An Empirical Evaluation of Java Decompilation and Intellectual Property Protection via Software Watermarking

MPhil Transfer Report

James Hamilton
Supervisor: Sebastian Danicic

Goldsmiths, University of London
United Kingdom

Abstract

Decompilation of Java bytecode is the act of transforming Java bytecode to Java source code. Although easier than that of decompilation of machine code, problems still arise in Java bytecode decompilation. These include type inference of local variables and exception-handling. We evaluate the currently available Java bytecode decompilers using an extension of the criteria used in a previous original study. Although there has been a slight improvement since this study, it was found that none passed all of the tests, each of which were designed to target different problem areas.

Decompilation and other attacks on software are problems for the software industry, with the global revenue loss due to software piracy estimated to be more than $50 billion in 2008. We present a survey of a specific software protection techniques - software watermarking - in the context of Java decompilers.

Software watermarks can be used to prove ownership of stolen software. However, many watermarks are easily removed rendering their protection useless.

We evaluate static watermarking techniques, highlight their failings and present directions for future work.

List of Publications

Refereed Conference Papers

2010 **An Evaluation of Static Java Bytecode Watermarking** (James Hamilton, Sebastian Danicic), *In Proceedings of the International Conference on Computer Science and Applications (ICCSA'10)*, 2010.

Refereed Workshop Papers

2009 **An Evaluation of Current Java Bytecode Decompilers** (James Hamilton, Sebastian Danicic), *In Ninth IEEE International Workshop on Source Code Analysis and Manipulation*, IEEE Computer Society, volume 0, 2009.

Contents

List of Figures

Listings

Chapter 1

Introduction

Programmers write applications in a high-level language like Java or C which is understandable to them but which cannot be executed by the computer. The textual form of a computer program, known as source code, is converted into a form that the computer can directly execute. Java source code is compiled into an intermediate language known as Java bytecode which is not directly executed by the CPU but executed by a virtual machine. Programmers need not understand Java byte code but doing so can help debug, improve performance and memory usage [72].

Compilation is the act of transforming a high-level language, into a low-level language such as machine code or bytecode. Decompilation is the reverse. It is the act of transforming a low-level language into a high-level language [19].

Decompilation can be used to assist software theft, also known as software piracy - the act of copying a legitimate application and illegally distributing that software, either free or for profit. There are many methods to protect software including legal methods such as copyright laws, patents and license agreements but these do not always dissaude people from stealing software, especially in emerging markets where the price of software is high and incomes are low. Ethical arguments, such as fair compensation for producers, by software manufacturers, law enforcement agencies and industry lobbyists also do little to counter software piracy. The global revenue loss due to software piracy was estimated to be more than $50 billion[1] in 2008 [17]. We therefore require technical measures to reduce software piracy, such as software watermarking and/or obfuscation.

Software watermarking involves embedding a unique identifier within a piece of software, to discourage software theft. Watermarking does not prevent theft but instead discourages software thieves by providing a means to identify the owner of a piece of software and/or the origin of the stolen software. The hidden watermark can be extracted, at a later date, by the use of a recogniser to prove ownership of stolen software.

[1] This figure is said to be inflated, by some, as the report's conclusions are based on mathematical models not empirical data revealing specific instances of piracy [137, 111]

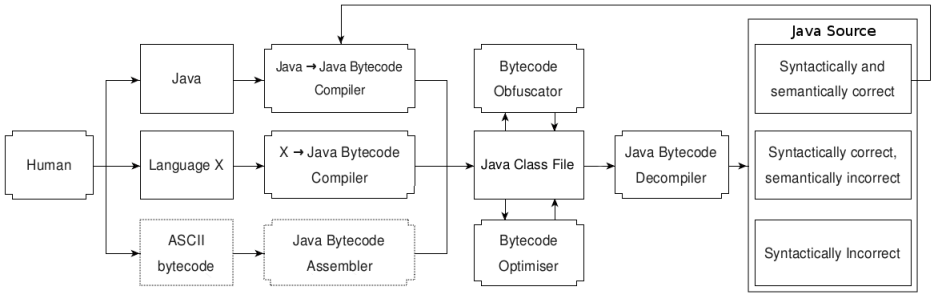


Figure 1.1: Compilation and decompilation of Java Bytecode

1.1 Background

1.1.1 Java and the Java Virtual Machine

The Java Virtual Machine is essentially a simple stack based machine which can be separated into five parts: the heap, program counter registers, method area, Java stacks and native method stacks [168] . The Java Virtual Machine Specification [108] defines the required behaviour of a Java virtual machine but does not specify any implementation details. Therefore the implementation of the Java Virtual Machine Specification can be designed different ways for different platforms as long as it adheres to the specification. A Java Virtual Machine executes Java bytecode in class files conforming to the class file specification which is part of the Java Virtual Machine Specification [108] and updated for Java 1.6 in JSR202 [16].

An advantage of the virtual machine architecture is portability - any machine that implements the Java Virtual Machine Specification [108] is able to execute Java bytecode hence the slogan "Write once, run anywhere" [97]. Java bytecode is not strictly linked to the Java language and there are many compilers, and other tools, available which produce Java bytecode [**?** 68] such as the Jikes compiler, Eclipse Java Development Tools or Jasmin bytecode assembler. Another advantage of the Java Virtual Machine is the runtime type-safefy of programs. These two main advantages are properties of the Java Virtual Machine not the Java language [68] which, combined, provides an attractive platform for other languages.

Java bytecode can be generated in three ways:

1. from a Java source program using a Java compiler (such as Sun's *javac*),
2. using a language other than Java to Java bytecode compiler (such as JGNAT [3] - an open-source Ada to Java bytecode compiler) or
3. by writing a class file by hand.

The tedious task of hand-writing a Java class file can be made easier by using a Java assembler, such as Jasmin [113], which accepts a human readable form of Java bytecode instructions and generates a Java class file. We use Jasmin source in this paper for examples programs which are written manually, or for examples which require inspection of the bytecode.

Java bytecode can also be manipulated by tools such as obfuscators and optimisers which perform semantics preserving transformations on bytecode contained within a Java class file. Figure **??** shows the Java bytecode cycle from generation to decompilation to Java source.

Java bytecode retains type information about fields, method returns and parameters but it does not, for example, contain type information for local variables. The type information in the Java class file renders the task of decompilation of bytecode easier than decompilation of machine code [54]. Decompiling Java bytecode, thus, requires analysis of most local variable types, flattening of stack-based instructions and structuring of loops and conditionals. The task of bytecode decompilation, however, is much harder than compilation. We show that often decompilers cannot fully perform their intended function [23, 74].

1.1.2 Decompilation

The problems to be solved by a general decompiler can be divided into different categories [54]:

1. separation of code from data
2. separation of pointers from constants
3. separation of original and offset pointers
4. declaration of data
5. recovery of parameters and returns
6. analysis of indirect jumps and calls
7. type analysis
8. merging of instructions
9. structure of loops and conditionals

The decompilation of machine code requires all 9 of these tasks to be solved whereas the decompilation of Java bytecode requires only 3 of these tasks to be solved due to the amount of information stored in a class file. A fourth problem caused by exceptions and synchronisation stems from their implementation using arbitrary control flow and possibly overlapping exception handlers.

Decompiling Java bytecode requires analysis of most local variable types, merging of stack-based instructions and structuring of loops and conditionals. Java bytecode retains type information for fields, method returns and parameters but it does not contain type information for local variables. This information encoded in the class file makes the task of type inference easier compared to decompilation of machine code.

Decompilation has many applications including legitimate uses, such as the recovery of lost source code for a crucial application [56] and non-legitimate uses such as reverse-engineering a proprietary application. Consider the case in which a company has lost the source code for their application and to continue development on the software they require recovery of source code from Java class files. The company must decompile the Java class files and attempt to recover Java source equivalent to the originally lost source. In this case, in comparison to an illegitimate use, it is likely that the company knows more about how the Java class files were generated. Knowledge of how class files are generated provides information useful in the recovery of the original source as a decompiler can be optimised for the compiler used.

The purpose of a decompilation task indicates the level of decompilation to be achieved. Recompilation of a decompiled program does not require that the source be easy to read or tidy. It does not even require that type inference is performed as long as the program is correctly type-casted. Many of the existing Java decompilers do not perform type inference [55] and, while much research has gone into creating efficient type inference algorithms [63, 13], code with correct type-casts is semantically equivalent.

On the other hand, if the purpose of decompilation is to produce maintainable, extensible code then readable code with full typing is desirable. Such a decompiler should produce code that is similar to a human programmer so that future work on the code can be undertaken easily. Some decompilers produce output which is more readable than others, while optimisations have been been to some decompilers to produce more readable code [128, 127].

If the purpose of decompilation is to simply understand a program, the syntactical correctness of a complete decompiled program may not be a high-priority. Correct portions of an incorrect program could help in the understanding of a program, in contrast to the case of source recovery where correct source is needed. Software theives would be able to analyse partial Java programs [42] allowing them to remove, for example, product key validation code.

There is a fair amount of literature on decompilation but not a lot covering specifically Java decompilation. A lot of interesting research in this subject and generally Java optimisation is performed by

the Sable Research Group[2] at McGill University. They have created Soot [164] - a Java optimisation framework which includes a Java decompiler called Dava [116]. Several commercial decompilers exists (e.g. Class Cracker) thought most have been unmaintained for several years.

The design of a decompiler is made easier if it only has to decompile code produced by Sun's *javac* Java compiler as it mostly means inverting a known compilation strategy [116]. *javac* is open-source so developers can see the implementation of the compiler and relatively easily invert it; of course there are still some problems to overcome. Many of the existing decompilers expect classfiles generated by *javac*.

The Java bytecode Verifier and, from version 1.6 onwards, the Java bytecode Checker check the validilty of Java bytecode as a matter of program security. This bytecode validation phase ensures that programs are 'well-behaved'. The verification process does not require that a program is compiled with *javac* and it is possible to create class files which no Java compiler can produce yet they pass the Verifier with flying colours [100]. This has caused security concerns since the early days of Java.

The decompilation of arbitrary, verifiable bytecode is more difficult than that of decompiling *javac* produced bytecode due the ability to create or change class files in ways that are able to pass verification but do not contain expected bytecode. The decompilation of such arbitary bytecode causes many Java decompilers to fail which prompted the development of Dava [116] - a decompiler designed to deal with arbitrary bytecode.

As an example *javac* produces bytecode which leaves the stack height the same before and after any statement. This is not a requirement of the Verifier and any classfiles which do not follow this break certain decompilers [54]. Some of the early decompilers are fooled by the insertion of a *pop* opcode at the end of a method as this is unexpected even though it passes the Java Verifier.

One source of the problem of arbitrary bytecode is that the class file format is entirely independent of the Java language and bytecode is more powerful than the Java language [100]. For example in the Java Virtual Machine specification [108] in relation to exception handling it lists the conditions with which *javac* produces exception handling bytecode but it notes that there are no restrictions enforced by the Verifier to check these conditions and suggests this does not pose a threat to the integrity of the Java Virtual Machine. Therefore unexpected exception handling code can be written and still pass the verification process.

Tools such as bytecode optimisers and code obfuscation change bytecode which can result in verificable bytecode which doesn't easily translate to syntactically correct Java source code.

1.1.3 Software Watermarking

Software watermarking involves embedding a unique identifier within a piece of software, to discourage software theft. Watermarking does not prevent theft but instead discourages software thieves by providing a means to identify the owner of a piece of software and/or the origin of the stolen software [123]. The hidden watermark can be extracted, at a later date, by the use of a recogniser to prove ownership of stolen software. It is also possible to embed a unique customer identifier in each copy of the software distributed which allows the software company to identify the individual that pirated the software. It is necessary that the watermark is hidden so that it cannot be detected and removed. It is also necessary, in most cases, that the watermark is robust - that is, resilient to semantics preserving transformations (such as optimisations or obfuscations). However, in some cases it is desirable that a watermark is fragile in the sense that if semantics preserving transformations are performed on the software the watermark becomes invalid. This is useful in the context of software licensing where any changes to a program could disable it.

Watermarking techniques are used extensively in the entertainment industry to identify multimedia files such as audio and video files, and the concept has extended into the software industry. Watermarking does not aim to make a program hard to steal or indecipherable like obfuscation but it discourages theft as thieves know that they could be identified.

Most literature discusses techniques and problems of automatically watermarking software and there is only a small amount of literature which compares automatic watermarking with manual watermarking [131]. A manual watermark is inserted by the programmer of the application, rather than a using a third-party automatic tool. Some semi-automatic watermarking systems also exist (e.g. The Collberg-Thomborson algorithm implemented in Sandmark[26]) - where a programmer inserts markers into a

[2] `http://www.sable.mcgill.ca/`

program during development and the finished software is then augmented by a software watermarking tool.

Watermarks can be classified as either visible, where the recongiser is public knowledge or invisible where the recongniser, or some component (such as an encryption key), is not public knowledge. Visible watermarks can act as a deterrent but also show an adversary the location of a watermark making the task of removing the watermark easier.

Difficulties of Software Watermarking

Software watermarks present several implementation problems and many of the current watermarking algorithms are vulnerable to attack. Watermarked software must meet the following conditions:

program size must not be increased significantly. program efficiency must not be decreased significantly. robust watermarks must be resilient to semantics preserving transformations (fragile watermarks, by definition, should not be). watermarks must be sufficiently well hidden, to avoid removal. watermarks must be easy for the software owner to extract. Perhaps the most difficult problem to solve is keeping the watermark hidden from attackers while, at the same time, allowing the software owner to efficiently extract the watermark when needed. If the watermark is too easy to extract then an attacker would be able to extract the watermark too. If a watermark is too well hidden then the software owner may not be able to find the watermark, in order to extract it. Some watermark tools (such as Sandmark [26]) use markers to designate the position of the stored watermark - this is problematic as it poses a risk of exposing the watermark to an adversary.

Watermarks should be resilient to semantics preserving transformations and ideally it should be possible to recognise a watermark from a partial program. Semantics preserving transformations, by definition, result in programs which are syntactically different from the original, but whose behaviour is the same. The attacker can attempt, by performing such transformations, to produce a semantically equivalent program with the watermark removed. Redundancy and recognition with a probability threshold may help with these problems [117].

The watermark code must be locally indistinguishable from the rest of the program so that it is hidden from adversaries [167]. For example, imagine a watermark which consists of a dummy method with 100 variables - this kind of method will probably stand out in a simple analysis of the software (such as using software metrics techniques [73, 90]). It could be difficult to programatically generate code which is indecipherable from the human-generated program code but statisical analysis of the original program could help in generating suitable watermarks [117].

Ideally, software watermarks should be resilient to decompilation-recompilation attacks, as decompilation of Java is possible (though not perfect [74]).

Software watermarks must be efficient in several ways:

- cost of embedding time.
- cost of runtime.
- cost of recognition time.

The cost of embedding a software watermark can be divided into two areas: developer time and embedding cost. The former simply quantifies the time that a developer spends embedding a watermark, while the latter quantifies the execution time of a software watermarking tool. Embedding costs are not a significant problem except in certain cases such as live multimedia streaming.

Developer time is important in use of software watermarks as the developer should not have to spend a large amount of time preparing a software watermark. The complexity of a software watermark is proportional to the resilience of the watermark - that is, the greater amount of time a developer spends embedding a watermark the harder it may be for an adversary to crack. For example, a developer could spend days introducing a subtle semantic property into the program which is unique to the software and very hard to discover. Consider a program which when a certain key combination is entered one pixel in the program's interface changes to a certain colour - this would be hard for an attacker to detect but may take a developer time to implement.

Now consider an automatic watermarker which programatically generates dummy code and injects this into random places within the program - such code will be easier to discover than code that was generated by a developer and carefully inserted.

In the middle of the scale is a semi-automatic watermark which involves a developer preparing a program before a watermarking tool embeds the watermark. The preparations could include inserting markers where watermark code should be inserted, or creating dummy methods which watermarks could use. Monden et al. [121] describe a watermarking algorithm which requires the production of a dummy method in a program for the watermark to be stored. A programmer must create this dummy method manually and then execute watermarking software to embed the watermark.

The cost of runtime depends on the effect that the transformations applied by the watermark have had on the size and execution time. For example, Hattanda *et al.* [78] found that the size of a program, watermarked with Davidson/Myhrvold [44] algorithm, increased by up to 24% and the performance decreased by up to 14%.

Dummy methods, which are not executed, will have minimal effect on runtime cost but dynamic watermarks may have a high runtime cost as the watermark is built during program execution. The fidelity of watermark, 'the extent to which embedding the watermark detoriates the original content' [131], should also be taken into account for the effects caused by watermarking, for example embedding a watermark may introduce unintentional errors.

The ideal recognition time of a watermark will most likely be quick but in some cases it may be important to artificially slow watermark recognition time to prevent oracle attacks [131]. Such attacks rely on the repetitive execution of a recongiser thus fast recognition time helps an adversary.

Types of Watermark

Nagra *et al.* define four types of watermark [131]:

Authorship Mark identifying a software author, or authors. These watermarks are generally visible and robust.

Fingerprinting Mark identifying the channel of distribution, i.e. the person who leaked the software. The watermarks are generally invisible, robust and consist of a unique identifier such as a customer reference number.

Validation Mark to verify that software is genuine and unchanged, for example like digitally signed Java Applets. These watermarks must be visible to the end-user to allow validation and fragile to ensure the software is not tampered with.

Licensing Mark used to authenticate software against a license key. The key should become ineffective if the watermark is damaged therefore licensing marks should be fragile.

Watermarking Techniques

Software watermarks can be broadly divided into two categories: static and dynamic [30]. The former embeds the watermark in the data and/or code of the program, while the latter embeds the watermark in a data structure built at runtime.

Static watermarks are embedded in the code and/or data of a computer program, a trivial example would be embedding a copyright notice in a string. Java class-files could contain software watermarks within their method bodies or their constant pool. The problem with storing a watermark as a string in a computer program is that unused variables could be easily removed with a simple dead-code analysis, and method or variable names are either lost during compilation or obfuscation. Other static watermarking techniques may involve transformations such as re-arranging or replacing instructions or basic blocks.

Some of the first static software watermarking techniques [44**?**] were describe in patents before academic researchers became interested in the area.

Dynamic watermarking techniques store a watermark in a program's execution state, rather than in the program code itself [29], therefore they should be resilient to semantics preserving transformations.

