try and Catch Blocks
          bb.jbco_ecvf − Embed Constants in Fields
          bb.jbco_ptss − Partially Trap Switches
```

Listing 8.1: *JBCO help listing.*



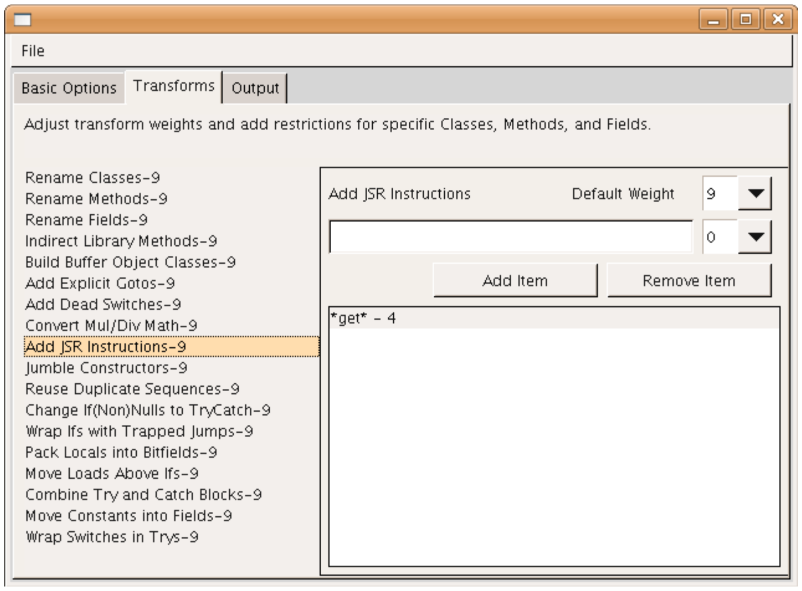
Minimum Memory (MB) 256
Maximum Memory (MB) 1024

Figure 8.1: *The JBCO graphical user interface — JVM Options.*

three items which allow the user to select a default set of obfuscations and weightings based on the desired outcome: high program performance, low program size, or high protection. These three combinations are detailed in Section 8.2.

The user can also change the weights individually for each obfuscation, which are indicated by the number following the transformation name. The weight can be changed by clicking on the name of the desired transformation and adjusting its "default weight". Specific weights can be assigned to methods by entering the method name or regular expression in the text box, selecting the appropriate weight to the right of the text box, and then clicking the "Add Item" button.

To execute the options specified, the user must select `File - Execute` from the menu. At this point an output panel will raise to the top and all console messages generated by JBCO will appear in this window. An option to save the output to a file is given at the bottom of the pane.



Adjust transform weights and add restrictions for specific Classes, Methods, and Fields.

Rename Classes-9
Rename Methods-9
Rename Fields-9
Indirect Library Methods-9
Build Buffer Object Classes-9
Add Explicit Gotos-9
Add Dead Switches-9
Convert Mul/Div Math-9
Add JSR Instructions-9
Jumble Constructors-9
Reuse Duplicate Sequences-9
Change If(Non)Nulls to TryCatch-9
Wrap Ifs with Trapped Jumps-9
Pack Locals into Bitfields-9
Move Loads Above Ifs-9
Combine Try and Catch Blocks-9
Move Constants into Fields-9
Wrap Switches in Trys-9

Add JSR Instructions   Default Weight 9

0

get - 4

Figure 8.2: *The JBCO graphical user interface — transformation options.*

8.2 Combining Useful Transformations for a Practical Obfuscator

The true power of our obfuscator comes from its combinatory nature. The many transformations that we have developed can all be applied to the same program in a sequential manner. By mixing multiple obfuscations together it becomes that much harder for a decompiler to perform any sort of analysis based on pattern matching and also makes control-flow based approaches a lot more costly due to a sharp rise in method complexity.

It must be noted, however, that the order in which the obfuscations are performed has a significant impact on performance. For example, if obfuscations which create try blocks are run before the try-catch block combiner (CTBCB) detailed in Section 7.6 then more "obfuscation sites" will be available to that obfuscation, causing more performance degradation. This is a topic for future research. In the combinations presented in this section the order of obfuscations was not chosen in any meaningful way and, indeed, better performance could have been achieved by carefully considering the effects each obfuscation would have on obfuscations later in the order. The orders that were used are shown in Table 8.1.

Table 8.1: *Order of obfuscations for three combinations*

Performance	Size	Protection
RI(C,M,F)	BLBC	RI(C,M,F)
BAPIBM	GIA	BLBC
BLBC	CAE2BO	BAPIBM
CAE2BO	CB2JI	GIA
ADSS	DCC	ADSS
GIA	RIITCB	CAE2BO
PLVB	RLAII	CB2JI
RDS	CTBCB	DCC
RIITCB	PTSS	RDS
RLAII		RIITCB
CTBCB		III
III		PLVB
		RLAII
		CTBCB
		PTSS

Excessive obfuscation can lead to very poor performance or severe program size bloat. The sequence of obfuscations chosen for a program should match an overall goal: if performance is not important but extreme protection is then all obfuscations should be applied, if it is important to minimize the program size then program-bloating obfuscations should be minimized or avoided altogether.

In order to best make these decisions we have included figures 8.3, 8.4, and 8.5 which graph each individual obfuscation's effect on benchmark complexity, size, and performance, respectively. In order to summarize the data and make it less visually dense an average, along with a low-high range, is given to represent all benchmarks instead of individual benchmark measurements.

From these figures we can derive default settings for the three most likely desired outcomes — high performance, small class file size, and full protection. We outline these settings in Table 8.2.