'Easter eggs' [30] are a type of dynamic watermark and, while they provide little protection, they demonstrate the idea of dynamic watermarking. An easter egg is hidden, by a programmer, within a program which usually performs some immediately perceptible action, such as displaying a message or exposing a hidden game. Easter eggs are usually easy to find and there are websites [165] dedicated to sharing easter eggs, in software, video games, music, TV shows and movies. Because easter eggs are usually easy to find and shared easily once found they are impractical to use for identifying software.

Once an easter egg has been found standard program analysis and transformation techniques such as slicing [169] can be used to remove the watermark.

Program Transformation Attacks

Program transformation attacks on watermarked software can be divided into three categories:

Additive An additive attack involves inserting another watermark into an already watermarked application, thus over-writing the original watermark. This attack will usually work if a watermark of the same type is embedded but not necessarily if a different type of watermark is embedded [125].

Subtractive A subtractive attack involves removing the section, or sections, of code where the watermark is stored while leaving behind a working program. This could be achieved by dead code elmination, statistical analysis or program slicing.

Distortive Distortive attacks involve applying semantics preserving transformations to a program, such as obfuscations or optimisations thus removing any watermarks which rely on program syntax. For example, renaming variables, loop transformations, function inlining, etc.

Both static and dynamic watermarks can be susceptible to program transformation attacks. Myles *et al.* [125] conducted an evaluation of dynamic and static versions of the Arboit algorithm by watermarking and obfuscating test files. They found the dynamic version to be only minimally stronger than the static version, and both versions could be defeated by distortive attacks.

1.2 Overview Of This Report

We have introduced Java, decompilation and software watermarking which we evaluate in further chapters. Chapter 2 contains an evaluation of current Java bytecode decompilers. Chapter 4 presents a survey of static software watermarking techniques and an evaluation of their effectiveness for software protection. Finally, we present details for further work and a timetable for completion in chapter 5.

Chapter 2

An Evaluation of Current Java Bytecode Decompilers

The decompilation of Java bytecode involves transforming the low-level, stack based bytecode instructions into high-level Java source. There are several main problems [116, 54] to solve in the decompilation of Java bytecode:

- local variable typing
- merging stack-based instructions into expressions
- arbitrary control flow
- exceptions and synchronisation

Decompiling Java bytecode is easy when compared to decompiling machine code as there are far fewer problems to overcome due to the amount of information contained within a class file. Java bytecode decompilers have the following advantages [54] over machine code decompilers:

- Data is already separated from code as all the data is separately stored a class file's constant pool.
- Separating pointers (references in Java) from constants is easy (e.g. some opcodes such as *aload* work with references whereas *iconst* works with constant integers).
- Method parameter, method return and field types are stored in the class file.
- There is no need to decode global data since everything is stored within class files.

In this chapter we present an evaluation of existing commercial, free and open-source decompilers and attempt to measure their effectiveness using a series of test programs.

We base this chapter on a survey performed in 2003 [55] which tested 9 decompilers. Some of the originally tested decompilers have been updated, some are now unavailable and there are also some new decompilers. We perform the original tests with some new decompilers and re-test other decompilers with the latest version of Java class files. We also add some of our own tests which add some decompilation problem areas which were not included in the original survey.

This chapter is organised as follows: In Section 2.4.2, the method of decompiler evaluation is described. In Section 2.4, we present the results of our evaluation and in Section 2.6, we present our conclusions.

2.1 Java Decompilation Example

In this section, we present a step by step decompilation of the Jasmin code in listing 2.1 using a stack evaluation method and taking advantage of Java to Java bytecode patterns.

Listing 2.1: Jasmin code for sum method

```
.method public static sum([I)V
    .limit stack 3
    .limit locals 3
a:
    iconst_0
    istore_1
    iconst_0
    istore_2
b:
    iload_2
    aload_0
    arraylength
    if_icmpge c
    iload_1
    aload_0
    iload_2
    iaload
    iadd
    istore_1
    iinc 2 1
    goto b
c:
    iload_1
    ifge d
    getstatic java.lang.System.out Ljava/io/PrintStream;
    ldc "total is less than zero :("
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    goto e
d:
    getstatic java.lang.System.out Ljava/io/PrintStream;
    new java/lang/StringBuilder
    dup
    invokespecial java/lang/StringBuilder/<init>()V
    ldc "total is "
    invokevirtual java/lang/StringBuilder/append(Ljava/lang/String;)Ljava/lang/StringBuilder;
    iload_1
    invokevirtual java/lang/StringBuilder/append(I)Ljava/lang/StringBuilder;
    invokevirtual java/lang/StringBuilder/toString()Ljava/lang/String;
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
e:
    return
.end method
```

Firstly, we can see that the method signature, `public static sum([I)V`, is similar to Java source: it is public, static, the name of the method is sum and the parameters are shown in parenthesis. However, the syntax for parameters is not the same as Java, and the method return type is at the end of the signature.

In Java bytecode the `I` denotes an integer type and the square bracket preceding it declares a one dimensional array; notice also, that the parameter does not have a name. The capital V denotes the void return type. So we can deduce that the method signature in Java source will be (we've made up the variable name):

```
public static void sum(int[] numbers)
```

The next two lines are instructions for the compiler - the maximum stack height and the number of local variables that this method uses.

The first bytecode instruction, `iconst_0`, pushes a 0 onto the stack and the next bytecode instruction, `istore_1`, pops an integer from the stack and stores it in local variable slot 1. Local variable slot 0 contains the parameter array (if this was an instance method local variable slot 0 would contain a reference to the instance of the class i.e. the `this` variable in Java). This combination can be thought of as assigning the value of 0 to an integer variable in Java; we must also declare the variable if it hasn't already been declared:

```
int total = 0;
```

Similarly, the next two bytecode instructions declare another integer variable and assign it the value 0:

```
int i = 0;
```

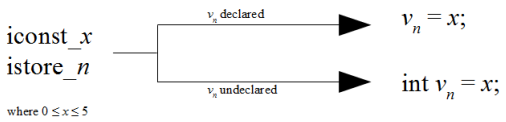


Figure 2.1: Decompilation Pattern - Integer variable assignment

The next three instructions can be considered together: `iload_2`, `aload_0` and `arraylength`. When we're decompiling a program we can use the stack to build up expressions by pushing and popping the variables and expressions, instead of their values. The first instruction pushes the integer in local variable slot 2 onto the stack, the second pushes the parameter array onto the stack. So after these instructions our expression stack is:

numbers
total

Next, the `arraylength` instruction pops an array from the stack and pushes it's length. We pop the numbers variable from our expression stack and push the `numbers.length` expression:

numbers.length
total

The next instruction, `if_icmpge`, is an integer comparison jump instruction; this instruction pops two integers a, b from the stack and jumps to the given label if $a \geq b$. This is not, however, to be decompiled as an if-expression in the Java source; it is, in-fact, the condition in a loop. We know that this is a `while` loop because the instruction before the target of the jump instruction is a `goto` instruction; this `goto` instruction jumps backward, to before the conditional instruction, which indicates that it is a `while` loop and not an `if` statement. A `do-while` loop condition would appear with the `goto` at the beginning. We must use the inverse of the condition in our while loop because the bytecode condition transfers control flow to the end of the loop if the condition is true; however, we want the loop to start at the beginning. Our condition is therefore `i < numbers.length`.

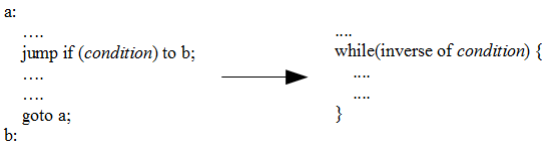


Figure 2.2: Decompilation Pattern - While Loop

The next 3 instructions push the values of the 3 variables in our method onto the stack; we push the variables onto our expression stack:

i
numbers
total

The next instruction, `iaload`, pops an integer i and an array reference arr from the stack; it then pushes the value at index i in the array arr onto the stack. We push the expression $numbers[i]$ onto our expression stack:

numbers[i]
total

Next, the instruction `iadd` pops two integers from the stack and pushes their sum back onto the stack. We therefore pop our two variables from our expression stack and push the expression `total + numbers[i]`.

The following instruction, `istore_1`, pops an integer from the stack and stores it in local variable slot 1. We pop an expression from our expression stack and assign it to our variable `total`.

Figure 2.3: Decompilation Pattern - Integer Array

```
total = total + numbers[i];
```

The last instruction within the loop is an integer increment instruction `iinc 2 1` - it increments the integer value in local variable slot 2 by 1. This is equivalent to `i++`.

Figure 2.4: Decompilation Pattern - Integer Increment

So the entire `while` loop looks like this:

```
int i = 0;
while(i < numbers.length) {
    total = total + numbers[i];
    i++;
}
```

After the loop, the first instruction is `iload_1` which pushes the value in local variable slot 1 onto the stack; we push the variable onto our expression stack:

Next, we have another condition jump instruction, `ifge d`, which pops an integer value from the stack and jumps to the specified label if the value is greater than or equal to zero. This time we are not dealing with a `while` loop; we know this because the `goto` instruction preceding the jump target is a forward jump, rather than a backward jump. Therefore, the code between the conditional jump and the `goto` is the if's *then*; the code from the `goto` until the `goto`'s jump target is the *else*. Again, we must invert the conditional giving us the condition `total < 0`.

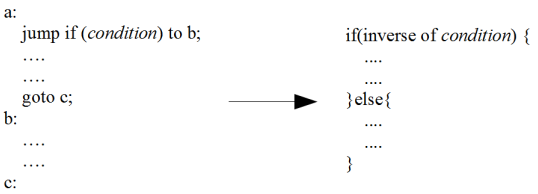
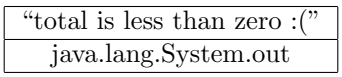


Figure 2.5: Decompilation Pattern - If-then-else

The first instruction in the body of the *then* clause is `getstatic` which pushes a reference to the specified static field onto the stack - in this case the field
`java.lang.System.out`. The next instruction, `ldc`, pushes a constant onto the stack. Our expression stack therefore looks like this:



The next instruction `invokevirtual` invokes the specified instance method, by first popping the correct number of arguments from the stack and lastly popping the object reference, on which the instance method acts, from the stack. The method specified is `java/io/PrintStream/println(Ljava/lang/String;)V`

- that is, in Java, the `void println(String s)` in the
`java.io.PrintStream` class. The object reference `java.lang.System.out`, on the bottom of the stack, is an instance of `java.io.PrintStream`. We therefore pop the string from the stack, followed by the object reference and build our method call for Java:

```
java.lang.System.out.println("total is less than zero :(");
```

The *else* clause is similar to the *then* clause but we build up a longer string using a `java.lang.StringBuilder` instance. The first instruction, again, pushes
`java.lang.System.out` onto the stack. The next instruction is `new` which creates an instance of the specified class and pushes a reference to the instance onto the stack. This is followed by `dup` which duplicates the item at the top of the stack, giving us an expression stack like this:

new java.lang.StringBuilder()
new java.lang.StringBuilder()
java.lang.System.out

The `invokespecial` instruction is then used to call the `java.lang.StringBuilder`'s constructor; it pops the object reference from the top of the stack. The next instruction `ldc` pushes the constant `total is` onto the stack:

"total is"
new java.lang.StringBuilder()
java.lang.System.out

The `invokevirtual` instruction then pops the string from the stack, followed by the `java.lang.StringBuilder` instance and invokes the
`public StringBuilder java.lang.StringBuilder.append(String s)`
method; it pushes the result back onto the stack:

new java.lang.StringBuilder().append("total is")
java.lang.System.out

Then `iload_1` is used to push the integer value in local variable slot 1 onto the stack. In our case, we push the variable `total` onto the stack and then use `invokevirtual` to call the append method again:

new java.lang.StringBuilder().append("total is").append(total)
java.lang.System.out

Finally, we invoke the
`public String java.lang.StringBuilder.toString()` method and invoke the
`public void java.io.PrintStream.println(String s)` method to print out the result. We end up with an empty stack and the following code:

```
java.lang.System.out(new java.lang.StringBuilder().append("total is").append(total).toString());
```

Putting all this together gives us the Java code in listing 2.2. We can tidy the code up slightly by removing `java.lang.`, converting the *while* loop to a *for* loop and turning the `StringBuilder` into standard string concatenation to obtain listing 2.3.

Listing 2.2: Java code for sum method

```
public static void sum(int[] numbers) {
  int total = 0;

  int i = 0;
  while(i < numbers.length) {
    total = total + numbers[i];
    i++;
  }

  if(total < 0) {
    java.lang.System.out.println("total is less than zero :(");
  }else{
    java.lang.System.out(new java.lang.StringBuilder().append("total is").append(total).toString
        ());
  }
}
```

Listing 2.3: Java code for sum method - nicer version

```
public static void sum(int[] numbers) {
  int total = 0;

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

  if(total < 0) {
    System.out.println("total is less than zero :(");
  }else{
    System.out.println("total is " + total);
  }
}
```

2.2 The Decompilers

The currently available decompilers are summarised in figure 2.6 and explained further in this section.

decompiler	type	status	2003 version	current version	last update
Mocha	free	obsolete	0.1b	0.1b	1996
SourceTec	commercial	obsolete	1.1	1.1	1997
SourceAgain	commercial	obsolete	1.10j	1.1	2004
Jad	free	unmaintained	1.5.8e	1.5.8e	2001
JODE	open-source	unmaintained	unknown	1.1.2-pre1	2004
ClassCracker3	commercial	obsolete	3.01	3.02	2005
jReversePro	open-source	unmaintained	1.4.1	1.4.2	2005
jdec	open-source	current	N/A	2.0	2008
Dava	open-source	current	2.0.1	2.4.0	2010
Java Decompiler	free	current	N/A	0.3.2	2010

Figure 2.6: The Decompilers

2.2.1 Mocha

Mocha [1], released as a beta version in 1996, was one of the first decompilers available for Java along with a companion obfuscator named Crema. Mocha can only decompile earlier versions of Java as it is an old program. Mocha is obsolete but is still available on several websites as the original license permitted its free distribution. The product was discontinued and never made it out of beta.

2.2.2 SourceTec (Jasmine)

SourceTec [2], also known as Jasmine, is another unmaintained old compiler which is a patch to Mocha. The installation process involves providing Mocha's class files which are then patched by SourceTec (Jasmine).

2.2.3 SourceAgain

SourceAgain [156] was a commercial decompiler from Ahpah Software, Inc. It is no longer sold or supported though they keep a web based version of their decompiler available on their website which can only decompile single class files. The original survey [55] used the Pro version which is no longer available.

2.2.4 ClassCracker3

ClassCracker3 [147] is another commercial decompiler which seems not to have been updated for at least four years. An evaluation version of the program is available at the Mayon Software Research's website which states that it will decompile the first 5 methods of a Java class file.

2.2.5 Jad

Jad [96] is a popular decompiler that is free for non-commercial use but is no longer maintained[1]. It is a closed source program written in C. The last update for Linux and Windows version for Jad was in 2001, while a small update added an OS X version in 2006. Jad is used as the back-end by many decompiler GUIs including an Eclipse IDE plug-in named JadClipse [69].

2.2.6 JODE

JODE [82] is an open-source decompiler that also includes a bytecode optimiser. The latest version 1.1.2-pre1 was released February 24, 2004. JODE performed best in the original survey [55] by decompiling six out of ten programs correctly.

[1]The original website is no longer active as of 25/02/2009 but Jad can still be obtained from `http://www.varaneckas.com/jad`

2.2.7 jReversePro

jReversePro [99] is an open-source disassembler and decompiler project which is currently at version 1.4.2 though hasn't been updated for several years.

2.2.8 Dava

Dava [114, 128, 127, 115, 116] is a decompiler which is part of the Soot Java Optimisation Framework [164] from the Sable Research Group[2] at McGill University in Montreal, Quebec, Canada. Soot is under constant development at the Sable Research Group and the latest release was version 2.4.0 on March 29th, 2010. The first version of our evaluation used version 2.3.0.

2.2.9 jdec

jdec [14] is an open-source decompiler written in Java which was last released at version 2.0 in May, 2008. jdec is aimed at the decompilation of bytecode generated by Sun's *javac* compiler and therefore will probably have problems decompiling the arbitrary code.

2.2.10 Java Decompiler

Java Decompiler [47] is a free Java decompiler aimed at decompiling Java 5 and above class files. It is in its early stages, at only version 0.3.2, and has been in development for about a year. The first version of our evaluation used version 0.2.7.

2.2.11 NMI Code Viewer

NMI Code Viewer was included in the original survey with results 'startlingly similar' to Jad [55]. We do not include this (unmaintained) decompiler as, in actual fact, it is a front-end for Jad [96].

2.2.12 jAscii

jAscii was included in the original survey, with poor results [55], but it is now obsolete and unavailable for our evaluation.

[2] `http://www.sable.mcgill.ca/`

2.3 Problems with Java Decompilation

2.3.1 Casting

The program in listing 2.4 shows an example in which some decompilers omit the int typecast [55].

Listing 2.4: Java Casting example

```
public class Type {

  public static void main(String[] args) {
    char c = 'a';

    System.out.println("ascii code for " + c + " is " + (int)c);
  }

}
```

2.3.2 Inner Classes

The implementation of inner classes in Java is handled by the compiler - the Java Virtual Machine has no concept of inner classes. The Java compiler converts the inner classes in a Java source file into separate class files therefore a decompiler needs to be able to reconstruct the original class file from several separate class files.

2.3.3 Type Inference

Type inference is a general problem in decompilation and is needed to recover type information lost during the compilation process. Type analysis in Java bytecode decompilation is easier than that of machine code decompilation due to the explicit typing of class fields and parameters in a Java class file. It is easier to decompile Microsoft's CIL bytecode as all types are explicit in the assembly files [54].

One of the first papers written on decompiling Java was a description of a decompiler named Krakatoa [140] which focused on two problems: merging stack-based instructions into expressions and turning arbitrary control flow into high-level Java constructs. The type-inference problem was dismissed as trivial and solvable by well known techniques such as those used by the Java bytecode Verifier. The Verifier subjects Java programs to a verification process in order to guarantee that the code will behave normally [149].

In actual fact, the type inference problem is not the same as that performed by the Verifer and is NP-Hard in the worst case [63]. The type analysis performed by the bytecode Verifier estimates the types of local variables at each program point. This is used to ensure that each bytecode instruction is operating on the correct type. For example if the Verifier encounters the *iadd* opcode it will ensure that there are two integers at the top of the stack at that program point [106]. The type inference problem for decompilation must give types to each variable that are valid for all uses of those variables.

If the type inference algorithm is optimised for the common-case rather than the worst case it is still possible to perform type analysis for most real-world code as worst-case scenerios are unlikely [13]. Bellamy et al. [13] provide a common-case optimised algorithm to perform type analysis which outperforms Gagnon et al. [63]'s worst-case optimised algorithm. The algorithm relies on the assumption that multiple inheritance, via interfaces, is rarely used in real-world Java programs which could prove a problem if their assumption is false, or becomes false in the future.

Java's restricted form of multiple inheritance, interfaces, leads to problems in finding a static type in some cases where the code is valid but not generated directly from Java source [63]. This causes problems for some decompilers which expect *javac* generated code. Using an static single assignment form, as in [92], may change the type hierarchy when types conflict due to interfaces [116].

Listing 2.5, page 28 (from [160]) shows a program which some decompilers may decompile better than others. If a decompiler does not perform type inference it will type x as Object and insert a typecast in the invocation of method m2(Comparable) whereas others may correctly type x as Comparable. The Java Virtual Machine Specification states that "the merged state contains a reference to an instance of the first common superclass of the two types" in regards to the bytecode verifier - the first common

Listing 2.5: What is the type of x? [160]

```
//Java:
  public void m1(Integer i, String s) {
    Comparable x;

    if(i != null)
      x = i;
    else
      x = s;

    m2(x);
  }

  public void m2(Comparable x) {
  }

//bytecode:

public void m1(java.lang.Integer, java.lang.String);
   0: aload_1
   1: ifnull  9
   4: aload_1
   5: astore_3
   6: goto  11
   9: aload_2
   10:  astore_3
   11:  aload_0
   12:  aload_3
   13:  invokevirtual #12; //Method m2:(Ljava/lang/Comparable;)V
   16:  return

public void m2(java.lang.Comparable);
   0: return
```

superclass of Integer and String is Object. Therefore the verifier has to insert a run-time check to ensure that both Integer and String are of type Comparable.

2.3.4 Control Flow

The *goto* statement is found in many programming languages which causes the execution of a program to jump to another position, usually labelled with an identifier or a number depending on the language. Java bytecode has two forms of *goto*: conditional and unconditional.

The Java language has restricted variants of *goto* in the form of the *break* and *continue* statements. The *break* statement is either labelled or unlabelled. The unlabelled form can be used to terminate a loop or to terminate *case* statements within a *switch* block. A labelled *break* statement terminates the labelled statement, such as a for-loop, and is useful when using nested loops to break an outer loop.

Java *switch* statements are effectively multi-way *goto* statements where the execution flow is changed based on an expression and possible values of the evaluated expression.

Due to the arbitrary control flow in Java bytecode and the more restricted high-level constructs in Java it can be difficult to translate Java bytecode program into Java.

Ramshaw's *goto* elmination technique can be used to replace *goto*s in a program, if and only if the control flow graph is reducible, by replacing them with multi-level loop exit statements [146]. The Krakatoa decompiler uses an extended version of Ramshaw's *goto* elimination technique [140].

Java bytecode can contain non-reducible control flow which cannot be represented in Java source requiring techniques such as described in [84] to be applied to transform irreducible control flow graphs to reducible control flow graphs.

2.3.5 Exceptions

An exception, as described in the Java Language Specification [67], is an error signalled to a program by the Java virtual machine when a program violates the semantic constraints of the language. When an exception occurs control is transferred from the point where the exception occurred to a point specified by a programmer (the *catch clause*); this is known as *throwing* and *catching*.

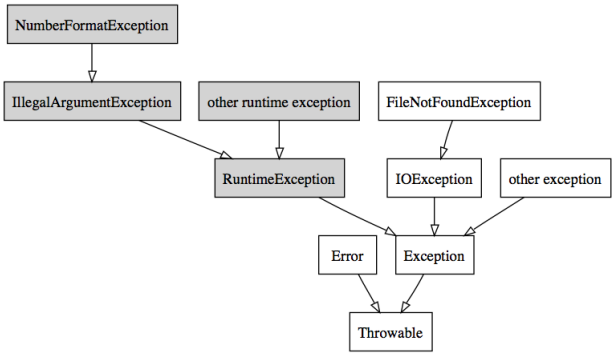


Figure 2.7: Java Exception Hierarchy. Unchecked exception classes are coloured grey.

Listing 2.6: Example of try-catch

```
try {
  Integer.parseInt(s);
}catch(NumberFormatException e) {
        System.out.println("Enter an integer");
}catch(Exception e) {
  System.out.println("some other error");
}
```

Errors in Java are organised in a hierarchy where the root object is the class Throwable, which has two direct subclasses: Exception and Error. An object must be of type Throwable or one of it's subclasses to be thrown.

Exception and its subclasses, except RuntimeException, are errors which are commonly expected to occur such as an IOException that the program may wish to recover from. RuntimeException and subclasses are different in that they are used to handle events which are not generally expected to occur but from which recovery is possible, such as a NumberFormatException due to wrong user input.

Error and its subclasses represent serious errors that a program cannot be expected to recover from such as an OutOfMemoryError. They are distinct from Exception to allow programmers to catch errors from which recovery is usually possible (Exception) and not require the inclusion of extra code to catch errors from which recovery is usually impossible (Error). Figure 2.7, page 29 shows the Java exception hierarchy.

Figure 2.6, page 29 shows a try-catch block with a call to the method *Integer.parseInt* that takes a Unicode string and returns it's *int* value or throws a NumberFormatException if the string does not represent an integer. If an exception is thrown control passes to the catch block corresponding to the class, or a super-class, of the exception thrown. The first catch clause, in order of appearance in the source file with a matching type is executed - in this case if s is not an integer the message "Enter an integer" will be displayed. The order of catch clauses is important because more than one catch clause could handle the same exception. For example the clause *catch(Exception e)* {...} can also handle the NumberFormatException as Exception is a superclass; subclass catch clauses must precede superclass catch clauses.

Programmers can define their own exceptions by extending Throwable or any of its subclasses. The Java Language Specification [66] recommends that all user defined exceptions extend Exception rather than Throwable or RuntimeException because Exception and its subclasses are **checked** exceptions whereas RuntimeException and its subclasses are **unchecked**.

A Java compiler ensures that a program contains handlers for checked exceptions during compilation so for each method which could possibly throw a checked exception there must be a handler or the method must declare that it itself throws the exception. If a method declares that it throws an exception it must be caught or thrown in the invoking method, and so on.

Listing 2.7, page 30 shows three methods which return a FileReader object for the specified filename. FileReader's constructor throws a FileNotFoundException if the file specified was not found. This is a checked exception therefore any invocation of the constructor requires an exception handler or that the

Listing 2.7: Example of checked exceptions

```
public class CheckedException {

        public static void main(String[] args) {

                try {
                        FileReader f = getFileReader1(args[0]);
                }catch(FileNotFoundException e) {
                        //handle file not found exception
                }

                //returns null if file not found
                FileReader g = getFileReader2(args[0]);

        }

        public static FileReader getFileReader1(String filename) throws FileNotFoundException {
                return new FileReader(filename);
        }

        public static FileReader getFileReader2(String filename) {
                try {
                        return new FileReader(filename);
                }catch(FileNotFoundException e) {
                        return null;
                }
        }
        /*
        won't compile
  public static FileReader getFileReader3(String filename) {
    return new FileReader(filename);
  }
  */
}
```

method containing the invocation declares that itself throws FileNotFoundException (or a superclass of FileNotFoundException).

The first method declares that it throws a FileNotFoundException and therefore doesn't need to include a try-catch block. There is an exception handler at the first method's invocation in the main method.

The second method contains a try-catch block to handle the FileNotFoundException and the third method will cause a compilation-time error as it does not handle or throw FileNotFoundException or one of its superclasses.

Error and its subclasses are unchecked as they can occur at any point in a program and it is usually impossible to recover from them. Runtime exceptions are unchecked because many operations could throw a runtime exception and there isn't enough information available to the compiler for it to determine that a runtime exception cannot occur; it would also need too much exception handling code.

Programmers can throw exceptions themselves using the *throw* keyword. Listing 2.8, page 31 shows a method yesOrNo which takes a string and returns true if the string contains 'yes' and false if the string contains 'no' (including combinations of upper and lower case letters). If the string contains anything else WrongInputException is thrown.

WrongInputException is a subclass of Exception which means that it is a checked exception therefore when the yesOrNo method is invoked an exception handler must be used or the calling method must declare that it throws a WrongInputException.

Java bytecode contains the necessary information to implement exceptions as specified by the Java Language Specification [66]. Every method which contains a try-catch block in Java has an exception table attribute attached which includes the ranges of bytes for which to catch exceptions, the byte at which the exception handler starts and the type of exception. Figure 2.8 shows the bytecode, generated by *javac* 1.6, for listing 2.6, page 29. It contains two entries in the exception table corresponding to the two types of exception which are to be caught.

The first exception table entry lists an exception handler for the type NumberFormatException between bytes 0 (inclusive) to 5 (exclusive) and the second entry lists an exception handler for the type Exception between bytes 0 (inclusive) to 5 (exclusive). The Java Virtual Machine searches the exception

Listing 2.8: Throwing an exception

```
public class YesOrNo {

  public static void main(String[] args) {
                try {
                        System.out.println(yesOrNo(args[0]));
                }catch(WrongInputException e) {
                        System.out.println(e);
                }
  }


        public static boolean yesOrNo(String s) throws WrongInputException {
                if(s.toLowerCase().equals("yes")) {
                        return true;
                }else if(s.toLowerCase().equals("no")) {
                        return false;
                }else{
                        throw new WrongInputException("found " + s + ", expected yes or no.");
                }
        }

}



public class WrongInputException extends Exception {
      WrongInputException() { super(); }
  WrongInputException(String s) { super(s); }
}
```

```
public static void s(java.lang.String);
  Code:
   Stack=2, Locals=2, Args_size=1
   0: aload_0                                          // push s onto the stack
   1: invokestatic java/lang/Integer.parseInt:(Ljava/lang/String;)I   // invoke Integer.parseInt(s), push result onto stack
   4: pop                                              // pop stack
   5: goto 29                                          // goto 29 (skip over catch section)
   8: astore_1                                         // begin catch. pop exception from stack, store in local 1
   9: getstatic java/lang/System.out:Ljava/io/PrintStream;   // push System.out onto stack
   12: ldc "Enter an integer"                          // push "Enter an integer" onto the stack
   14: invokevirtual java/io/PrintStream.println:(Ljava/lang/String;)V   // invoke System.out.println("Enter an integer")
   17: goto 29                                         // end catch. goto 29 (skip over catch section)
   20: astore_1                                        // begin catch. pop exception from stack, store in local 1
   21: getstatic java/lang/System.out:Ljava/io/PrintStream;   // push System.out onto stack
   24: ldc "some other error"                          // push "some other error" onto the stack
   26: invokevirtual java/io/PrintStream.println:(Ljava/lang/String;)V   // invoke System.out.println("some other error")
   29: return                                          // end catch. return.

  Exception table:
   from   to  target type
     0    5    8   Class java/lang/NumberFormatException
     0    5   20   Class java/lang/Exception
```

Figure 2.8: Java bytecode for listing 2.6, page 29.

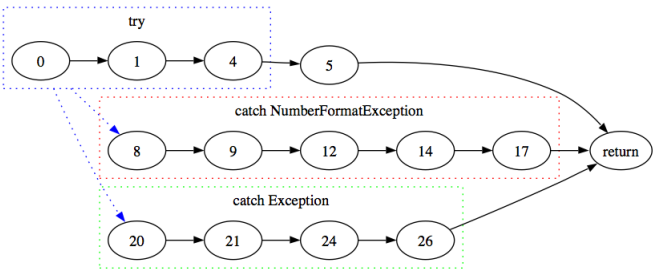


Figure 2.9: Java bytecode control flow graph for listing 2.6, page 29 with highlighted try-catch sections.

table starting at the top for the first matching entry for the type of exception that was thrown. So if a NumberFormatException is thrown control is transferred to byte 8 and if an Exception or a subclass excluding NumberFormatException is thrown control is transferred to byte 20.

If no matching exception handler is found the method completes abruptly, the method frame is discarded and the exception is re-thrown in the invokers method frame and so on. If no matching exception handler is found in the method execution chain the thread within which the exception was thrown is terminated.

The first opcode of an exception handler is always an **astore** instruction which stores the Throwable object which should be on the top of the stack into some local variable slot. The bytecode verifier must check that a Throwable object will be on the stack at the beginning of an exception handler section.

An exception is thrown in bytecode using the **athrow** opcode which pops a Throwable object from the stack and throws it.

Try-Finally

The finally clause of a try statement is guaranteed to execute even if the try block completes abruptly. The subroutine for the finally clause is invoked at each exit point of a try block and its associated catch block. The finally clause allows 'clean-up' code to be executed such as closing of database connections or deleting temporary files, even if the try block completes abruptly. The finally clause is executed after the try block completes normally and also if the try block exits with a return, break or throwing of an exception.

Many of the old decompilers expect the use of subroutines for try-finally blocks but *javac* 1.4.2+ generates in-line code instead. An arbitrary decompiler, such as Dava, may also have problems decompiling this simple program due to the way it analyses bytecode for decompilation.

Java bytecode subroutines, like subroutines in any other programming language, were designed to allow code re-use for the implementation of finally clauses. For each exit point in a try clause the same finally block must be executed which seems to suggest subroutines would be a good idea to implement this. Java 1.4.2 and above repeat the finally clause bytecode at every try exit point rather than using subroutines. This has the disadvantage that the code size is much bigger than when using subroutines but data flow analysis is much simplified [60]. It also means that bytecode verification is simplified as the verifier performs a form of data flow analysis on the bytecode. However, bytecode produced by *javac* $<$ 1.4.2 and other compilers could still contain subroutines.

The *jsr* opcode takes a two-byte operand indicating the offset from the *jsr* instruction to the subroutine. When the *jsr* opcode is executed the address of the next opcode is pushed onto the stack and control is passed to the *jsr* specified by the operand.

The first instruction of a finally clause pops the stack and stores the return address (bytecode type **returnAddress**) in a local variable.

The *ret* opcode takes one operand - the index of the local variable slot where the return address is stored - execution then continues at the address loaded from this local variable slot.

Listing 2.9: Multiple local variable types

```
0: iconst_0 //push 0 onto stack
1: istore_0 //pop integer from stack, store in local 0
2: ldc "hello"  //push String constant onto stack
3: astore_0 //pop object reference from stack, store in local 0
4: return
```

2.3.6 Variable Re-Use

It is possible for a local variable slot to take values of distinct types at different places in a method [63] which is not possible in Java. For example listing 2.9 shows a simple valid method in which local variable 0 is used to store an integer then a String. At byte 1 local variable 0 contains a variable of type integer but at byte 3 it contains an object reference type. This is perfectly valid bytecode and is akin to the declaration of an integer followed by the declaration of a String in Java, but in the bytecode these two distinct variables are using the same local variable slot.

Multiple types for local variables is valid bytecode as long as the lifetimes of the two uses of the local variable does not overlap [92] i.e. a local variable only has the type (and value) of the last variable stored in it.

This may be overcome by converting the bytecode to static single assignment form [6, 150, 41] where all static assignments to a variable will have unique names. The possibility of multiple types for a single local variable slot causes a problem for decompilation as each variable needs a single type which is valid for all uses of that variable.

2.3.7 Arbitrary bytecode

Arbitrary bytecode may differ from compiler generated bytecode and may code that isn't easily translatable into Java source. In most cases this might not be a problem; for example, is there really a need for decompiling to Java and program compiled from Ada. However, optimising a *javac* generated class file turns the bytecode into a form of arbitrary code - that is, the optimisations transform the code in ways which decompilers don't expect.

Listing 2.10: Fibo Test Program

```
class Fibo {
    private static int fib (int x) {
        if (x > 1)
            return (fib(x - 1) + fib(x - 2));
        else return x;
    }

    public static void main(String args[]) throws Exception {
      int number = 0, value;

        try {
            number = Integer.parseInt(args[0]);
        } catch (Exception e) {
            System.out.println ("Input error");
            System.exit (1);
        }
        value = fib(number);
        System.out.println ("fibonacci(" + number + ") = " + value);
    }

}
```

2.4 Empirical Evaluation

Our evaluation is based on a previous survey conducted in 2003 [55] which tested many of the decompilers that are still available today. We attempt to decompile a set of test programs using each decompiler and manually inspect the results to classify them as one of our ten categories of decompiler effectiveness (section 2.4.2 describes our effectiveness measure). We use the test programs from the original survey, where possible, and also extend the evaluation to include more problem areas such as the correct decompilation of try-finally blocks and local variable slot re-use.

2.4.1 Test Programs

The test programs were taken from sources dealing with different problem areas of bytecode decompilation and they provide interesting problems for decompilers.

The original survey showed that different decompilers are sometimes better in different areas but no decompiler passed all the tests [55]. Of the decompilers tested in the original survey, JODE [82] performed the best by correctly decompiling 6 out of the 9 test programs, with Dava [114] and Jad [96] close behind.

In our tests we include two types of Java class file: those generated by *javac* and those generated by other tools, which will be refered to as *arbitrary* bytecode class files. Java source files are compiled with *javac* version 1.6.0_10. Arbitrary Java class files include two hand-written using the Jasmin assembler version 2.1, one optimised using the Soot Framework [164] and one compiled by JGNAT [3].

Fibo

Fibo (Listing 2.10, page 34) is a fairly simple program to output the Fibonacci number of a given input number. It should be a trivial test for a decompiler but contains many of the basic Java language constructs. The program tests decompilation of basic Java language constructs such as *if* statements, methods declarations & invocations and exception handling.

Casting

Casting [132] is a simple program to test if a decompiler can correctly detect the need to cast a *char* to an *int*. The program (Listing 2.11, page 35) iterates through the first 128 ASCII characters and prints out the ASCII code and the character. A cast from *char* to *int* is used to achieve this. The original survey [55] showed that some decompilers were unable to detect the need for the cast and instead printed the ASCII character instead of the ASCII code.

Listing 2.11: Casting Test Program

```
public class Casting {
    public static void main(String args[]){
        for(char c=0; c < 128; c++) {
            System.out.println("ascii " + (int)c + " character "+ c);
        }
    }
}
```

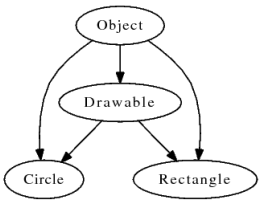


Figure 2.10: Type hierarchy for the type inference test program. *Drawable* is the lowest common ancestor of *Circle* and *Rectangle*.