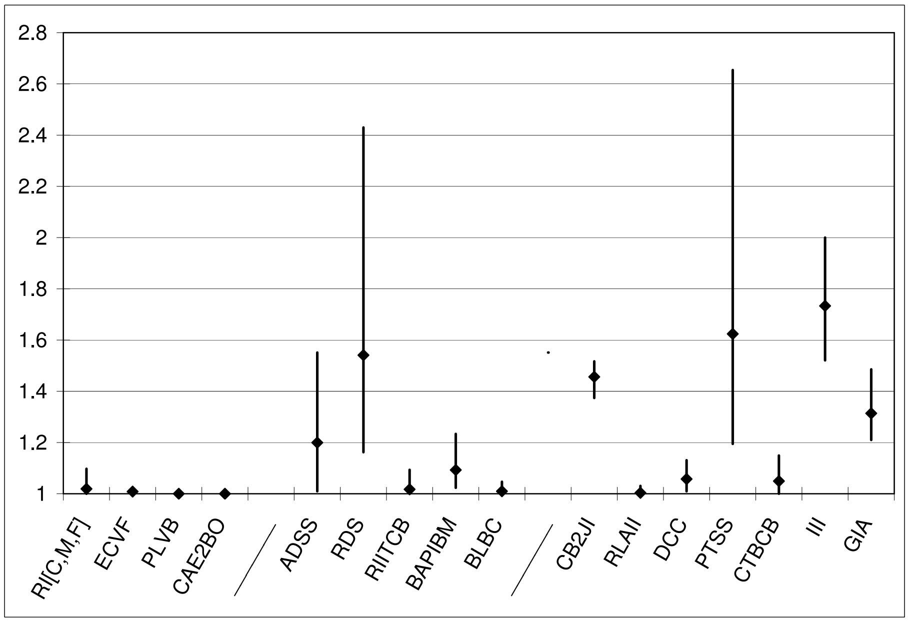


Figure 8.3: *Average complexity increase (control-flow graph edges + nodes) with high-low range. Values are aggregated over all benchmarks tested.*

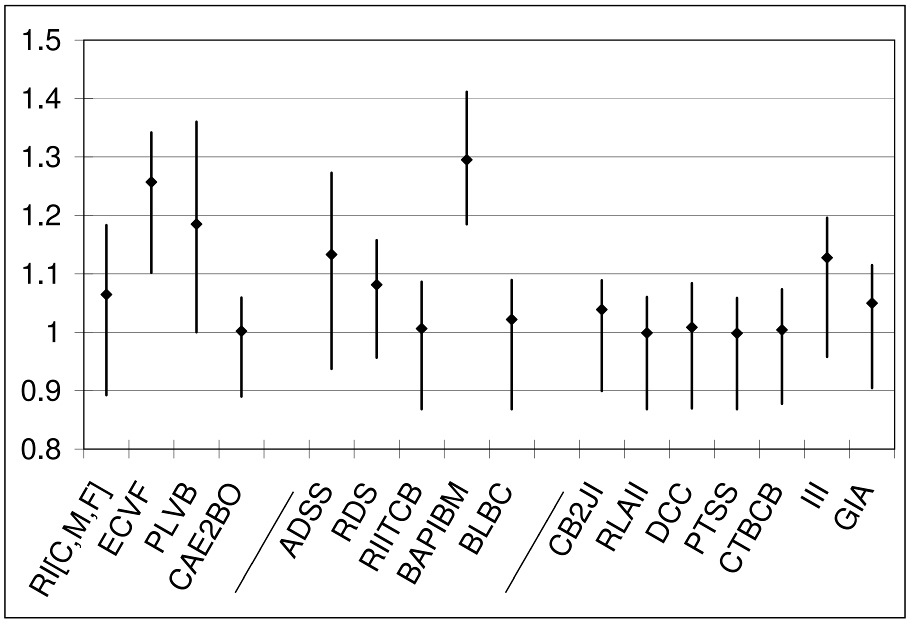


Figure 8.4: *Average class file size increase with high-low range. Values are aggregated over all benchmarks tested.*

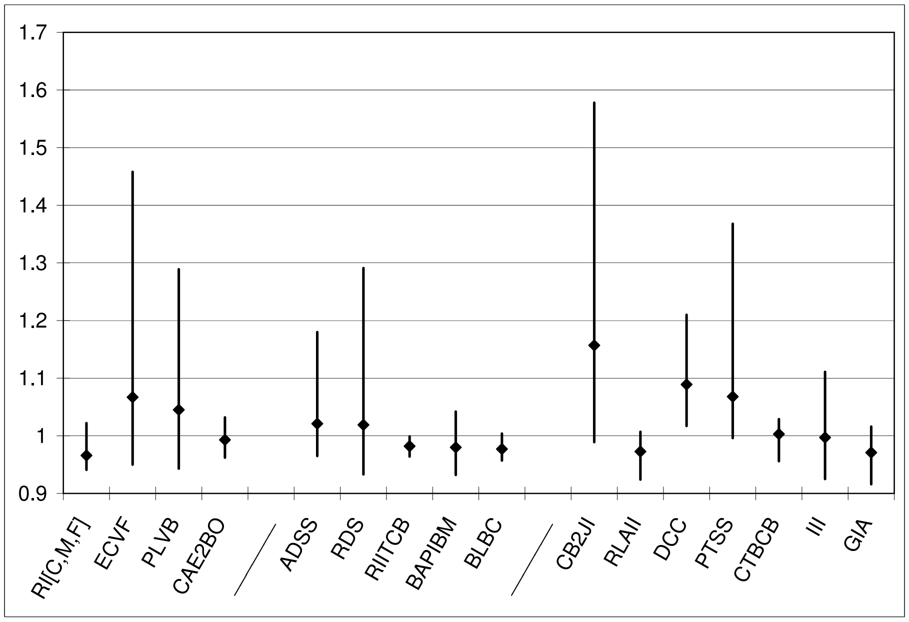


Figure 8.5: *Average performance degradation with high-low range (Server Mode). Values are aggregated over all benchmarks tested.*

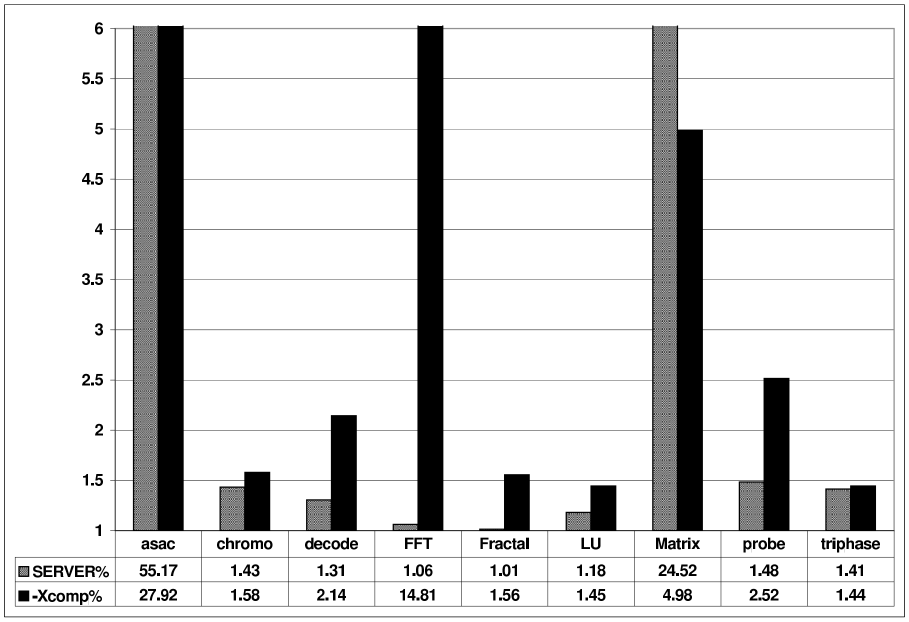
Table 8.2: *Weights used for obfuscations for three combinations*

Combination	Performance	Size	Protection
Renaming Identifiers: classes, fields and methods	9	0	5
Embedding Constant Values as Fields	0	0	0
Packing Local Variables into Bitfields	3	0	2
Converting Arithmetic Expressions to Bit-Shifting Ops	9	9	9
Adding Dead-Code Switch Statements	6	0	5
Finding and Reusing Duplicate Sequences	3	0	7
Replacing if Instructions with Try-Catch Blocks	9	9	9
Building API Buffer Methods	9	0	6
Building Library Buffer Classes	9	9	9
Converting Branches to jsr Instructions	0	9	9
Reordering loads Above if Instructions	9	9	9
Disobeying Constructor Conventions	0	9	5
Partially Trapping Switch Statements	0	9	9
Combining Try Blocks with their Catch Blocks	9	9	9
Indirecting if Instructions	6	0	9
Goto Instruction Augmentation	9	6	9

8.2.1 Performance

Speed of execution is the important factor for this combination and is most likely the best default. The user desires some obfuscation but requires that the program's performance is not significantly degraded. In this case we identified all transformations whose average performance loss was less than 5%. For those obfuscations which had a high range above 10% we applied them with a weight of 6 and for those over 20% we applied them with a weight of 3. These settings are presented in column 1 of Table 8.2.

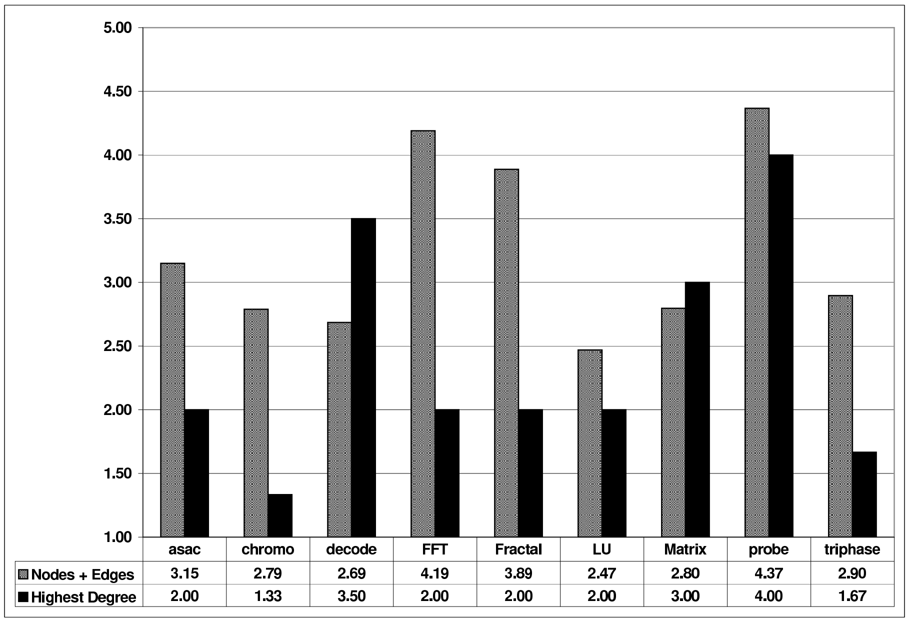
The performance degradation due to this combination, shown in Figure 8.6, is below 1.5 for most of the benchmarks but astronomically degrading for `asac` and `Matrix`. Brief investigation into the programs will quickly show that most often a single method is responsible for almost all of the performance slowdown. Some quick program profiling and a little tweaking of the JBCO settings (such as limiting the obfuscation performed on that one method) can easily reduce these performance issues to almost nothing while still maintaining a suitable level of obfuscation.