Listing 2.12, page 36 shows the Jasmin source for the casting test program. The Java compiler (*javac*) converts string concatenations into string buffer appends for efficiency. The line marked #1 is the point at which the integer ASCII code is appended to the string buffer, and the line marked #2 is the point at which the ASCII character is appended to the string buffer - notice that line #1 invokes the method *append(integer)*, while line #2 invokes the method *append(character)*.

Usa

Usa [132] is a simple program (Listing 2.13, page 36) containing inner classes. The implementation of inner classes in Java is handled by the compiler - the Java Virtual Machine has no concept of inner classes. The Java compiler converts the inner classes in a Java source file into separate class files. For example, the Usa test program's main class is called *Usa* and it has an inner class called *England* - the Java compiler generates a class named *Usa$England* for the inner class and the inner class contains a private variable *this$0* which is a reference to the outer class. Some decompilers in the original survey [55] could not reconstruct the original Java source from the constituent class files.

Sable

Sable [116] is a program which tests a decompiler's ability to perform type inference for local variables. Variable d in the program (line 3, Listing 2.14, page 37) is difficult for a decompiler to type because it depends on the value of the method parameter, i. If $i > 10$ then d becomes a *Rectangle* object, whereas if $i <= 10$ d becomes a *Circle* object. Variable d should be typed *Drawable* as this is the lowest common ancestor of *Circle* and *Rectangle* (see Figure 2.10, page 35). The original paper [116], written by the developers of Dava, tested three different decompilers with the Sable test program, against Dava. They showed that their decompiler could correctly type variable d whereas the other decompilers failed to do so. Some decompilers do not perform type inference but insert a typecast where necessary in order to produce a semantically equivalent program [116, 55].

TryFinally

TryFinally is a simple test program (Listing 2.18, page 38) to determine whether decompilers can decompile the implementation of try-finally blocks using in-line code instead of Java bytecode subroutines.

Listing 2.19, page 39 shows the Jasmin source for the TryFinally test program. Sections b and d contain duplicate code which is the finally clause of the TryFinally block. Compare this with listing 2.20, page 40 which shows the Jasmin source for the TryFinally test program compiled with *Jikes*. This

Listing 2.12: Casting Test Program in Jasmin

```
; Produced by NeoJasminVisitor (tinapoc)
; http://tinapoc.sourceforge.net
; The original JasminVisitor is part of the BCEL
; http://jakarta.apache.org/bcel/
; Thu Jun 11 13:22:44 BST 2009

.bytecode 50.0
.source <Unknown>
.class public Casting
.super java/lang/Object

.method public <init>()V
    .limit stack 1
    .limit locals 1

    aload_0
    invokespecial java/lang/Object/<init>()V

    return

.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 3
    .limit locals 2
    .var 0 is arg0 [Ljava/lang/String; from Label2 to Label0

    Label2:
    iconst_0
    istore_1

    Label1:
    iload_1
    sipush 128
    if_icmpge Label0
    getstatic java.lang.System.out Ljava/io/PrintStream;
    new java/lang/StringBuilder
    dup
    invokespecial java/lang/StringBuilder/<init>()V
    ldc "ascii "
    invokevirtual java/lang/StringBuilder/append(Ljava/lang/String;)Ljava/lang/StringBuilder;
    iload_1
    invokevirtual java/lang/StringBuilder/append(I)Ljava/lang/StringBuilder;      ; #1
    ldc " character "
    invokevirtual java/lang/StringBuilder/append(Ljava/lang/String;)Ljava/lang/StringBuilder;
    iload_1
    invokevirtual java/lang/StringBuilder/append(C)Ljava/lang/StringBuilder;      ; #2
    invokevirtual java/lang/StringBuilder/toString()Ljava/lang/String;
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    iload_1
    iconst_1
    iadd
    i2c
    istore_1
    goto Label1

    Label0:
    return

.end method
```

Listing 2.13: Inner class Test Program

```
public class Usa {
    public String name = "Detroit";
    public class England {
        public String name = "London";
        public class Ireland {
            public String name = "Dublin";
            public void print_names() {
                System.out.println(name);
            }
        }
    }
}
```

Listing 2.14: Type Inference Test Program

```
public class Sable {
    public static void f(short i) {         // 2
        Circle c; Rectangle r; Drawable d; // 3
        boolean is_fat;

        if (i > 10) {                       // 6
            r = new Rectangle(i, i);        // 7
            is_fat = r.isFat();             // 8
            d = r;                          // 9
        }
        else {
            c = new Circle(i);              // 12
            is_fat = c.isFat();             // 13
            d = c;                          // 14
        }
        if (!is_fat) d.draw();              // 16
    }                                       // 17

    public static void main(String args[]) // 19
        { f((short) 11); }                  // 20
}
```

Listing 2.15: Drawable interface for Type Inference Test Program

```
public interface Drawable {
    public void draw();
}
```

Listing 2.16: Circle class for Type Inference Test Program

```
public class Circle implements Drawable {
    public int radius;
    public Circle(int r) {radius = r;}
    public boolean isFat() {return false;}
    public void draw() {
        // Code to draw...
       }
     }
```

Listing 2.17: Rectangle class for Type Inference Test Program

```
public class Rectangle implements Drawable {
    public short height, width;
    public Rectangle(short h, short w) {
        height = h; width = w; }
    public boolean isFat() {return (width > height);}
    public void draw() {
        // Code to draw ...
           }
           }
```

Listing 2.18: Try Finally Test Program

```
public class TryFinally {

  public static void main(String[] args) {
    try {
      System.out.println("try");
    }finally{
      System.out.println("finally");
    }
  }

}
```

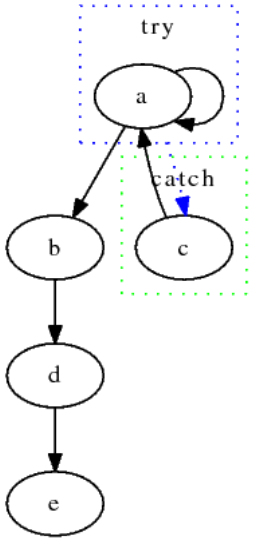


Figure 2.11: Control flow graph for the control flow program.

version uses a Java subroutine (section d) to implement the finally clause of the test program, which is invoked from sections b and e.

In our tests, the decompiled TryFinally test programs must use a try-finally block to be classified as correct, rather than relying on a combination of try-catch and in-lined finally clause code.

ControlFlow

ControlFlow [116] is a program which tests a decompiler's handling of control flow. The program contains two while loops: one outer, infinite loop; and one inner loop which is contained within a try-catch block. If the inner loop throws a *RuntimeException* then the outer loop continues otherwise the outer loop is broken.

Exceptions

The Exceptions test program [116] contains two intersecting try-catch blocks. The intersecting try-catch block is allowed in Java bytecode but would not be generated by a Java compiler - the program here is created using Jasmin. The program used in the original survey [55] is incorrect and Dava, which should be able to decompile the program, exits with a null pointer exception. A re-written version (Listing 2.23, page 41) is used in our tests based on the call graph in the original paper [116]. Figure 2.12, page 41 shows the program's control flow graph and outlines the try-catch blocks.

Listing 2.19: Try Finally Test Program *javac* - Jasmin Source

```
.bytecode 50.0
.source <Unknown>
.class public TryFinally
.super java/lang/Object

.method public <init>()V
    .limit stack 1
    .limit locals 1

       0: aload_0
       1: invokespecial java/lang/Object/<init>()V
       4: return

.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 2

    .catch all from a to b using c
    .catch all from c to d using c
a:
       0: getstatic java.lang.System.out Ljava/io/PrintStream;
       3: ldc "try"
       5: invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
b:
       8: getstatic java.lang.System.out Ljava/io/PrintStream;
      11: ldc "finally"
      13: invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
      16: goto e
c:
      19: astore_1
d:
      20: getstatic java.lang.System.out Ljava/io/PrintStream;
      23: ldc "finally"
      25: invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
      28: aload_1
      29: athrow
e:
      30: return
.end method
```

Listing 2.20: Try Finally Test Program *Jikes* - Jasmin Source

```
; Produced by NeoJasminVisitor (tinapoc)
; http://tinapoc.sourceforge.net
; The original JasminVisitor is part of the BCEL
; http://jakarta.apache.org/bcel/
; Tue Jun 16 16:26:04 BST 2009

.bytecode 48.0
.source <Unknown>
.class public TryFinally
.super java/lang/Object

.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 3

    .catch all from a to c using b
    .catch all from e to f using b

a:
    getstatic java.lang.System.out Ljava/io/PrintStream;
    ldc "try"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    goto e

b:
    astore_1
    jsr d
c:
    aload_1
    athrow
d:
    astore_2
    getstatic java.lang.System.out Ljava/io/PrintStream;
    ldc "finally"
    invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
    ret 2
e:
    jsr d
f:
    return

.end method

.method public <init>()V
    .limit stack 1
    .limit locals 1

    aload_0
    invokespecial java/lang/Object/<init>()V

    return

.end method
```

Listing 2.21: Control Flow Test Program

```
public class ControlFlow {
    public static int foo(int i, int j) {
        while (true) {
            try{
                while (i < j)
                    i = j++/i;

            }catch (RuntimeException re) {
                i = 10;
                continue;
            }
            break;
        }
        return j;
    }

  public static void main(String[] args) {
    System.out.println(foo(1,2));
  }
}
```

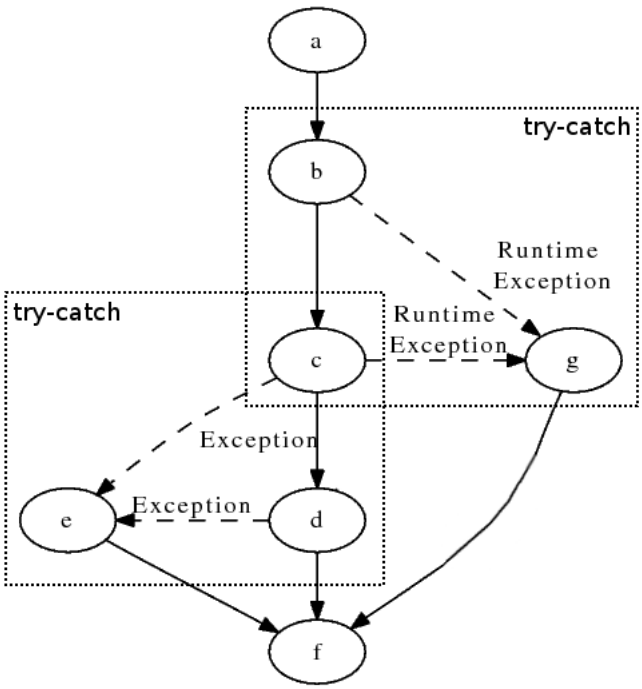


Figure 2.12: Control flow graph for the exceptions test program [116]. Solid edges indicate normal control flow while dashed edges represent exception control flow.

Listing 2.22: Control Flow Test Program

```
; Produced by NeoJasminVisitor (tinapoc)
; http://tinapoc.sourceforge.net
; The original JasminVisitor is part of the BCEL
; http://jakarta.apache.org/bcel/
; Thu Jun 18 15:43:18 BST 2009

.bytecode 50.0
.source <Unknown>
.class public ControlFlow
.super java/lang/Object


.method public <init>()V
    .limit stack 1
    .limit locals 1

    aload_0
    invokespecial java/lang/Object/<init>()V

    return

.end method


.method public foo(II)I
    .limit stack 2
    .limit locals 4

    .catch java/lang/RuntimeException from a to b using c
a:
    iload_1               ;1
    iload_2               ;2
    if_icmpge b               ;3
    iload_2               ;4
    iinc 2 1                ;5
    iload_1               ;6
    idiv                ;7
    istore_1                ;8
    goto a                ;9
b:
    goto d                ;10
c:
    astore_3                ;11
    getstatic java.lang.System.out Ljava/io/PrintStream;    ;12
    aload_3               ;13
    invokevirtual java/io/PrintStream/println(Ljava/lang/Object;)V  ;14
    bipush 10               ;15
    istore_1                ;16
    goto a                ;17
d:
    iload_2               ;18
e:
    ireturn               ;19

.end method
```

Listing 2.23: Exception Test Program (Jasmin)

```
.class  Exceptions
.super java/lang/Object

.method  <init>()V
    .limit stack 1
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 1
    .limit locals 1
    new Exceptions
    invokespecial Exceptions/<init>()V
    return
.end method

.method public Exceptions()V
  .limit locals 1
  .limit stack 2

  .catch java/lang/Exception    from c to f using e
  .catch java/lang/RuntimeException from b to d using g

a:
  getstatic java/lang/System/out Ljava/io/PrintStream;
       ldc "a"
       invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
b:
  getstatic java/lang/System/out Ljava/io/PrintStream;
       ldc "b"
       invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
c:
  getstatic java/lang/System/out Ljava/io/PrintStream;
       ldc "c"
       invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
d:
  getstatic java/lang/System/out Ljava/io/PrintStream;
       ldc "d"
       invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
f:
  getstatic java/lang/System/out Ljava/io/PrintStream;
       ldc "f"
       invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

  return

;catch blocks
e:
  astore_0
  getstatic java/lang/System/out Ljava/io/PrintStream;
       ldc "e"
       invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  goto f
g:
  astore_0
  getstatic java/lang/System/out Ljava/io/PrintStream;
       ldc "g"
       invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
  goto f

.end method
```

Optimised

Optimised was generated by using the Soot optimiser [164] on the *TypeInference* test program (listing 2.14, page 37). An example of an optimisation that has been performed is the replacement of the *dup* opcode (which duplicates the value on the top of the stack) with *load* and *store* instructions. Listing 2.24, page 45 shows the Jasmin source for the TypeInference test program and listing 2.25, page 46 shows the Jasmin souce for the Optimised test program. The optimised program's f method is 3 bytes smaller than in the original program.

Args

Args (listing 2.26, page 47) re-uses the local variable slot 0 in the main method. At the start of the method local variable slot 0 is of type *String[]* but it is then re-used as type *int*.

connectfour

The connectfour test program [59] is an implementation of the game Connect Four originally written in Ada and compiled with JGNAT [3] to Java bytecode. This program provides an example of a source language other than Java for a Java decompiler to handle. Such programs potentially contain code which cannot easily be decompiled to Java source due to unexpected bytecode sequences generated by a non-Java to Java bytecode compiler. The original survey used a different Ada program compiled to Java bytecode which we could not obtain.

2.4.2 Measuring the Effectiveness of Java Decompilers

The effectiveness of a Java decompiler, depends heavily on how the bytecode was produced. Arbitrary bytecode can contain instruction sequences for which there is no valid Java source due to the more powerful and less-restrictive nature of Java bytecode. For example there are no arbitrary control flow instructions in Java but there are in Java bytecode. In fact, many Java bytecode obfuscators rely on the fact that most decompilers fail when encountering unexpected, but valid, bytecode sequences [12].

Naeem et al. [129] suggest using software metrics [73, 90] for measuring the effectiveness of decompilers. They compare the output of several decompilers against the input programs using software metrics. Software metrics are a good measure of the complexity of the output of a decompiler however they cannot quantify the effectiveness of the general output of a decompiler.

Decompilers produce output of varying degrees of correctness and some produce no output at all. Comparing program metrics of a source program to a syntactically incorrect output program could prove difficult as it may require parsing of a syntactically incorrect program. Class files generated using an assembler or other language to bytecode compiler also present a problem for this suggested technique as the output Java source has no equivalent input Java source to be compared against.

The output of a Java decompiler can, crudely, be divided into three categories:

1. semantically and syntactically correct,
2. syntactically correct and semantically incorrect and
3. syntactically incorrect.

Clearly, a decompiler which produces output of the first category is desirable.

For each program, decompiler pair, we give a score between 0 and 9 (see Figure 2.13). A score of 0 is a perfect, or good, decompilation whereas 9 means that the decompiler failed and no output was produced. This is loosely based on Java decompiler evaluation categories used by Dyer [48].

The first two categories, 0 and 1 indicate output which is both syntactically correct Java and semantically equivalent to the original. Category 1 programs, however contain code which is less readable (e.g. excessive use of labels, while loops instead of for loops etc) and/or code for which no type inference has been performed.

Programs classified as 2, 3, 4 indicate varying levels of syntactically incorrect programs. A category 2 program indicates a program with small syntax errors, such as a missing variable declaration, that can be easily corrected to produce a semantically correct program. Category 3 programs contain syntactic

Listing 2.24: Jasmin source for the TypeInference test program

```
.bytecode 50.0
.source <Unknown>
.class public Sable
.super java/lang/Object

.method public <init>()V
    .limit stack 1
    .limit locals 1

       0: aload_0
       1: invokespecial java/lang/Object/<init>()V
       4: return

.end method

.method public static f(S)V
    .limit stack 4
    .limit locals 5
Label3:
       0: iload_0
       1: bipush 10
       3: if_icmple Label0
       6: new Rectangle
       9: dup
      10: iload_0
      11: iload_0
      12: invokespecial Rectangle/<init>(SS)V
      15: astore_2
      16: aload_2
      17: invokevirtual Rectangle/isFat()Z
      20: istore 4
      22: aload_2
      23: astore_3
      24: goto Label1
Label0:
      27: new Circle
      30: dup
      31: iload_0
      32: invokespecial Circle/<init>(I)V
      35: astore_1
      36: aload_1
      37: invokevirtual Circle/isFat()Z
      40: istore 4
      42: aload_1
      43: astore_3
Label1:
      44: iload 4
      46: ifne Label2
      49: aload_3
      50: invokeinterface Drawable/draw()V 1
Label2:
      55: return

.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 1
    .limit locals 1

      0: bipush 11
      2: invokestatic Sable/f(S)V
      5: return

.end method
```

Listing 2.25: Jasmin source for the Optimised (TypeInference) test program

```
.bytecode 46.0
.source Optimised.java
.class public Optimised
.super java/lang/Object

.method public <init>()V
    .limit stack 1
    .limit locals 1

       0: aload_0
       1: invokespecial java/lang/Object/<init>()V
       4: return

.end method

.method public static f(S)V
    .limit stack 3
    .limit locals 2
Label3:
       0: iload_0
       1: bipush 10
       3: if_icmple Label0
       6: new Rectangle
       9: astore_1
      10: aload_1
      11: iload_0
      12: iload_0
      13: invokespecial Rectangle/<init>(SS)V
      16: aload_1
      17: invokevirtual Rectangle/isFat()Z
      20: istore_0
      21: aload_1
      22: astore_1
      23: goto Label1
Label0:
      26: new Circle
      29: astore_1
      30: aload_1
      31: iload_0
      32: invokespecial Circle/<init>(I)V
      35: aload_1
      36: invokevirtual Circle/isFat()Z
      39: istore_0
      40: aload_1
      41: astore_1
Label1:
      42: iload_0
      43: ifne Label2
      46: aload_1
      47: invokeinterface Drawable/draw()V 1
Label2:
      52: return

.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 1
    .limit locals 1

       0: bipush 11
       2: invokestatic Optimised/f(S)V
       5: return

.end method
```

Listing 2.26: Variable re-use test program

```
.class Args
.super java/lang/Object

.method  <init>()V
    .limit stack 1
    .limit locals 1
    aload_0
    invokespecial java/lang/Object/<init>()V
    return
.end method


.method public static main([Ljava/lang/String;)V
    .limit stack 2
    .limit locals 1

     iconst_0
     istore_0

     getstatic java/lang/System/out Ljava/io/PrintStream;
     iload_0
     invokevirtual java/io/PrintStream/println(I)V

     return

.end method
```

errors which are harder to correct, such as programs with goto statements (which do not exist in Java), and category 4 programs contain extreme syntax errors which are very difficult or impossible to correct.