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	55.17	1.43	1.31	1.06	1.01	1.18	24.52	1.48	1.41
■ -Xcomp%	27.92	1.58	2.14	14.81	1.56	1.45	4.98	2.52	1.44

Figure 8.6: *Effects of applying the performance-minded obfuscation settings shown in Table 8.2 column 1. Default server-mode and full `-Xcomp` compilation are shown.*

Approaching the problem from a different angle, the user could apply the `-Xcomp` JVM flag when running the program. This flag, which enforces full JIT compilation of all methods, may slow the initial program startup time but in almost all of the cases during our research where obfuscations had serious performance penalties this setting was able to improve performance significantly, as the second data series in Figure 8.6 shows for `asac` and `Matrix`. This indicates that the obfuscator is doing its job! The methods are complex enough (see Figure 8.7) that the JIT's early optimization phases (which are most likely optimistic and simplistic compared to later phases) are unable to improve performance significantly.



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ Nodes + Edges	3.15	2.79	2.69	4.19	3.89	2.47	2.80	4.37	2.90
■ Highest Degree	2.00	1.33	3.50	2.00	2.00	2.00	3.00	4.00	1.67

Figure 8.7: *Performance-minded complexity ratio (total nodes + edges of performance-obfuscated program divided by total nodes + edges of original program and highest graph degree of obfuscated program divided by highest graph degree of original program).*

However, the full strength optimizations can have a negative effect. For short-running benchmarks such as `FFT`, `decode`, and `probe` the `-Xcomp` flag actually slows the execution down since the added time of optimization is more than the time required to execute the program normally with dynamic compilation. `FFT` is the worst hit as it has a normal running time of about 0.17 seconds. Full compilation, even for small programs, can add a few seconds or more to the VM startup time. This results in a very large slowdown for `FFT` because a few seconds is much longer than the original running time of the entire program.

Figure 8.8 shows the class file size increases significantly due to this combination of obfuscations. Much of the bloat can be attributed specifically to building API buffer methods (BAPIBM), since every library call will cause the creation of a new method. While the numbers are quite large, program size was the least important factor in defining this performance-minded combination. With the prevalence of high-speed Internet access, limiting program size is not as much of an issue as it once was.

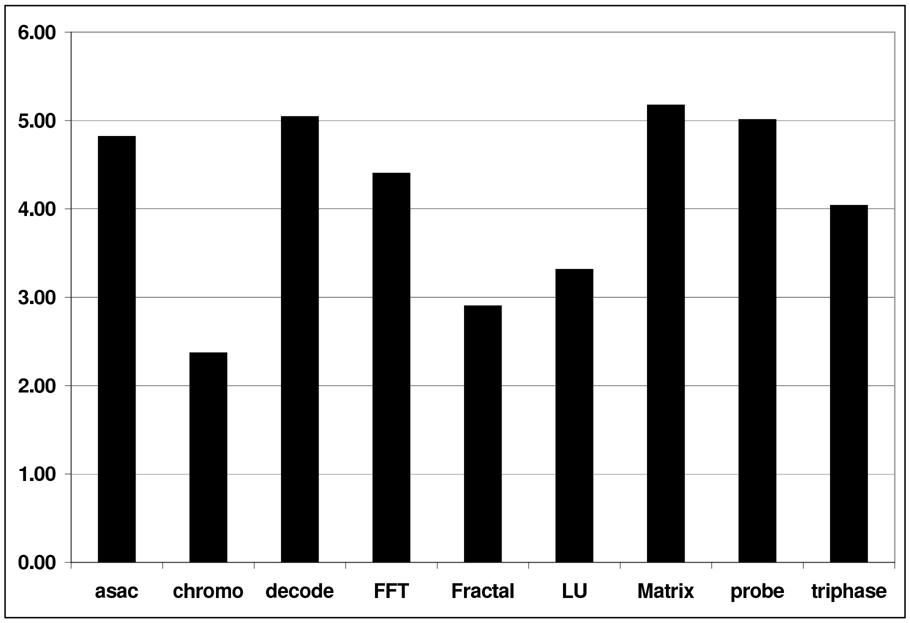
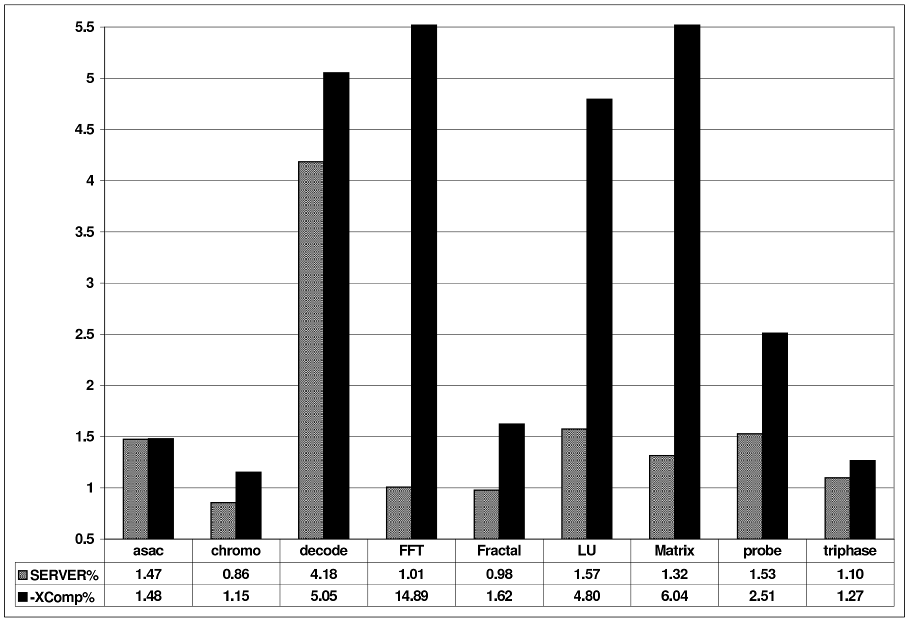


Figure 8.8: *Effects on class file size after applying the performance-minded obfuscation settings shown in Table 8.2 column 1.*

8.2.2 Size

This combination is most concerned with limiting the class file size blowup due to any obfuscations performed. This might be used for programs which are distributed over the Internet at every execution. For this combination we followed the formula in the performance description above except for size blowup per obfuscation. Obfuscations with averages below 5% were included. Obfuscations with a high range above 10% were assigned a weight of 6, for those over 20% we assigned a weight of 3. The settings are outlined in column 2 of Table 8.2.

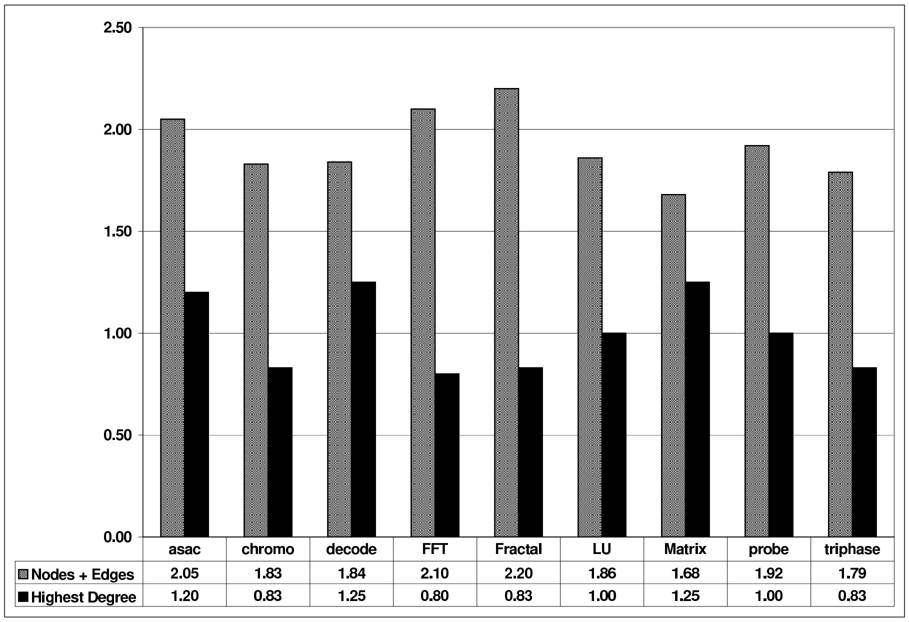
Not surprisingly, the limited number of obfuscations performed by this combination results in very little execution slowdown in most cases (see Figure 8.9). However, using the `-Xcomp` flag worsens the speed in all cases. This is most likely due to the fact that there isn't nearly as much complexity (as shown in Figure 8.10) and therefore there is little to be gained by using an aggressive compilation approach on such short running programs.