Programs classified as 5, 6, 7 are syntactically correct but are not semantically equivalent to the input program. These categories of programs are, in a sense, worse than syntactically incorrect programs as they re-compile without error and may contain subtle semantic errors which are not obvious, thus large programs in these categories would require a lot of testing to ensure their correctness. Programs in category 5 contain minor semantic errors which when corrected produce a program semantically equivalent to the input program. Category 6 and 7 contain harder to correct semantic errors and indicate programs that are dramatically semantically different from the input program.

Programs classified as category 8 are incomplete programs which are not decompiled in their entirety, for example a program which is missing inner classes.

Category 9 indicates that a decompiler failed to produce any output at all, and most likely failed to parse the input file. Problems could occur due to arbitrary bytecode or the latest Java class files which decompilers are not expecting.

As the categories increase the difficulty to read and understand the code also increases, for example both category 0 and 1 are semantically correct but category 1 is hard to read. Category 2 and higer programs are harder to understand as they are syntactically and/or semantically incorrect.

2.4.3 Summary of Results

No decompiler was able to decompile all test programs with JODE decompiling the most programs correctly. JODE only managed to decompile 5 programs while four (unmaintained) decompilers could not decompile any of the test programs correctly. The top four decompilers (excluding obsolete SourceAgain) were Java Decompiler, JODE, Dava and Jad whereas the worst decompilers were the unmaintained commercial decompilers - SourceTec, ClassCracker3 and Mocha. Some decompilers failed simply because they could not parse the latest class files or arbitrary class files.

Dava, surprisingly, was better at decompiling the *javac* bytecode test programs than the arbitrary test programs. Dava is the joint second best (with JODE) based on our effectiveness measures (excluding SourceAgain), but one of the best arbitrary bytecode decompilers along with Java Decompiler - Java Decompiler slighlty outperforms Dava in the arbitrary bytecode tests. Dava performs similarly to jdec with *javac* generated bytecode, both decompiling two of these correctly. Overall Dava decompiles correctly twice the number of programs that jdec decompiles.

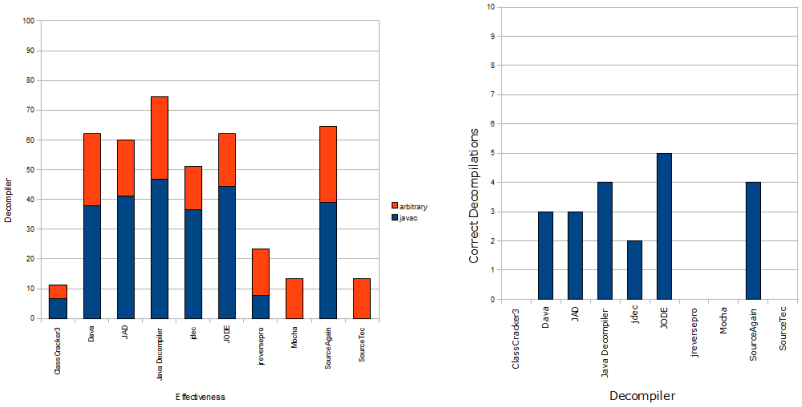
Java Decompiler scored the highest using our effectiveness measures, beating Jad and JODE by

Score	semantics	syntax	output result	examples
0	correct	correct	semantically and syntactically correct program with perfect/good source code layout	perfect decompilation
1	correct	correct	semantically and syntactically correct program with 'ugly' source code layout and/or no type inference	unreconstructed control flow statements, unreconstructed string concatenation, unused labels, no type inference
2	incorrect	incorrect	easy to correct syntax errors which produce a semantically correct program	boolean typed as int, missing variable declaration
3	incorrect	incorrect	difficult (but possible) to correct syntax errors which produce a semantically correct program	code with goto statements
4	incorrect	incorrect	very difficult (or nearly impossible) to correct syntax errors required to produce a semantically correct program	invalid variable use, obviously incorrect code, massive source re-write required
5	incorrect	correct	easy to correct semantic errors which produce a semantically correct program	missing typecasts
6	incorrect	correct	difficult (but possible) to correct semantic errors which produce a semantically correct program	incorrect control flow
7	incorrect	correct	very difficult (or nearly impossible) to correct semantic errors required to produce a semantically correct program	incorrectly nested try-catch blocks, massive source re-write required
8	incorrect	incorrect	incomplete decompilation	missing large sections of source, missing inner classes
9	Fail	Fail	decompiler fails upon execution/produces no source output	decompiler fails to parse arbitrary bytecode

Figure 2.13: Decompilation correctness classification

		ClassCracker3	Dava	JAD	Java Decompiler	jdec	JODE	jreversepro	Mocha	SourceAgain	SourceTec
javac	Fibonacci	8	0	0	0	0	0	9	9	0	9
	Casting	8	5	5	5	5	0	5	9	5	9
	InnerClass	8	8	2	1	8	9	8	9	8	9
	TypeInference	8	2	1	4	4	0	9	9	1	9
	TryFinally	8	4	8	0	0	4	8	9	4	9
	ControlFlow	8	1	1	2	4	1	8	9	1	9
arbitrary	Exceptions	8	8	4	4	8	9	9	9	7	9
	Optimised	8	2	4	3	4	2	3	9	3	9
	VariabeReUse	8	1	2	0	3	0	2	2	0	2
	Ada	8	3	9	4	8	9	8	4	3	4

Figure 2.14: Decompiler Test Results



(a) Decompiler Effectiveness

(b) Correct Decompilations

performing slightly better at decompiling the arbitrary bytecode programs. JODE was able to decompile one more program correctly than Java Decompiler. JODE was the best decompiler in the original survey and is still one of the best in our evaluation but Java Decompiler beats JODE with our effectiveness measures and only decompiles one less program correctly than JODE.

Every decompiler that could parse the latest Java class file format could decompile the Fibonacci test program. The decompilers showed varying degrees of success with the other programs; we discuss the results here.

Figure 2.14, page 49 shows a table of raw results detailing our effectivess score given to each decompiler for each program. Figures 2.15(a) and 2.15(b), page 49 show the the decompiler effectiveness scores. The charts show percentages of effectiveness with higher percentage meaning a higher effectiveness score, calculated from table 2.14, page 49.

2.5 Discussion of Results

In this section we discuss the results of the tests, explaining the output of the decompilers when they failed to decompile a program. Program listings of the decompiled programs are in appendix A, beginning on page 113. Some decompilers, such as Mocha and SourceTec, could not parse the latest Java class file versions.

2.5.1 ClassCracker3

ClassCracker3 did not decompile any of our test programs completely with just the method signatures in the resulting Java source file. The original survey found a similar result and the author concluded that the decompiler 'needs some work to cater for the tricky cases covered in these tests' [55].

2.5.2 Dava

Dava, being a decompiler aimed at arbitrary bytecode, performed better than most other decompilers in tests with class files that were not generated by *javac*. However, for the others it did not perform as well. It could not decompile the trivial, *TryFinally* program which most other decompilers could. Dava attempts to reconstruct the program's control flow whereas other decompilers recognise the pattern of a try-finally block. Dava perfectly decompiles the Exception test program which is unsurprising as this test program is from the creators of Dava. Interestingly, Dava was unable to correctly decompile the Sable test program, which was created originally to show that Dava outperforms other decompilers, due to a failure to insert a simple typecast for an argument in a method invocation. However, the main problem that the Sable test program is designed to show, type inference of a specific variable, was performed correctly.

Casting

The decompiled casting test program (Listing A.2, page 113) indicates that Dava was unable to detect the need to insert a cast, and therefore Dava produced a semantically incorrect program. The decompiled program iterates through the characters as in the original program (except using Unicode instead of ASCII) but in the *System.out.println* invocation there is no cast inserted. The output of the program, when executed, is therefore two lists of characters rather than the original ASCII code followed by ASCII character.

Args

Dava correctly decompiled the Args test program (Listing A.3, page 113) but typed the int local variable as byte, and inserted two unnecessary typecasts. We therefore classify this program as semantically equivalent but with 'untidy' source code.

ControlFlow

Dava produces a semantically equivalent ControlFlow program (Listing A.4, page 114) which is slightly different from the original program. Dava's program contains a *labelled while loop*, and uses *labelled continue* statements to continue iterating from two points: inside the try section and inside the catch section. This version of the program has only one loop and uses a *labelled continue* to mimic the inner loop in the original program (Listing 2.21, page 41). Both Java class files contain the same bytecode sequences though they are generated from different Java source (Listing 2.22, page 42).

Sable

Dava correctly performs type inference on the Sable test program (Listing 2.14, page 37) but unfortunately fails to insert a typecast in the method invocation for method f, which results in a syntactically incorrect Java program (Listing A.5, page 115). The main problem, type inference for variable d, was solved correctly and inserting a type cast results in a semantically and syntactically correct program.

Usa

Dava failed to correctly decompile the Usa test program resulting in a class file with out the inner classes (Listing A.6, page 115). The resulting program contains none of the inner classes present in the original program (Listing 2.13, page 36).

Exceptions

Dava fails to correctly decompile the Exceptions test program, producing an incomplete program (Listing A.1, page 113). This is surprising because the program was designed by the creators of Dava to demonstrate their decompiler [116]. The previous version of Dava, 2.3.0, could decompile the program correctly and the result decreased the effectiveness score in this version.

TryFinally

The decompiled TryFinally program (Listing A.7, page 116) is interesting as it decompiles a nearly syntactically, and somewhat semantically correct program. The decompiled program contains a small syntactic error (variable $\$r5$ was originally $\$r4$, which had already been declared) which when corrected produces a compiled Java program which produces an equivalent output to the original program. Even though both programs produce the same output they are not identical. The decompiled program does not make use of a try-finally block, and instead uses a mix of try-catch blocks and a labelled while-loop and a labelled continue statement. If re-compiled the program does not produce the same bytecode as the original program.

Optimised

The optimised program is incorrect as it is missing a typecast; this is the same problem as the type inference and casting test programs.

connectfour

Dava failed to correctly decompile the connectfour test program with several syntactic and semantic errors. Listing A.9, page 114 shows an extract from the decompiled program where a *boolean* is compared with an *int*.

2.5.3 Jad

Jad produced some pleasing results, with a similar overall score to Dava. Jad performs best with *javac* generated code and fails to correctly decompile arbitrary bytecode. Jad also fails the trivial TryFinally test program which is due to Jad being outdated - finally blocks used to be implemented using subroutines but *javac* no longer generates subroutines and instead in-lines finally blocks. Jad correctly decompiles three *javac* generated class files whereas Dava only correctly decompiles two of these, which demonstrates the difference between between the two types of decompiler. Jad is comparable to JODE and Dava and overall performs very similarly.

Sable

Jad produced a semantically equivalent Java program (Listing A.10, page 118) but did not perform type inference. Variable *d* (*Obj* in the decompiled program) is typed as *Object*, and an explicit typecast is inserted at the point where the *draw*() method is invoked. We therefore classify this program as category 1 - semantically and syntactically correct, but no type inference was performed.

ControlFlow

Jad decompiles the ControlFlow program correctly (Listing A.11, page 118) though produces slightly different Java source than the original. The program uses a for-loop instead of a while loop for the inner loop and a do-while loop instead of a standard while loop for the outer loop.

Casting

The casting program is missing a typecast - the same problem that Dava had with this program.

Usa

The Usa program is syntactically incorrect as it includes a program statement before a *super*() call to the parents constructor. In Java, the *super*() call must be the first statement in a constructor.

Exceptions

Jad fails to correctly decompile the Exceptions test program, producing a syntactically incorrect program (Listing A.14, page 120). The program contains some illegal Java code, including *goto* statements and the word '*this*'. The program also contains only one try-catch block which includes 'c' and 'd' in the try clause and 'e' in the catch clause. The program is syntactically incorrect and not equivalent to the original.

Optimised

The Optimised test program is incorrectly decompiled by Jad. The program (Listing A.15, page 121) contains only one local variable in the f method compared with 4 in the original. The program also contains some bytecode instructions where Jad could not determine how the objects are constructed using the optimisations. Jad was expecting *javac* generated code so could not deal with the unexpected optimised sequence for object construction.

TryFinally

Jad produces a syntactically incorrect TryFinally program (Listing A.16, page 121). The decompiled program contains illegal Java code as it had trouble analysing the control flow of the program.

Args

Jad produces a syntactically incorrect Java program (Listing A.17, page 122). Jad attempts to assign an integer to an array of String objects, which is syntactically incorrect.

connectfour

Jad failed to parse the connectfour test program.

2.5.4 Java Decompiler

Java Decompiler is a newer decompiler which outperforms all other decompilers in terms of our effectiveness measures. The reason which Java Decompiler out performs Dava is that it is correctly able to decompile the TryFinally and Usa test programs, which are both *javac* generated. Java Decompiler has trouble decompiling arbitrary bytecode but does this better than all other decompilers, including Dava.

We originally evaluated version 0.2.7; our updated evaluation uses 0.3.2. There are some improvements since 0.2.7 but some decompilations are worse. However, the latest version is overall a slight improvement.

Casting

Java Decompiler produces a semantically incorrect program (Listing A.19, page 123) in the same way that Dava does (Listing A.2, page 113). The decompiled program loop iterates through the characters as in the original program (except using unicode instead of ASCII) but in the *System.out.println* invocation there is no cast inserted. The output of the program, when executed, is therefore two lists of characters rather than the original ASCII code followed by ASCII character

Usa

The Usa program is semantically correct but contains some redundant code (Listing A.20, page 123).

Sable

Java Decompiler produces a syntactically incorrect program, with the same problem as in the Casting test program - failure to insert a typecast. Dava also failed to insert the same typecast. Further interesting syntactic errors include object instantiations that are spread over two lines where the first uses the *new* keyword along with the class name. Java Decompiler attempts to call the object's constructor separately, on the next line. An object's constructor method is known by the special name `<init>` in bytecode and this is what Java Decompiler attempts to invoke. The program also contains too few variables and an attempt is made to assign a *boolean* to a *short*.

ControlFlow

Java Decompiler produces a semantically incorrect program (Listing A.22, page 124) containing an extra break statement and some labels. If the syntactic errors are removed the program is almost semantically correct.

Exceptions

Java Decompiler fails to correctly decompile the Exceptions test program resulting in a syntactically incorrect program (Listing A.23, page 125). The program uses the word 'this' as a variable name which is a reserved word in Java. Correcting these syntactic errors leads to a semantically incorrect program (see Figure 2.12, page 125).

Optimised

Java Decompiler produces a syntactically incorrect program with some interesting syntactic errors (Listing A.24, page 125) in the same way as the TypeInference test program.

Args

Java Decompiler produces a semantically correct Java program (Listing A.25, page 125).

connectfour

Java Decompiler produces a syntactically incorrect connectfour program with several syntactic errors. One such error is similar to the optimised test program which suggests the Ada-to-Java compiler uses an optimised form of object instantiation (Listing A.26, page 123). Many of the methods which throw exceptions are missing the class name of the exception (Listing A.27, page 123).

2.5.5 jdec

jdec is another *javac* orientated decompiler but does not perform as well as the other newer decompilers. Java Decompiler can correctly decompile inner classes while jdec cannot. jdec also cannot decompile arbitrary bytecode correctly.

Casting

jdec decompiles the Casting test program with the same semantic error (Listing A.28, page 126) as other decompilers such as Dava (Listing A.2, page 113) - the crucial cast from *char* to *int* is missing. The decompiled program loop iterates through the characters as in the original program but in the *System.out.println* invocation there is no cast inserted. The output of the program, when executed, is therefore two lists of characters rather than the original ASCII code followed by ASCII character

Usa

jdec fails to decompile the Usa test program, decompiling only the main class (Listing A.29, page 127). Dava also has this problem.

Sable

jdec produces a syntactically and semantically incorrect test program (Listing A.30, page 128). This program contains two *Rectangle* objects rather than the one *Rectangle* original and one *Drawable* object in the original.

ControlFlow

jdec produces a syntactically incorrect control flow test program with a *catch* clause intersecting a *else* clause which, if corrected produces a semantically incorrect program (Listing A.31, page 129).

Exceptions

jdec only partially decompiles the exceptions test program, with blocks 'd', 'e' and 'f' missing. The program is syntactically and semantically incorrect (Listing A.32, page 130).

Optimised

jdec produces a syntactically and semantically incorrect optimised program, with several syntactic errors (Listing A.33, page 131). The program contains variable assignment statements within constructor invocation parameters and uses the variable *this* in many places. The method is static so should contain no use of the *this* keyword. The method also contains no arguments which is strange because jdec introduced a variable called $arg0$.

Args

jdec produces a syntactically incorrect program (Listing A.34, page 132), in a similar way to Jad by attempting to assign an integer to a string array. jdec also attempts to re-declare the method argument.

connectfour

jdec produces a seemingly incomplete, and syntactically incorrect program with many empty blocks (for example, listing A.37, page 133) and syntax errors. The program has several examples of missing commas within method parameter lists (for example, listing A.36, page 127) and assignment statements within array declarations (for example, listing A.35, page 133).

2.5.6 JODE

JODE has a similar overall result to Dava and Jad but is beaten using our effectiveness measures by Java Decompiler. However, Java Decompiler and Jad decompile fewer test programs completely correct than JODE. Surprisingly JODE is unable to correctly decompile the TryFinally test program. JODE was the best decompiler in the original survey and is still one of the best in our evaluation.