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.47	0.86	4.18	1.01	0.98	1.57	1.32	1.53	1.10
■ -XComp%	1.48	1.15	5.05	14.89	1.62	4.80	6.04	2.51	1.27

Figure 8.9: *Effects of applying the size-minded obfuscation settings shown in Table 8.2 column 2. Default server-mode and full `-Xcomp` compilation are shown.*

It is interesting to note that the highest degree of complexity actually drops for four of the benchmarks in Figure 8.10. While it is unclear why this is happening one must remember that some obfuscation actually move control flow branches to new units. The obfuscation (CB2JI) from Section 7.2, for example, can sometimes cause a single unit with two incoming branches to split into two units, each with one incoming branch. This can happen when a unit *u* is targeted by two different `goto` instructions. If one `goto` is replaced by a `jsr` a new `pop` is placed before *u* and a new `goto u` is placed before the



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ Nodes + Edges	2.05	1.83	1.84	2.10	2.20	1.86	1.68	1.92	1.79
■ Highest Degree	1.20	0.83	1.25	0.80	0.83	1.00	1.25	1.00	0.83

Figure 8.10: *Size-minded complexity ratio (total nodes + edges of size-obfuscated program divided by total nodes + edges of original program and highest graph degree of obfuscated program divided by highest graph degree of original program).*

`pop`. Now, *u* is not targeted by any unit except the newly created `goto`. One old `goto` instructions was replaced by a `jsr` pointing to the `pop` and the other `goto` now points to the *new* `goto u` above the `pop`. The new `goto u` is targeted by one branch, the `pop` is targeted by one branch, and the original unit *u* is targeted by one branch. There are no units targeted by more than one branch now, thereby potentially reducing the highest degree complexity of that control graph.

Comparing the class file size blowup of the previous combination in Figure 8.8 to that of the blowup from this combination in Figure 8.11 we can see that this combination is fairly successful at limiting the amount of program bloat. However, the average blowup is still in the range of 250%. This might not be acceptable for very large programs expected to be distributed over a network.

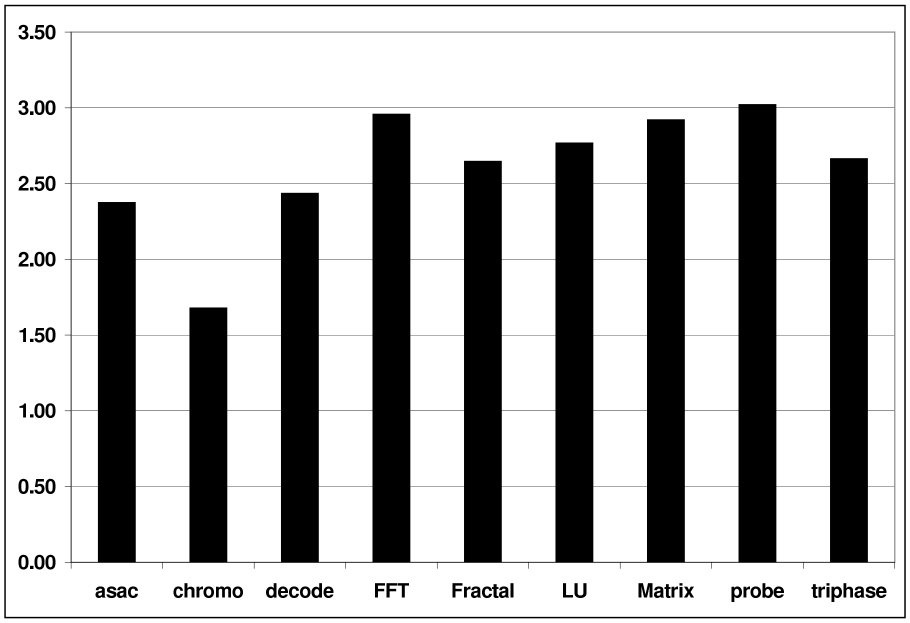


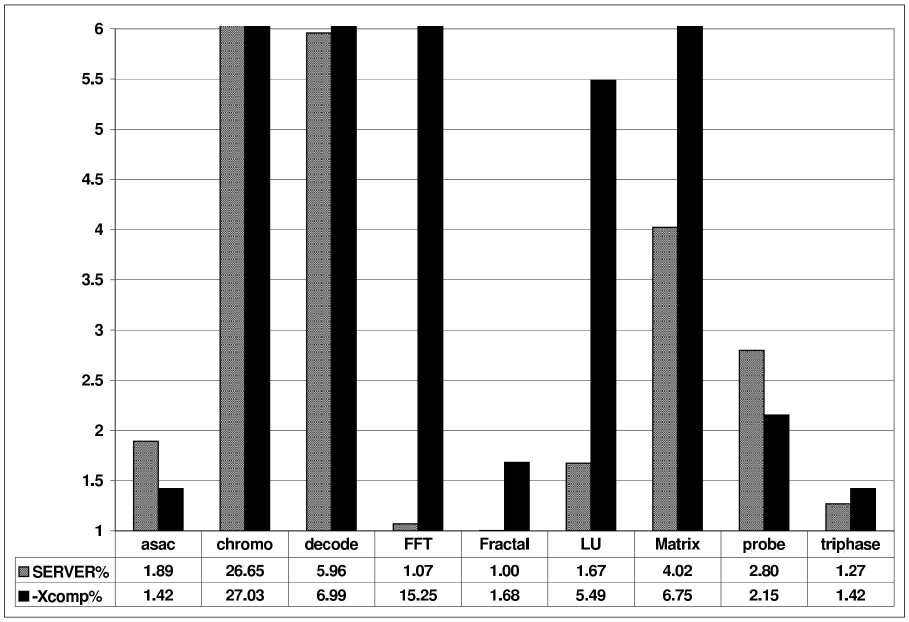
Figure 8.11: *Effects on class file size after applying the size-minded obfuscation settings shown in Table 8.2 column 2.*

8.2.3 Protection

This combination is meant to provide the most possible protection through obfuscation. Performance and size are the much lesser concerns compared to protecting the intellectual property contained within the code base. For each obfuscation we averaged the weights used in the performance and size combinations above and added the percentage complexity the obfuscation was able to add, on average. The settings are shown in column 3 of Table 8.2.

Surprisingly, the execution speeds of the benchmarks after this combination is applied, shown in Figure 8.12, is overall not as bad as the performance-minded combination. This clearly shows that the combination and order of obfuscations can have a very large impact on performance. Specifically, `asac` does not perform that badly whereas it showed a staggering 5500% slowdown after the performance-minded combination of obfuscations.

Figure 8.13 shows that this combination is able to significantly increase complexity, but the numbers are somewhat similar to those of the performance-minded obfuscation

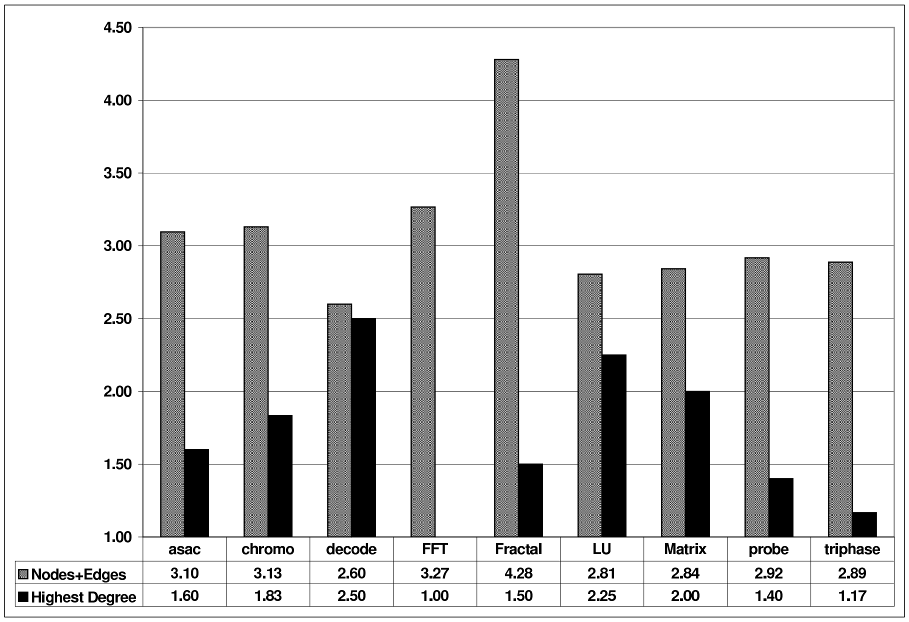


	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ SERVER%	1.89	26.65	5.96	1.07	1.00	1.67	4.02	2.80	1.27
■ -Xcomp%	1.42	27.03	6.99	15.25	1.68	5.49	6.75	2.15	1.42

Figure 8.12: *Effects of applying the protection-minded obfuscation settings shown in Table 8.2 column 3. Default server-mode and full `-Xcomp` compilation are shown.*

complexity (see Figure 8.7). This means the combination is working as it should by providing a large jump in method complexity, but it is unclear if the performance-minded and protection-minded combinations provide two unique sets of options to the user. Even the class file size increases due to this combination, as seen in Figure 8.14 are very similar to those increases due to the performance-minded combination, shown in Figure 8.8.

These are very surprising results because, overall, the protection-minded combination is applying more obfuscations and with heavier weights. More clear differences between the performance-minded combination results would normally be expected. However, the flaw of JBCO is that its weighting mechanism is not deterministic. It uses a random number generator to decide when to apply an obfuscation based on its weight. It is entirely possible, within the probabilities of a random number generator, that an obfuscation with a weight of 6 would, in fact, never be applied. A more stable approach would require an extra pass through the code for each obfuscation, at which point statistics could be collected to report the number of obfuscation sites available. JBCO could then randomly select from these sites to ensure that, if the weight for an obfuscation was 6 then 60% to 70% of the sites



	asac	chromo	decode	FFT	Fractal	LU	Matrix	probe	triphase
▨ Nodes+Edges	3.10	3.13	2.60	3.27	4.28	2.81	2.84	2.92	2.89
■ Highest Degree	1.60	1.83	2.50	1.00	1.50	2.25	2.00	1.40	1.17

Figure 8.13: *Protection-minded complexity ratio (total nodes + edges of size-obfuscated program divided by total nodes + edges of original program and highest graph degree of obfuscated program divided by highest graph degree of original program).*

would get obfuscated.

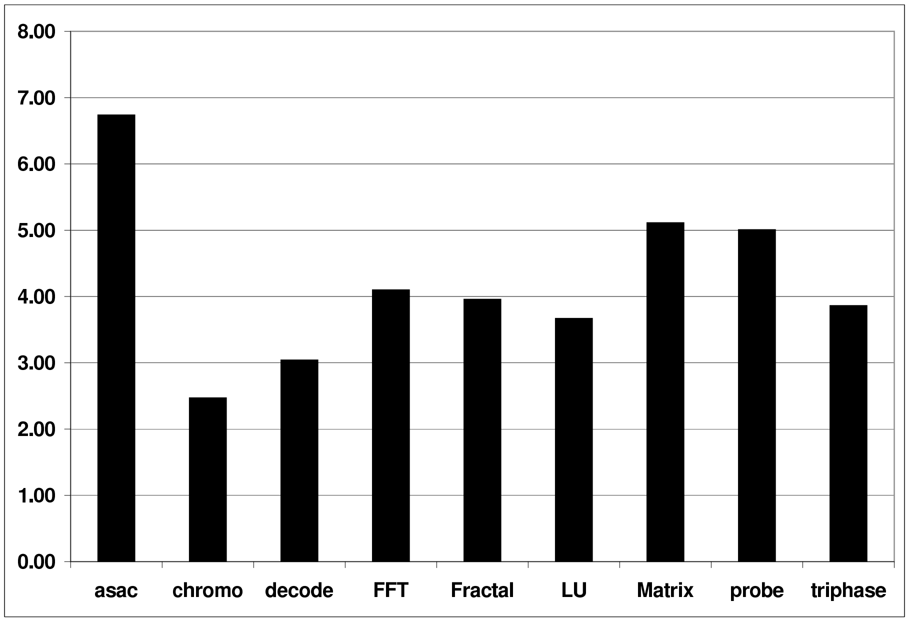


Figure 8.14: *Effects on class file size after applying the protection-minded obfuscation settings shown in Table 8.2 column 3.*

Chapter 9
Conclusions

We have presented a number of obfuscating transformations with an automatic tool called JBCO. While we have shown it to be effective at protecting compiled Java bytecode against modern decompilers, there are nevertheless opportunities for improvement.