TryFinally

JODE's decompilation of the TryFinally test program (Listing A.38, page 133) fails to include a try-finally block but instead includes a try-catch block and an incorrectly typed exception variable. The catch clause includes an exception variable typed as *Object* but such a variable must be typed *Exception* or a sub-class of *Exception*. This makes the decompiled program both syntactically and semantically incorrect. The finally clause code is duplicated after the try-catch block, and if the syntactic error is corrected the program produces the same output as the original. However, this is not the same as the original program which uses a try-finally block rather than a try-catch block.

Usa

JODE could not parse the inner class test program, exiting with an exception.

ControlFlow

JODE produces a semantically correct decompilation (Listing A.39, page 134) of the control flow test program (Listing 2.21, page 41). However, the source is more obtuse than the original. JODE's decompilation uses for-loops instead of while loops which, while semantically equivalent, do not look as user-friendly as the original. In this version of the program there is no *continue* statement in the catch clause but instead a break is used in the try clause. Control flows from the catch clause to the beginning of the loop as there is no break clause.

Exceptions

JODE could not decompile the exceptions test program, exiting with an exception regarding the intersecting try-catch blocks (Listing A.41, page 134).

Optimised

JODE has a slight problem with object construction in the optimised test program, similar to other decompilers such as Java Decompiler. If the constructor problem is corrected the program is semantically correct, including the correctly typed variable (Listing A.41, page 134).

connectfour

JODE could not parse the Ada test program and exits with an exception (Listing A.42, page 135), due to the arbitrary nature of the bytecode instructions.

2.5.7 jReversePro

jReversePro performed badly in many of the tests and was unable to decompile the test programs correctly, as it can not parse the latest class file format. In fact, jReversePro failed to parse many of the test programs and only partially decompiled others. The remaining programs produced were semantically incorrect.

Fibo

jReversePro is unable to decompile the trivial Fibo test program as it could not parse the latest class file format (Listing A.43, page 135).

Casting

jReversePro was able to parse the simpler casting test program but produced a semantically incorrect program (Listing A.44, page 136). The program fails to include the crucial cast in the same way as other decompilers such as Jad and Java Decompiler but also contains an infinite loop due to the inclusion of an extra variable.

Usa

The inner class program, as decompiled by jReversePro, contains none of the inner classes (Listing A.45, page 136).

Sable

jReversePro is unable to parse the type inference test program and exits with an exception.

TryFinally

jReversePro does not fully decompile the try finally test program as it leaves out the most important part: the try-finally block.

Exceptions

jReversePro exits with an exception due to intersecting try-catch blocks (Listing A.47, page 137), like JODE.

Optimised

jReversePro was able to parse the optimised program, most likely because the optimiser output an earlier version class file format, but produced a syntactically incorrect program (Listing A.48, page 138). The program contains several syntactic errors, included attempting to assign a boolean to an int and constructor problems.

Args

jReversePro incorrectly decompiled the variable re-use test program producing a syntactically incorrect program (Listing A.49, page 139), similar to Jad and Java Decompiler.

connectfour

jReversePro produced a syntactically incorrect and incomplete connectfour test program. jReversePro has many blocks of missing code (Listing A.50, page 133), like jdec (Listing A.37, page 133) and contains many syntactic errors such as incomplete if-then-else blocks appearing within loops (Listing A.51, page 133).

2.5.8 Mocha

Mocha is an obsolete decompiler which never made it past a beta version, though we include it here as it is still available. The results from Mocha are not surprising as they confirm that Mocha is no longer a viable decompiler for the latest Java class files. Mocha fails to parse all programs generated by the latest version of *javac*, and also fails to parse two of the arbitrary test programs.

Args

Mocha fails to produce a syntactically correct program, producing a similar program to other decompilers such as jReversePro and Jad, which attempts to assign an int to an array of strings (Listing A.52, page 139).

Exceptions

Mocha could not decompile the exceptions test program and exits with a 'Method Exceptions: Flow analysis could not complete' error message, due to the intersecting try-catch blocks.

connectfour

Mocha's decompilation of the connectfour test program contains many syntactic errors (for example, listing A.53, page 133), and some bytecode instructions are left in place (for example, listing A.55, page 140 and listing A.54, page 135).

2.5.9 SourceAgain

SourceAgain is the only commercial decompiler that performed well in our tests. SourceAgain is able to perfectly decompile four of our test programs including the Args test program which is only decompiled correctly by JODE and Dava. However, the product is no longer sold or supported and is only available online to decompiler single class files. SourceAgain is therefore obsolete and not useful for any real-world decompilation tasks.

Casting

SourceAgain, like other decompilers such as Jad, fails to insert the crucial cast resulting in a semantically incorrect program (Listing A.56, page 140).

Usa

SourceAgain fails to correctly decompile (Listing A.45, page 136) the inner class test program but unlike other decompilers, such as jReversePro, this is because the decompiler is web-based and only accepts single class files - the inner classes could not be uploaded along with the main class. The original survey [55] used the professional version of the decompiler which was able to correctly decompile inner classes; this is no longer available.

Sable

SourceAgain produces a semantically equivalent type inference program but could not correctly type the variable *d* (Listing A.58, page 141). The online decompiler warns that it could not determine the inheritance relationship between the objects as a result of the restriction of decompiling single class files. In the original survey [55] the professional version of SourceAgain did not correctly type the variable.

TryFinally

SourceAgain produces a syntactically incorrect try finally test program containing two try-finally blocks, duplicated finally clause code and an undeclared variable (Listing A.59, page 141).

ControlFlow

SourceAgain correctly decompiles the control flow program producing a semanticaly equivalent, though some-what obtuse, program (Listing A.60, page 142). SourceAgain produces a very similar output to Dava (Listing A.4, page 114) which is slightly different from the original program. Both Java class files contain the same bytecode sequences though they are generated from different Java source (Listing 2.22, page 42).

Exceptions

SourceAgain produces a semantically incorrect exceptions test program (Listing A.61, page 142) which contains all the necessary blocks, unlike other decompilers, but is not semantically equivalent to the original. For example, block 'd' should be contained within a catch clause, leading to block 'e' if an exception is thrown.

Optimised

SourceAgain produces a syntactically incorrect program (Listing A.62, page 143) containing, like other decompilers, object initialisation and constructor problems. SourceAgain also includes self assignments which aren't necessary.

connectfour

SourceAgain produces a syntactically incorrect connectfour test program but with fewer errors than other decompilers. If problems such as misnamed *this* variables (Listing A.63, page 140) and multiple variable declarations (Listing A.64, page 140) are removed the program becomes compilable and semantically correct.

2.5.10 SourceTec (Jasmine)

SourceTec (Jasmine), a patch to Mocha, also fails to parse all *javac* generated class files producing the same results as Mocha.

2.6 Conclusion and Future Work

Many of the companies producing commercial decompilers have disappeared and their decompilers have been left unmaintained. Even some free and/or open-source decompilers such as Jad and JODE have been unmaintained for some time. Jad is not open-source so the project cannot be taken up by others and the last major update was in 2001.

Decompilation has many uses in the real world, such as the recovery of lost source code for a crucial application [56], therefore if the quality of Java decompilers increased they might be of more use commercially.

One of the most active decompiler projects is the open-source Dava [114, 128, 127, 115, 116] decompiler, part of the Soot Optimisation Framework [164], which is a research project carried out by the Sable Research Group at McGill University. Dava differs from other decompilers in that it aims to decompile arbitrary bytecode whereas other decompilers rely on known patterns produced by Java compilers (and this is usually *javac*). Dava is better at decompiling arbitrary bytecode whereas other decompilers are better at decompiling *javac* generated bytecode. However, Java Decompiler was just as good at decompiling arbitrary bytecode according to our effectivess score. Java Decompiler is aimed at decompiling *javac* generated bytecode.

A decompiler aimed at decompiling arbitrary bytecode, like Dava, can be more useful in some instances than a decompiler aimed at bytecode generated by a specific compiler. Java bytecode can be generated by tools other than a Java decompiler and many decompilers are aimed at patterns produced by Java decompilers and some specifically *javac*. Knowing the patterns that a compiler will produce makes decompilation of bytecode easier and it can sometimes be just a matter of reversing those patterns.

Decompiling bytecode arbitrarily, i.e. not by inverting known patterns produced by compilers, can be a disadvantage in some cases, for example Dava could not correctly decompile the trivial TryFinally test program. Other decompilers could decompile this test program by finding a known pattern produced by a compiler for try-finally blocks.

Though there is a lack of commercial Java decompilers Java bytecode decompilation and decompilation in general are fruitful research areas. One of the main areas for research is type inference both in bytecode (e.g. [13, 115, 92]) and machine code (e.g. [122]). The task of type inference in Java bytecode is simpler than that of machine code due to the information contained within a Java class file - a Java class file contains type information for fields and method parameters and returns.

Type inference is an interesting problem in decompilation and two of the best decompilers tested (Dava and JODE) were both able to correctly type the variables in the type inference test. Most other decompilers, which did not perform type inference, typed variables as *Object* and inserted a typecast where necessary. The type inference problem is NP-Hard in the worst case [63], however, if the type inference algorithm is optimised for the common-case rather than the worst case it is possible to perform type analysis efficiently for most real-world code as worst-case scenarios are unlikely [13].

All the decompilers tested had some problems decompiling some of the tests. In terms of our effectiveness measures, Dava, Jad, Java Decompiler and JODE were the four best decompilers (excluding SourceAgain). Of these, JODE and Java Decompiler are the best decompilers: JODE correctly decompiles 5 out of the 10 test programs correctly and Java Decompiler performs best using our effectiveness measures but decompiles one less program correctly. JODE is open-source and is therefore open to further improvements but is not currently maintained, while Java Decompiler is in development and available free (but is not open-source). Unfortunately Jad is unmaintained and so is not guaranteed to work for future versions of Java class files. The commercial decompiler SourceAgain performs well in the effectiveness measures, but was only able to decompile 4 programs correctly. SourceAgain performed similarly to Dava but is now obsolete and only available as a web application which can decompile single class files. We therefore exclude SourceAgain from our top decompilers, as it is not generally useful for real-world programs - for example, it can't decompile inner classes.

Java Decompiler is a newer decompiler in active development, which performs highest in our effectiveness measures and correctly decompiles 4 out of 10 programs. This decompiler performs best at *javac* generated bytecode and may improve in the future even more as it is in development.

Knowing the tool that generated a class file can be useful in knowing which decompiler to use. If a class file was generated by *javac* then a *javac* specific decompiler would be more useful than an arbitrary decompiler such as Dava. If the class file was generated by other means, or modified by an obfuscator or optimiser, a *javac* specific decompiler would most likely fail so an arbitrary decompiler would be more

useful in this case.

It is interesting that some decompilers are better at certain things; this indicates that good decompilation is theoretically possible. Further work will include investigating the possibility of implementing a better decompiler, combining the best techniques from various decompilers.

We have demonstrated the effectiveness of several Java decompilers on a small set of test programs, each of which were designed to test different problem areas in decompilation. Such a small test set of programs may not be representative of real-world Java programs and, in fact, some problem areas tested may not be of high relevance in real-world programs.

We proposed a system to quantify the effectiveness of a decompiler on our set of test programs, however the analysis is some-what subjective and could be further formalised. This would enable us to generalise our effectiveness measures, as the effectiveness values assigned to our test programs may not be applicable in the context of other programs. We encourage the reader to study decompiler output in appendix A.

In terms of our evaluation, Dava, Java Decompiler and JODE are the best decompilers. Dava faces some challenges in decompilation of Java specific code while the other decompilers have problems with arbitrary bytecode.

We have shown that, though not perfect, decompilers are a real threat to distributed Java bytecode class files. Software companies would need to introduce technical measures to provide protection for their intellectual property. One such method is software watermarking which does not attempt to stop software theft but instead deters attackers by providing a method of proving ownership of copied software. The next chapter presents a survey of static software watermarking techniques.

Chapter 3

A Survey of Static Software Watermarking

Software watermarks can be broadly divided into two categories: static and dynamic [30]. The former embeds the watermark in the data and/or code of the program, while the latter embeds the watermark in a data structure built at runtime.

Figure 3.1 shows a very simple static watermarking system - The watermark W, the input program P and an optional key K are fed into the embedder. If a key is used then the watermark is encrypted. The embedder outputs a semantically equivalent program $P^{'}$ containing an extra field; this field contains the watermark. A recogniser takes the watermarked program $P^{'}$, looks for the watermark field and outputs the original watermark W.

Figure 3.2 shows a very simple dynamic watermarking system - The watermark W, the input program P and a key K are fed into the embedder. The key, in this case, is a secret input to the program. This input causes a specific path through the program execution to be taken. The embedder outputs a new program $P^{'}$ which, when executed with the secret input, generates the watermark. A recogniser inspects the program $P^{'}$ while running, looking for value of the watermark variable and outputs the original watermark W.

In comparison to a static system, a dynamic watermarking system needs to execute the program in order to retrieve the watermark. The watermark is stored in the semantics of the program rather than the syntax. Dynamic watermarks should, in theory, be resilient to semantics-preserving transformations. Software watermarking algorithms can be further categorised based on the method of embedding the watermark.

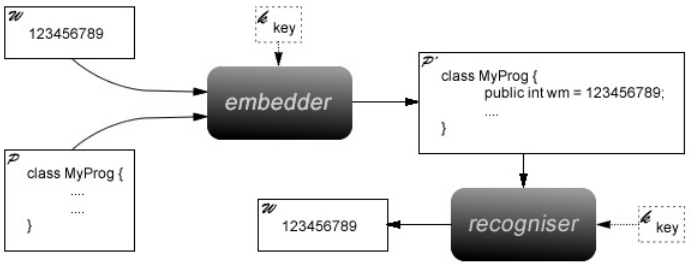


Figure 3.1: Simple Static Watermarking System

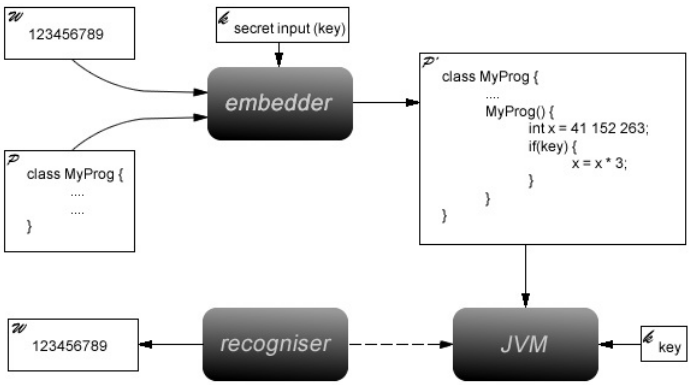


Figure 3.2: Simple Dynamic Watermarking System

Some of these categories include:

Abstract interpretation algorithms are static analysis based in which watermarking recovery is handled by an abstract interpretation of the code semantics [38].

Basic block re-ordering algorithms encode a watermark by re-ordering the basic blocks of a program and inserting jump statements to preserve the original control flow. One of the earliest software watermarking algorithms [44] uses this technique.

Code replacement algorithms encode a watermark by replacing parts of a program, for example by replacing certain op-codes with equivalent instructions. For example, Hydan, defines sets of functionally-equivalent instructions and encodes information in machine code by using the appropriate instructions from each set [52].

Constraint based algorithms encode the watermark in a set of extra constraints to the original algorithmic problem. The solution to this new problem will therefore satisfy the original problem and the watermark constraints [142].

Dynamic pat h algorithms encode watermarks based on the dynamic branching behaviour of programs by adding branches to program code [34].

Graph based algorithms encode the watermark in a graph structure and embed the graph structure in a program. Venkatesan *et al.* [167] describe a static graph watermarking system in which the watermark graph is merged with the program control flow graph.

Opaque predicate algorithms rely on being enclosed by a predicate whose outcome is known *a priori*. It is difficult for automated software analysis to know the value of the predicate; therefore it is not know whether the enclosed code (which may or may not be a watermark) could be removed [32].

Spread-spectrum algorithms were originally developed for multimedia watermarking [39]; in software the idea is to spread the watermark across a program by modifying instruction frequencies [159]

Thread based algorithms encode watermarks within distinctive behaviour of threads added to programs [130].

Some of these could be implemented statically or dynamically, for example there are static [167] and dynamic [30] graph watermarking algorithms.

3.1 Register Allocation Based Watermarks

Register allocation based watermarking algorithms are constraint-based static watermarking techniques. Figure 3.7 shows the evolution of this family of algorithms on which we report previous findings, describe some recent additions (including a correction to a published algorithm) and conclude by suggesting a direction for future work.

Change this to something better

3.1.1 Background

Keep background?

Graph Colouring

In graph theory, the simplest form of graph colouring is a way of colouring the vertices of a graph such that no two adjacent vertices share the same colour. The graph colouring problem is NP-complete[89].

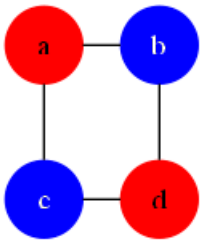


Figure 3.3: A graph with 4 vertices and 2 colours

Consider the simple graph in figure 3.3; the graph contains four vertices and four edges. Vertex a is adjacent to vertices b and c therefore if a is red then b and c cannot be. Vertex d, on the other hand, is not adjacent to a and can therefore also be coloured red. Vertices b and c are therefore coloured blue. The smallest number of colours needed to colour this graph is 2. This is known as the *chromatic number*.

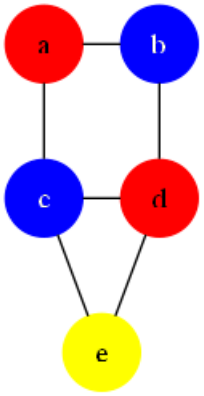


Figure 3.4: Example graph with 5 vertices and 3 colours

If we add another vertex, e, adjacent to c and d we must colour this with a third colour; e cannot be blue as it is adjacent to c and it cannot be red as it is adjacent to d (figure 3.4). The chromatic number of graph 2 is therefore 3.

Graph colouring has many real-world applications including register allocation in compilers [**?**].

Register Allocation

Register allocation is the process of assigning program variables to a finite number CPU registers [4]. Programmers can use any number of variables in their programs but there are a limited number of CPU registers to actually store the values of those variables.

Listing 3.1: 'Example Pseudocode'

```
v1 := 1 + 1;
v2 := 2 + 5;
v3 := 3 + v1;
v4 := v1 + v2;
v5 := v2 + v3;
```

An interference graph is used to model the relationship between the variables in a program method. Each vertex in the graph represents a variable and an edge between two variables indicates that their live ranges overlap. Simple put, a variable is live if it has been computed and will be used before being recomputed. We colour the graph in order to minimise the number of registers required and ensure that two live variables do not share a register.

Definition 1. *Variable liveness [20]: A program variable is considered live at point L in a program P if there is a control flow path from the entry point of P to a definition of X and then through L to a use of X at point U, which has the property that there is no redefinition of X on the path between L and the use of X at U.*

Consider the example pseudocode in listing 3.1 and it's interference graph in 3.5. From the pseudocode we can see that variable v5 is not live at the same time as any other variable and is therefore unconnected in the graph. Variable v1, on the other hand, is live at the same time v2 and v3 are live; therefore they are connected in the interference graph.

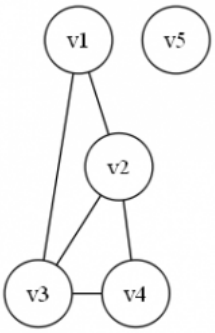


Figure 3.5: Example interference graph for listing 3.1

The graph in figure 3.5 can be coloured, as shown in figure 3.6, using 3 colours. This means that the minimum number of registers needed to store the variables in the sample program is 3. Variables v1, v4 and v5 can use the same register because they are not live at the same time.

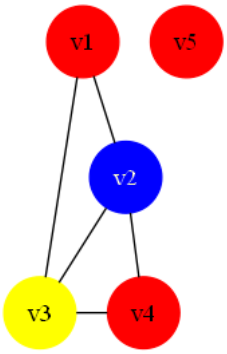


Figure 3.6: Example coloured interference graph for listing 3.1

The sample program could be converted into pseudo-assembly code, shown below, where *add X, Y,*

Listing 3.2: 'Example Pseudo-Assembly code'

```
add r1, 1, 1
add r2, 2, 5
add r3, 3, r1
add r1, r1, r2
add r1, r2, r3
```

Z adds together Y and Z and stores the result in X. Y or Z could be a literal value or the value in a register.

We can see that r1, r2 and r3 correspond to the three colours in the interference graph.

Register Allocation in Java Byte Code

The Java virtual machine contains no registers but each method has access to a local variable table. The compiler has to allocate each Java method's variables to local variable slots. Watermarking by register allocation can be implemented by changing the uses of the local variable slots. A watermark could be included in each method in a Java program, or the watermark could be split and spread throughout the program.

Expand on this?

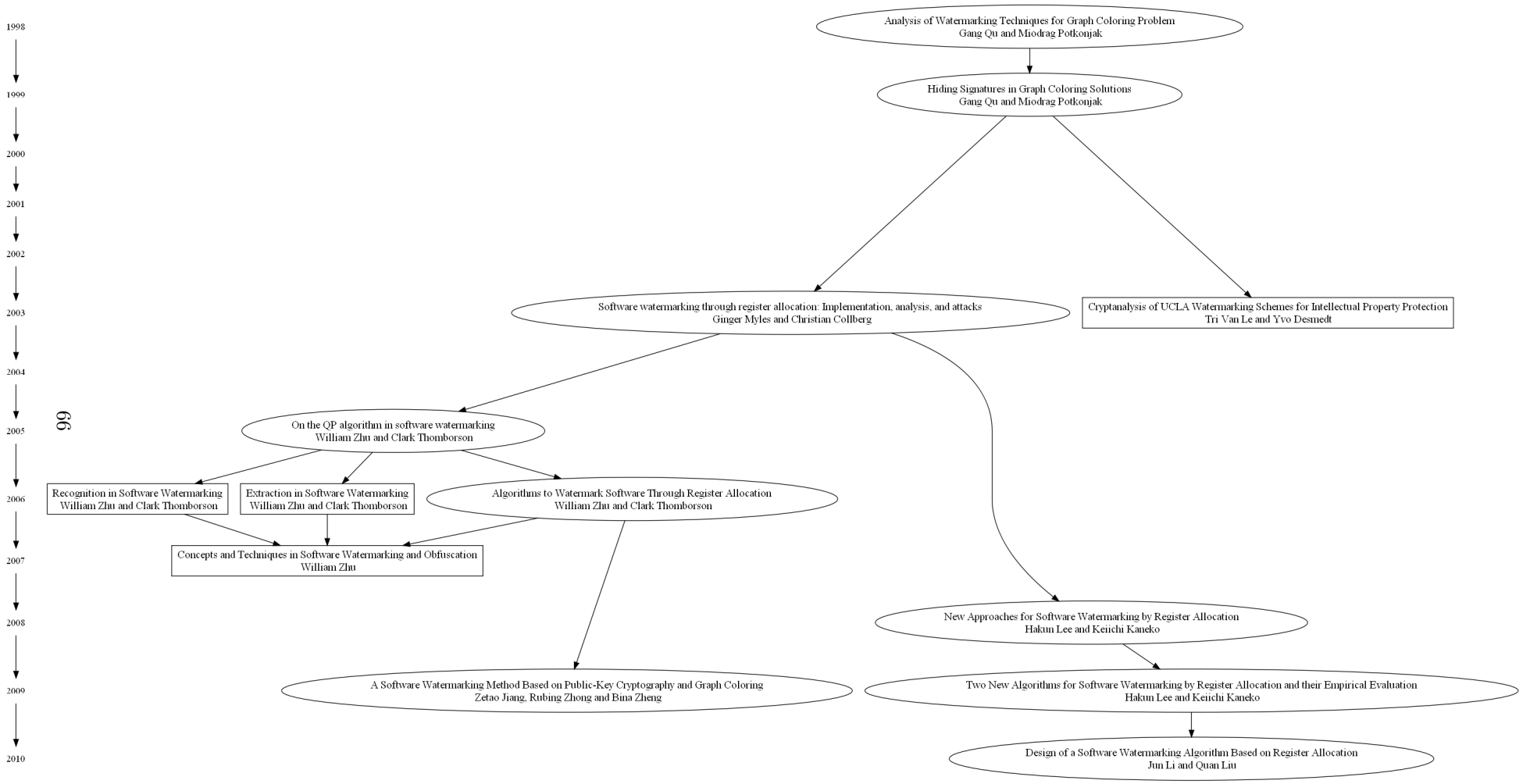


Figure 3.7: Evolution of Register Allocation Based Software Watermarking. Oval shapes represent new algorithms, while rectangular shapes represent evaluations.

3.1.2 The QP Algorithm

The QP algorithm [141] is a constraint-based watermarking algorithm based on the concept of graph colouring. In the QP algorithm edges are added to the graph based on the value of the watermark. When edges are added to an interference graph the vertices that become connected must be re-coloured - and they cannot be assigned the same registers. In other words, we add a new constraint to the problem (the extra edges) and when we compute the graph colouring we have a solution which solves the graph colour problem and one which also includes the watermark.

The QP algorithm was proprosed to embed watermarks in any graph colouring solution and can be applied to graph colouring for register allocation to embed watermarks in software.

The first step in the QP algorithm is to convert the message into binary, for example convert a string into binary by using ASCII codes. The next step is to add additional edges to the interference graph, in such a way that the extra edges encode the watermark. The QP algorithm requires that the vertices of the graph are indexed and relies on the ordering of such indices for embedding and extraction. The originally published QP algorithm contained a minor problem which assumed that every vertex in an arbitrary graph could contain one bit of information; Zhu *et al.* proposed a clarified version of the algorithm [174] shown in algorithm 1.

Algorithm 1 Clarified QP embedding algorithm

Input: a graph $G(V, E)$, a message $M = m_0 m_1 \ldots$
Output: a graph $G'(V', E')$ with embedded message M

copy $G(V, E)$ to $G'(V', E')$
j = 0
for each bit in message as m_i **do**
  if v_i has two candidate vertices v_{i1}, v_{i2} **then**
    j++
    if $m_j = 0$ **then**
      add edge (v_i, v_{i1}) to E'
    else
      add edge (v_i, v_{i2}) to E'
    end if
  end if
end for
return $G'(V', E')$

Algorithm 2 QP extraction algorithm

Input: a graph $G(V, E)$ with embedded message M
Output: a message $M = m_0 m_1 \ldots$

copy $G(V, E)$ to $G'(V', E')$
for each pair of unconnected vertices v_i, v_j **do**
  n = number of vertices not connected to v_i between v_i and v_j
  if n = 0 **then**
    $M = M + 0$
  else if n = 1 **then**
    $M = M + 1$
  else
    n = number of vertices not connected to v_i between v_j and v_i
    if n = 0 **then**
      $M = M + 0$
    else if n = 1 **then**
      $M = M + 1$
    else
      message bit undefined
    end if
  end if
end for
return M

Definition 2. *QP candidate vertices [141]: The nearest two vertices v_{i1} and v_{i2} which are not connected to v_i; $i_2 > i_1 > i$ (mod n) and $(v_i, v_{i1}), (v_i, v_{i2}) \notin E$ and $(v_i, v_j) \in E$ for all $i < j < i_1, i_1 < j < i_2$*

Figure 3.8 shows an example intereference graph for a program (it is not important which program, in these examples). In order to embed the watermark 1011011010_2 in the interference graph we use algorithm 1. Figure 3.9 shows the interference graph with dotted watermark edges; for clarity, these lines are labeled with the watermark value.

For each message bit we take the vertex with the corresponding index and look at the next two vertices which are not connected; if the message bit is 0 we add an edge between the current vertex and the next unconnected vertex, otherwise we add an edge to the second unconnected vertex.

For example, the first message bit is 1. Vertex v_0 is connected to vertices v_1 and v_2 therefore the next two unconnected vertices are v_3 and v_4. The message bit to embed is 1 so we add an edge between

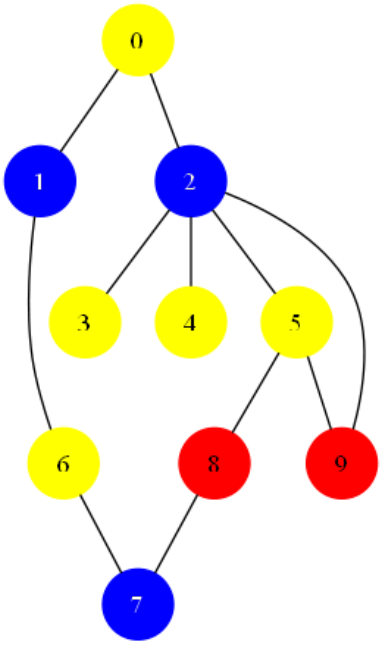


Figure 3.8: Example interference graph

v_0 and v_4. The next two unconnected vertices after v_9 are v_0 and v_1; we add an edge between v_9 and v_0 because the watermark bit is 0.

The chromatic number of the original interference graph (figure 3.8) is 3, while the chromatic number of the watermarked graph is 4.

In order to extract the secret message we consider all pairs of coloured vertices which do not have edges between them. For each pair v_i, v_j we count how many vertices there are with indices between i and j which are not connected to v_i.

Definition 3. *Value of the watermark bit in the QP extraction algorithm [142]:*

1. *If $n(i,j) = 0$, the watermark bit is 0*
2. *if $n(i,j) = 1$, the watermark bit is 1*
3. *otherwise, if $n(j,i) = 0$, the watermark bit is 0; if $n(j,i) = 1$, the watermark bit is 1; else the watermark bit is undefined.*

where $n(i,j)$ is the number of vertices between v_i and v_j which are not connected to v_i.

For example, consider our watermarked graph in figure 3.9, in order to extract the watermark we would have to consider only the colours; the dotted edge information would be unavailable to us in practise. Between the pair (v_0, v_4) there is one vertex, v_3, which is unconnected to v_0; thus the message bit is 1.

Qu and Potkonjak include two further embedding algorithms: selecting a maximal independent set (MIS) and adding vertices & edges. They also perform an experimental evaluation to compare the difficulty of colouring the original graphs vs. watermarked graphs, as well as the quality of the solution. They chose several graphs, including random and real-life benchmarks to perform the evaluation, and found that in almost all examples they obtain solutions of the same quality and with no overhead. Qu and Potkonjak built the first theoretical framework for analysing watermarking techniques and describe & provide proofs for their techniques.

Myles *et al.* [124] implemented the QP algorithm using register allocation; they found that a stricter embedding criteria are required due to the unpredictability of the colouring of vertices and the fact that one vertex can be used multiple times. Another, major, flaw in the QP algorithm is that it is possible to insert two different messages into an interference graph and obtained the same watermark graph [174]. The QP algorithm, therefore, is *not extractable* [173, 175].

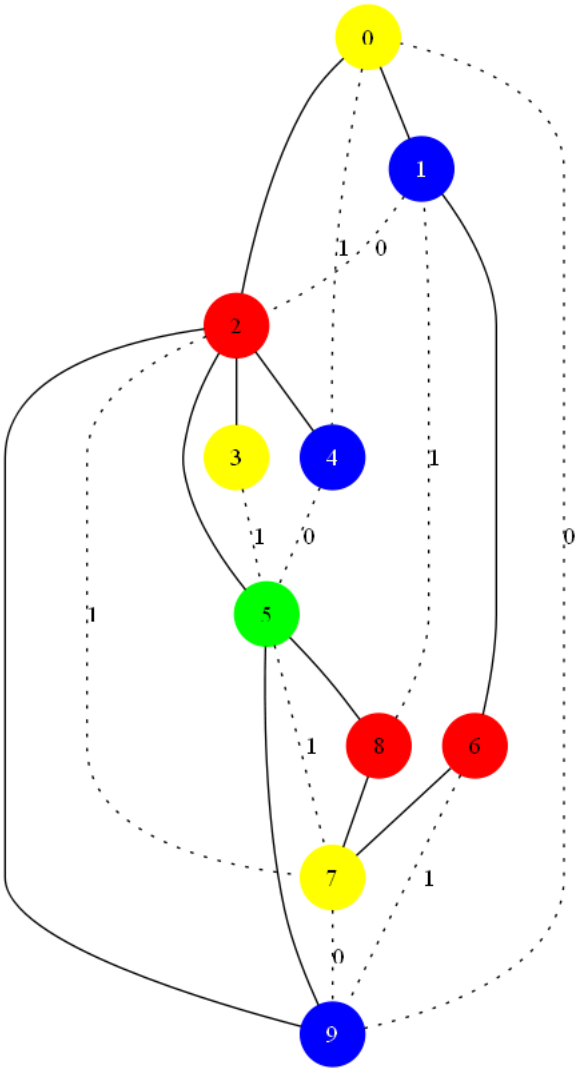


Figure 3.9: Example interference graph with embedded watermark

It has also been shown that the QP graph solution can be modified in such a way that any message could be extracted [103]. We can clearly see this from our example, figure 3.9: consider the pair (v_3, v_4); we can see that we could deduce that a watermark bit 0 is stored in an edge between these vertices. We know from our embedding that it is not, however it is impossible to tell this without the watermark edges. Qu and Potkonjak dismiss this problem, claiming that it will be hard to build a meaningful message particularily if the original message is encrypted by a one-way function [142].

Due to these flaws in the QP algorithm, Myles *et al.* [124] proposed an improvement which they call the QPS algorithm.

3.1.3 The QPS Algorithm

The key difference between the QP and the QPS algorithms is the selection of vertices. In the QPS algorithm triples of vertices are selected such that they are isolated units that will not effect other vertices in the graph. As watermark bits can only be inserted where there are coloured triples the data-rate of this algorithm is far lower than the QP algorithm.

Definition 4. *Coloured Triple [124]: Given a graph G, a set of three vertices v_1, v_2, v_3 (where $v_1 < v_2 < v_3$) is considered a coloured triple if they are vertices in G, are non-adjacent and they are all the same colour.*

Zhu *et al.* [173] provide clarified versions of the QPS algorithms which eliminate some ambiguities in the originals ; we use these here and they are shown in algorithms 3 and 4.

Figure 3.10 shows an example graph with a coloured triple {b, c, d}. The embedding algorithm is described, in pseudocode, in algorithm 3. Figure 3.11 shows the example graph, from figure 3.8, watermarked with 101_2 using the QPS algorithm. It is clear that the data-rate is much smaller than QPS; we can see that only 3 bits could be embedded using this algorithm.

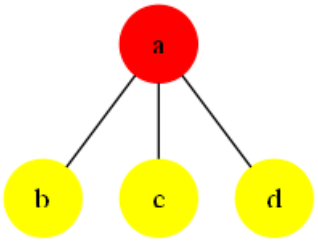


Figure 3.10: Coloured Triple

Algorithm 3 Clarified QPS embedding algorithm

Input: a graph $G(V, E)$, a message $M = m_0 m_1 \ldots$
Output: a graph $G'(V', E')$ with embedded message M

copy $G(V, E)$ to $G'(V', E')$
n = $|V|$ - 1
WV = V
j = 0
for all i from 0 to n **do**
  if possible find the nearest two vertices v_{i1}, v_{i2} in G'
  such that:
  v_i, v_{i1}, v_{i2} have the same colour
  and are a triple in G' and v_{i1}, $v_{i2} \in WV$.
  WV = WV - $\{v_{i1}, v_{i2}\}$
  j++
  if $m_j = 0$ **then**
    add edge (v_i, v_{i1}) to E'
  else
    add edge (v_i, v_{i2}) to E'
  end if
end for
return $G'(V', E')$

Algorithm 4 Clarified QPS extraction algorithm

Input: a graph $G'(V', E')$ with embedded message M
Output: a message $M = m_0 m_1 \ldots$

copy $G(V, E)$ to $G'(V', E')$
n = $|V|$ - 1
WV = V
j = 0
for all i from 0 to n **do**
  if possible find the nearest two vertices v_{i1}, v_{i2} in G'
  such that:
  v_i, v_{i1}, v_{i2} have the same colour
  and are a triple in G' and v_{i1}, $v_{i2} \in WV$.
  WV = WV - $\{v_{i1}, v_{i2}\}$
  j++
  if v_i and v_{i1} have different colours in G' **then**
    $m_j = 0$
    add edge (v_i, v_{i1}) to E'
  else
    $m_j = 1$
    add edge (v_i, v_{i2}) to E'
  end if
end for
return $M = m_1, m_2, \ldots m_j$

Myles *et al.* implemented the QPS algorithm in Sandmark [26], an open-source tool for the study of software protection algorithms. They performed a variety of empirical tests to evaluate their algorithm's overall effectiveness, examining five properties: credibility, data-rate, stealthiness, part protection and resilience. The results showed that the QPS algorithm has a very low data-rate and is susceptible to a variety of simple attacks, such as obfuscations. However, they also conclude that the QPS algorithm is quite stealthy and is extremely credible. In other words, the watermarks are hard to detect by an attacker whilst readily detectable by the watermark author.