9.1 Future Work

While we have implemented sixteen different obfuscation in JBCO there will always be room for more. There are still areas of interest within bytecode and the JVM which we did not have time to explore, such as the thread locking mechanisms of Java.

There are problems with JBCO, as well. The current limitations of JBCO are discussed in Section 8.2 and graphs are given which explain these problems. The most important issues are the excessive performance degradation and class file size bloat often seen after a combination of obfuscations has been applied.

9.1.1 New Obfuscations

While JBCO is adept at breaking decompilers and is very effective at transforming code into difficult to read gibberish, it still has room for improvement.

Many simple obfuscations such as embedding constant values in fields (ECVF), packing local variables into bitfields (PLVB) and converting arithmetic expressions to bit-shifting operations (CAE2BO) could be further obscured with the addition of opaque predicates. Furthermore, these same obfuscation sites could be used to insert dead code (false-branch) opaque predicate value changes.

JBCO has been designed with a very modular approach. Each obfuscation exists as its own phase within Soot. While this is useful in some ways and allows for selective obfuscation, often times combining multiple obfuscation techniques at the same time can yield better results. By making JBCO open source we are publishing the exact details of each obfuscation. Someone wishing to improve upon existing decompilers could inspect our code to develop more powerful reverse-engineering techniques. Because of this, combining multiple obfuscation techniques at the same time has the potential to provide more protection than our current approach since any given obfuscation site will have multiple transformations applied to it. Within the modular framework, it is possible that some areas of code could be obfuscated by one transformation and another distinct section of code be obfuscated by a different transformation. Without the combinatory power of these obfuscations, they are much easier to recognize and therefore easier to reverse.

Above and beyond the existing techniques in JBCO, there will always be room for more. Java's built-in thread locking mechanism is one area with untapped obfuscation potential. Java uses monitors to provide object locking and these are controlled through the `monitorenter` and `monitorexit` instructions. While some bytecode verification is performed with respect to these monitor ranges, not all locking situations can be statically ensured for safety. Therefore, the JVM will throw a special `IllegalMonitorState-Exception` dynamically during runtime if and when an illegal situation arises. Because of this duality that exists between the static and dynamic safety mechanisms, there is prime opportunity for obfuscation.

Additionally, while the library buffer classes and methods provide some object-oriented design obfuscation, there is still much potential. The Java interface, which provides a framework for object polymorphism, has not been exploited by any obfuscations in JBCO. This has been explored in the literature with some success by Sakabe, *et al.* [SSM03] and Sosonkin, *et al.* [SNM03] and is a possible avenue for future research.

9.1.2 Improvements to Performance

It has been noted in Section 4.3 that as much as 90% or more of the execution time of a program can usually be attributed to 10% or less of the actual static code. This is because the real work of a program is often contained within one method which is called over and over again. This method, which we call a “hot” method, will be particularly susceptible to slowdowns caused by obfuscation.

One possible approach to avoiding these slowdowns could be to develop various intra-procedural analyses within JBCO itself that detect loops, recursive calls, and object creation. JBCO could limit the obfuscations applied to these parts of the code which could potentially improve performance. However, many of the obfuscations in JBCO rely on these very constructs as obfuscations sites. It is questionable how much improvement could be obtained through this technique.

Another possible approach to avoiding this problem is to develop static techniques for identifying these hot sections of code. If JBCO was able to make some conservative estimate of how often a piece of code is likely to be executed then it could limit the number of obfuscations performed on that section.

This static identification could be accomplished by performing statistics analysis of profiling data and CPU sampling provided by the user. JBCO could then avoid those methods which take up most of the running time of the program. This technique can only ever have limited success on programs which are not deterministic. While one profiling run might cause the execution of one method over and over again, another profiling run might result in not a single call to that method, depending on input and other factors.

Hot methods in the code might also be identified by performing inter-procedural analysis within JBCO. Soot is already outfitted with inter-procedural analysis frameworks and can build call graphs. Unfortunately, this kind of analysis is on the forefront of compiler technology and might have a long way to go before it becomes truly useful. This approach would likely suffer as much, if not more, than the profiling approach due to the non-deterministic nature of most programs.

One final approach to limiting execution slowdown due to obfuscation could be to implement an internal JIT within the program itself which performs profiling. Two or more copies of each method could be maintained with various levels of obfuscation and, depending on how often the method is called, the JIT could choose to use a faster (less obfuscated) version or a more protected version. This would cause severe increases in the class file sizes but could potentially add even more obfuscation through the use of multiple versions of the same logic. However, the weakest link would always be the least obfuscated version of the logic.

9.2 Conclusions

Decompiler technology is powerful and avenues of protection for intellectual property must be developed. Obfuscation is one such avenue. This thesis introduced a number of obfuscations that can be applied to Java bytecode in order to hinder reverse-engineering and protect against automatic decompilers such as `Dava` and `SourceAgain`.

The obfuscations we detailed in chapters five through seven were implemented in an automatic tool called JBCO. We evaluated each technique individually in order to test the effect it had on program execution time, program size, and program complexity. The obfuscated code for each transformation was also decompiled in order to see how well it was able to garble code and confuse decompilers.

Many of these techniques are unique and do not exist in the literature. By concentrating our efforts on obfuscations which exploit the gap between the Java source language and its compiled bytecode representation, we were able to cause well known modern decompilers to produce incorrect and incomplete source code. Often times we were even able to cause these decompilers to crash or report errors.

Finally, we used our experimental results to develop three different combinations of obfuscations for three different scenarios. In the first, program performance is the most important concern and the obfuscations chosen to be included in the combination had limited performance impact. In the second, program size is the greatest concern. Only obfuscations with limited effect on size were selected. Finally, the third scenario concerned itself with maximum protection. In this scenario all but one obfuscation was included in the combination. These three different combinations were implemented as default options in the JBCO graphical user interface.

JBCO is a robust and stable obfuscator with unique obfuscation approaches that can not be found elsewhere. It is very flexible, allowing the user full control over which obfuscations are performed and on which methods, fields, and classes. While there is still much work that can be done, as outlined in Section 9.1, we feel that this initial version of JBCO is already quite useful and compares positively against existing obfuscators.

Bibliography

[App02] Andrew W. Appel. Deobfuscation is in NP, Aug. 21 2002. <citeseer.ist.psu.edu/553532.html> .

[BGB90] 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.

[BGI$^+$01] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. *Lecture Notes in Computer Science*, 2139:1–??, 2001.

[CK94] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[Cog01] Alessandro Coglio. Improving the official specification of Java bytecode verification. In *Proc. 3rd ECOOP Workshop on Formal Techniques for Java Programs*, Jun. 2001.

[Coh93] Frederick B. Cohen. Operating system protection through program evolution. *Comput. Secur.*, 12(6):565–584, 1993.

[Coh97] R. Cohen. The defensive Java virtual machine specication. Technical report, Computational Logic Inc., 1997.

[CT02] Christian S. Collberg and Clark Thomborson. Watermarking, tamper-proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, Aug. 2002, volume 28, pages 735–746.

[CTL98] Christian Collberg, Clark Thomborson, and Douglas Low. Breaking abstractions and unstructuring data structures. In *ICCL '98: Proceedings of the 1998 International Conference on Computer Languages*, 1998, page 28. IEEE Computer Society, Washington, DC, USA.

[FSA97] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building diverse computer systems. In *Workshop on Hot Topics in Operating Systems*, 1997, pages 67–72.

[GCT05] Jun Ge, Soma Chaudhuri, and Akhilesh Tyagi. Control flow based obfuscation. In *DRM '05: Proceedings of the 5th ACM workshop on Digital rights management*, Alexandria, VA, USA, 2005, pages 83–92. ACM Press, New York, NY, USA.

[GHM00] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for Java bytecode. In *Static Analysis Symposium*, 2000, pages 199–219.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. Addison Wesley, 2000.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[Gro] Trust Computing Group. Trusted computing group frequently asked questions. Available on: `https://www.trustedcomputinggroup.org/faq/CompleteFAQ/`.

[GVG04] Dayong Gu, Clark Verbrugge, and Etienne Gagnon. Code layout as a source of noise in JVM performance. In *Component And Middleware Performance workshop, OOPSLA 2004*, Vancouver, BC, Canada, 2004.

[GVG06] Dayong Gu, Clark Verbrugge, and Etienne M. Gagnon. Relative factors in performance analysis of Java virtual machines. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, Ottawa, Ontario, Canada, 2006, pages 111–121. ACM Press.

[HK81] S Henry and K Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[Jad] Jad - the fast JAva Decompiler. Available on: `http://www.geocities.com/SiliconValley/Bridge/8617/jad.html`.

[KBT05] Tom Kalibera, Lubomir Bulej, and Petr Tuma. Benchmark precision and random initial state. In *Proceedings of the 2005 International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS 2005), San Diego, CA, USA, July 26-29, 2004.*, Jul. 26–29, 2005, pages 484–490. SCS.

[Ker83a] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sceiences militaires*, IX:5–38, Jan. 1883.

[Ker83b] Auguste Kerckhoffs. La cryptographie militaire. *Journal des sceiences militaires*, IX:161–191, Feb. 1883.

[Ler01] Xavier Leroy. Java bytecode verification: An overview. *Lecture Notes in Computer Science*, 2102:265+, 2001.

[Mic01] SUN Microsystems. The Java HotSpot Virtual Machine, 2001. Technical White Paper.

[MK93] John C. Munson and Taghi M. Khoshgoftaar. Measurement of data structure complexity. *J. Syst. Softw.*, 20(3):217–225, 1993.

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

[MT06] Anirban Majumdar and Clark Thomborson. Manufacturing opaque predicates in distributed systems for code obfuscation. In Vladimir Estivill-Castro and Gillian Dobbie, editors, *Twenty-Ninth Australasian Computer Science Conference (ACSC 2006)*, 2006, volume 48 of *CRPIT*, pages 187–196. ACS, Hobart, Australia.

[MTG^{+}99] M. Merten, A. Trick, C. George, J. Gyllenhaal, and W. Hwu. A hardware-driven profiling scheme for identifying program hot spots to support runtime optimization. In Doug DeGroot, editor, *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, May 1–5 1999, volume 27, 2 of *Computer Architecture News*, pages 136–149. ACM Press, New York, N.Y.

[NH06] Nomair A. Naeem and Laurie Hendren. Programmer-friendly decompiled Java. In *Proceedings of the 14th IEEE International Conference on Program Comprhension*, Athens, Greece, 2006.

[PV05] Christopher J. F. Pickett and Clark Verbrugge. Software thread level speculation for the Java language and virtual machine environment. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC'05)*, Oct. 2005.

[Ray04] Eric Raymond. Email correspondence: If Cisco ignored Kerckhoffs' law, users will pay the price. Available on: `http://lwn.net/Articles/85958/`, May 2004.

[SNM03] Mikhail Sosonkin, Gleb Naumovich, and Nasir Memon. Obfuscation of design intent in object-oriented applications. In *DRM '03: Proceedings of the 3rd ACM workshop on Digital rights management*, Washington, DC, USA, 2003, pages 142–153. ACM Press, New York, NY, USA.

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

[SSM03] Yusuke Sakabe, Masakazu Soshi, and Atsuko Miyaji. Java obfuscation with a theoretical basis for building secure mobile agents. In *Communications and Multimedia Security*, 2003, pages 89–103.

[Str97] Bjarne Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[swr01] staff and wire reports. How Soviets copied America's best bomber during WWII. *CNN.com*, Jan. 2001.

[Uni] Title 17, United States Code, copyrights. Available on: `http://www.copyright.gov/title17/`.

[VRHS$^+$99] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, 1999, pages 125–135.

[WHKD00] Chenxi Wang, Jonathan Hill, John Knight, and Jack Davidson. Software tamper resistance: Obstructing static analysis of programs. Technical Report CS-2000-12, University of Virginia, Dec. 2000.