3.1.4 The QPI Algorithm

The QPS algorithm is an improvement on the QP algorithm, in terms of extractability, however the QPS algorithm has a very low data-rate. Zhu *et al.* [173] proposed a further improvement which they call the QPI algorithm. The QPI algorithm changes the definition of the nearest vertices v_{i1}, v_{i2} to v_i and the original QP algorithm used cyclic mod n order for numbers 1,2,...,n, while QPI uses $1 < 2 < \ldots n$.

Definition 5. *Two candidate vertices for the QPI algorithm [173]: for a vertex v_i of a graph G with $|V| = n$ and a colouring of G, v_i has two candidate vertices $v_{i1} \in V$ and $v_{i2} \in V$ if $i < i_1 < i_2 \leq n$ and vertices v_i, v_{i1}, v_{i2} to v_i have the same colour and $(v_i, v_{i2}) \notin E$; furthermore, $\forall j : i < j < i_1$ and $\forall j : i_1 < j < i_2 \leq n$, vertices v_i and v_j have a different colour.*

The QPI embedding and extraction algorithms (see algorithms 5 and 6) uses a new definition of candidate vertices, coloured triples and also changes the vertex colours. Figure 3.12 shows the example interference graph from figure 3.8 with an embedded watermark 1011_2. The first watermark bit to embed is 1 and the first vertex is v_0; vertex v_0 has two candidate vertices v_3 and v_4 - these are the same colour and aren't connected to v_0. As with the previous algorithms, we add an edge between v_0 and v_4 because v_4 is the second candidate vertex and the watermark bit is 1. We then change the colour of v_4.

After we change the colour of v_2 in the process of embedding the second bit we leave v_2 without any candidate vertices. We therefore skip vertex v_2 and move on to the next vertex with candidate vertices.

The QPI extraction algorithm requires the original interference graph and the watermarked graph in order to extract the watermark message. We find the candidate vertices from the original graph, then compare the colours in the watermarked graph. For example, vertex v_0 has two candidate vertices v_3 and v_4; v_3 is the same colour in the original and watermarked graph, whereas v_4 is a different colour - the watermark bit is therefore 1.

The QPI algorithm is an improvement on the QPS algorithm with an increased data-rate and it has been shown that QPI algorithm is extractable, unlike the original QP algorithm [173].

Algorithm 5 QPI embedding algorithm

Input: a graph $G(V, E)$ and a message $M = m_0 m_1 \dots m_k$
Output: a graph G' with embedded message M

```
n = |V| - 1
G' = G
j = 0
for all i from 0 to n do
  if v_i has two candidate vertices v_i1, v_i2 then
    j++
    if w_j = 0 then
      add edge (v_i, v_i1) in G'
      change the colour of v_i1 to a different colour from the current colors used in G'
    else
      add edge (v_i, v_i2) in G'
      change the colour of v_i2 to a different colour from the current colors used in G'
    end if
  end if
end for
return G'
```

Algorithm 6 QPI extraction algorithm

Input: an unwatermarked graph $G(V, E)$ and watermarked graph $G'(V', E')$
Output: a message M

```
n = |V| - 1
j = 0
for all i from 0 to n do
  if v_i has two candidate vertices v_i1, v_i2 in G then
    j++
    if v_i and v_i1 have different colours in G' then
      m_j = 0
      add edge (v_i, v_i1) in G
      change the colour of v_i1 to a different colour from the current colors used in G
    else
      m_j = 1
      add edge (v_i, v_i2) in G
      change the colour of v_i2 to a different colour from the current colors used in G
    end if
  end if
end for
return M = m_0 m_1 ... m_j
```

3.1.5 The Colour Change Algorithm

The Colour Change (CC) algorithm [104] is another improvement on the QPS algorithm. In the CC algorithm the colouring function is modified to embed a message, rather than modifying the interference graph; the modification only occurs for 1 bits but not 0 bits. The data-rate of the CC algorithm is higher than that of QPS and QPI because each vertex in the interference graph can store 1 watermark bit.

Figure 3.13 shows the example interference graph (from figure 3.8) with the watermark 1011011010_2 embedded; for clarity, the vertices are annotated with the watermark bits.

The first step in the CC algorithm is to colour the interference graph to obtain the colouring function γ. We then adapt the colouring function γ to produce a new colouring function γ' which includes the embedded watermark (see algorithm 7). For example, the first watermark bit to embed is 1 and the original colouring of vertex v_0 is yellow; we choose the lowest possible legal colour (see table 3.1 for colour encodings) to replace red. The only vertices which do not change colour are the vertices in which a 0 bit will be encoded.

In order to extract the watermark, the CC extraction algorithm (see algorithm 8) takes the original interference graph G and the graph colouring function γ generated by the embedding algorithm. The original graph colouring γ' is obtained by colouring G which is then compared with the colours obtained by γ. If the original colouring of a vertex matches the new colouring then the watermark bit is 1; otherwise the watermark bit is 0.

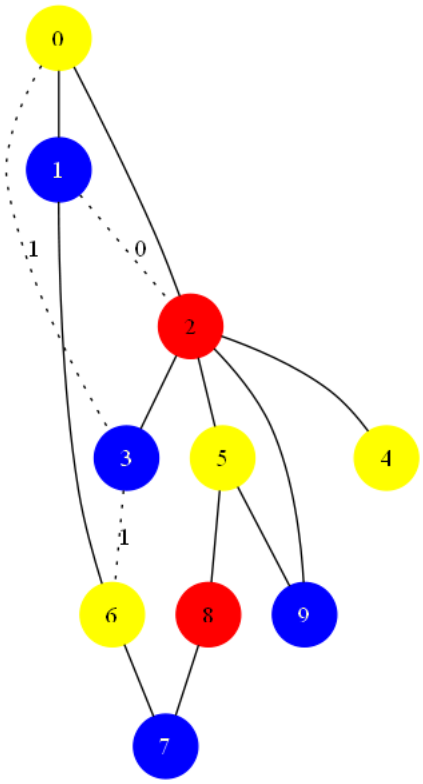


Figure 3.11: Example interference graph with embedded watermark using the QPS algorithm

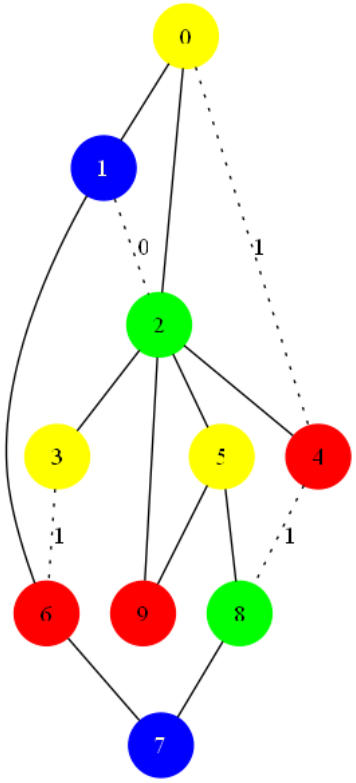


Figure 3.12: Example interference graph with embedded watermark using the QPI algorithm

Colour	Encoding
Yellow	1
Blue	2
Red	3
Green	4

Table 3.1

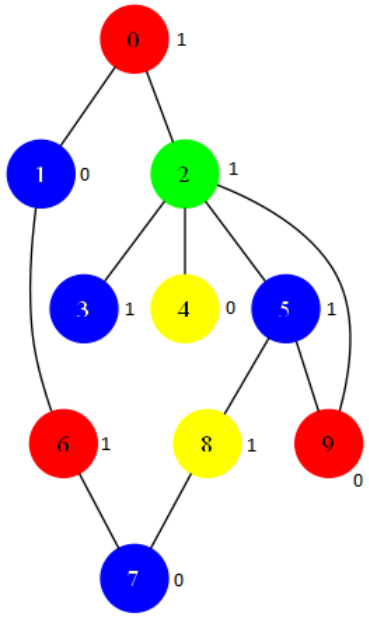


Figure 3.13: Example interference graph with embedded watermark using the CC algorithm

Table 3.2 shows the output of the original colouring function γ and the new colouring function γ' for each of the vertices.

Lee and Kaneko [105] experimentally evaluated their CC algorithm and found that, on average, it takes 0.75 extra colours to embed a message in an interefence graph. They suggest using the CC algorithm for programs which contain many variables, as the data-rate is equal to the number of vertices in the interference graph. They introduce a second algorithm - Color Permutation - for use in programs which require a large number of registers.

Vertex	0	1	2	3	4	5	6	7	8	9
γ	1	2	2	1	1	1	1	2	3	3
γ'	3	2	4	2	1	2	3	2	1	3

Table 3.2

Algorithm 7 Colour Change embedding algorithm

Input: a graph $G(V, E)$ and a message $M = m_0 m_1 \dots m_k$
Output: a colouring function γ

```
n = |V| - 1
if n < k then
  abort('embedding failed')
else
  γ = ColourGraph(G)
  for j = 0 to n do
    if m_j = 1 then
      find smallest i such that i ≠ γ[j] and
      i ≠ γ[j'] for any (j, j') ∈ E such that
      j' < j or m_j' = 0
      γ[j] = i
    end if
  end for
end if
return γ
```

Algorithm 8 Colour Change extraction algorithm

Input: a graph $G(V, E)$, graph colouring function γ
Output: message M

```
n = |V| - 1
γ' = ColourGraph(G)
for j = 0 to n do
  if γ[j] = γ'[j] then
    m_j = 0
  else
    m_j = 1
  end if
end for
return M = m_0 m_1 ... m_n
```

3.1.6 The Colour Permutation Algorithm

The Colour Permutation (CP) algorithm [104, 105] uses a similar idea to the CC algorithm, as they both change the colouring function for an intereference graph to encode the watermark bits. The CP algorithm converts the watermark bit string into a natural number M, and then chooses the M^{th} permutation of the lexicographically ordered colours to replace the original colour. The algorithm uses the relationship between the factorial number system and lexicographically ordered permutations [94, 10] to obtain the M^{th} permutation.

The encoding algorithm is show in algorithm 9. The published extraction algorithm is incorrect and we provide a corrected version here (algorithm 10).

The data rate of the CP algorithm is proportional to the number of colours c, given by $log_2 c!$, whereas the data rate of the CC algorithm is equal to the number of variables used. We can therefore only store two watermark bits in our example interefence graph (figure 3.8) as it contains 3 colours ($log_2 3! = 2$).

Figure 3.14 shows the example interference graph after encoding 10_2 with the CP algorithm. The embedding algorithm first converts 10_2 into the natural number 1; and then converts this to the factoradic $\{1, 0, 0\}$ and obtains the corresponding permutation $\{2, 1, 3\}$. The new colour permutation is applied to the original interference graph, so that yellow and blue colours are swapped.

To extract the embedded watermark we perform the opposite of the embedding algorithm (see algorithm 10). First the factoradic $\{1, 0, 0\}$ is obtained by comparing the original colouring and the new colouring which is then converted to the original natural number.

Table 3.3 shows the output of the original colouring function γ and the new colouring function γ' obtained using the CP algorithm.

Lee and Kaneko [105] experimentally evaluated their CP algorithm and found that the embedded watermark is stealthy and has a high credibility but low-resilience to semantics-preserving attacks.

The CP algorithm has the advantage that it will never introduce a new register; however, this means that the program must already use a large amount of registers in order to embed a large message. The

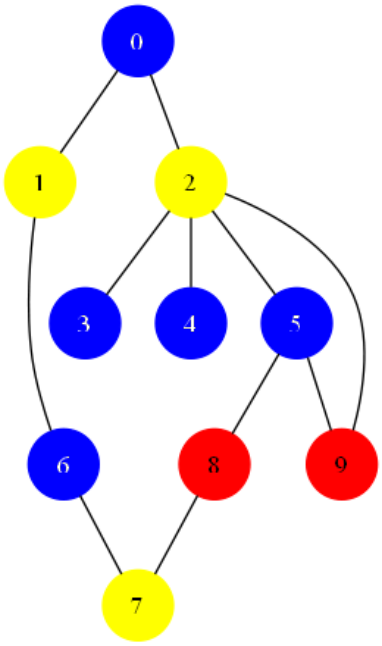


Figure 3.14: Example interference graph with embedded watermark using the CP algorithm

Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
γ	1	2	2	1	1	1	1	2	3	3
γ'	2	1	1	2	2	2	2	1	3	3

Table 3.3

CC and CP algorithms cannot embed a watermark consisting of all zeros.

3.1.7 The Selected Colour Change Algorithm

Li *et al.* [107] proposed a more efficient algorithm based on the CC algorithm which they call Selected Colour Change (SCC). The SCC algorithm is very similar to the CC algorithm, however, the efficiency increase is obtained by only changing the colours of either the 1 or the 0 bits, but not both. If the occurance of 0 bits is higher than 1 bits in the watermark then 0 bits are changed; otherwise 1 bits are changed. The choice of which bits are changed is stored in a virtual vertex v_{-1} to allow the embedding algorithm to extract the watermark.

Figure 3.15 shows the example intereference graph (from figure 3.8) using the SCC algorithm 11 to embed the watermark 1011011010_2. We change the colouring function for the 0 bits because the watermark string contains four 0 bits and six 1 bits; we also set the virtual node v_{-1} to 0 to inform the extraction algorithm of this. We change the colouring function in the same manner as the CC algorithm; that is, we choose the lowest possible legal colour as the replacement.

For example, the first bit to embed is 1 therefore we do not change the colouring function for vertex v_0. However, the next bit is a 0 therefore we so we change the colouring function to output a different colour for v_1. We choose red because v_1 is connected to two yellow vertices and v_1 is already blue (see table 3.1 for colour encodings). Table 3.4 shows the output of the original colouring function γ and the new colouring function γ' obtained using the CP algorithm.

In order to extract the watermark we observe the value of the virtual vertex v_{-1}. If $\gamma[-1] = 0$ then we assign a 1 to each watermark bit m_j where the original colouring for a vertex j matches the new colouring and a 0 bit if they differ. Otherwise, if $\gamma[-1] = 1$ then we assign a 0 to each watermark bit where the original colouring for a vertex matches the new colouring and a 1 bit if they differ (see algorithm 12).

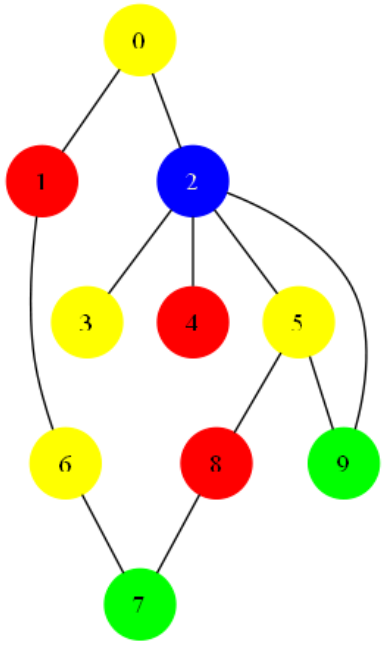


Figure 3.15: Example interference graph with embedded watermark using the SCC algorithm

Vertex	**-1**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
γ	*null*	1	2	2	1	1	1	1	2	3	3
γ'	0	2	1	1	2	2	2	2	1	3	3

Table 3.4

In our example $gamma[-1] = 0$ so we assign a 1 to each watermark bit m_j where the original colouring for a vertex j matches the new colouring and a 0 bit where they differ.

Li *et al.* [107] evaluate their SCC algorithm and compare the results to the QP, QPS and CC algorithms. They conclude that their SCC algorithm is equivilent to the CC algorithm in many ways, including data-rate, stealthiness and cost. They suggest the SCC algorithm is more efficient than the CC algorithm because it doesn't change all the colours in the interference graph; however, it is not clear how much the efficiency increase affects watermark embedding in real-world programs.

Algorithm 9 Colour Permutation embedding algorithm

Input: a graph $G(V, E)$, a message $M = m_0 m_1 \dots m_k$
Output: a colouring function γ

n = $|V|$ - 1
M = $\Sigma_{i=0}^{k} m_i 2^i$
γ = GraphColouring(G)
c = max(γ[0], γ[1], ..., γ[n])
$k' = [log_2 c!]$
if $k' < k$ **then**
  abort('embedding failed')
end if
for all $j = 1$ to c **do**
  $r[j] = M \bmod (c - j + 1)$
  $M = M \div (c - j + 1)$
end for
for all $h = 1$ to c **do**
  $u[h] = false$
end for
for all $j = 1$ to c **do**
  $cnt = 0$
  for all $h = 1$ to c **do**
    if $u[h] \neq true$ **then**
      if $cnt = r[j]$ **then**
        $u[h] = true$
        $p[j] = h$
        break
      else
        $cnt = cnt + 1$
      end if
    end if
  end for
end for
for all $j = 0$ to n **do**
  $\gamma[j] = p[\gamma[j]]$
end for
return γ

Algorithm 10 Corrected Colour Permutation extraction algorithm

Input: a graph $G(V, E)$, graph colouring γ
Output: a message $M = m_0 m_1 \dots m_k$

n = $|V|$ - 1
γ' = ColourGraph(G)
c = max(γ[0], γ[1], ..., γ[n])
for all $i = 1$ to c **do**
  $p[\gamma'[i]] = \gamma[i]$
end for
for all $i = 1$ to c **do**
  $u[i]$ = false
end for
for all $i = 1$ to c **do**
  $cnt = 0$
  for all $j = 1$ to c **do**
    if $u[j] \neq$ true **then**
      if $j = p[i]$ **then**
        $r[i] = cnt$
        $u[j]$ = true
        break
      end if
      $cnt = cnt + 1$
    end if
  end for
end for
$I = 0$
for all $j = 1$ to c **do**
  $I = I \times j + r[c - j + 1]$
end for
M = ""
while $I > 0$ **do**
  $M = M +$ (I mod 2)
  $I = I \div 2$
end while
return M

Algorithm 11 Selected Colour Change embedding algorithm

Input: a graph $G(V, E)$ and a message $M = m_0 m_1 \dots m_k$
Output: a colouring function γ

```
n = |V| - 1
if n < k then
  abort('embedding failed')
else
  n_0 = number of 0 bits in M
  n_1 = number of 1 bits in M
  γ = ColourGraph(G)
  if n_0 < n_1 then
    γ[−1] = 0
    for j = 0 to n do
      if m_j = 0 then
        find smallest i such that i ≠ γ[j]
        and i ≠ γ[j'] for any (j, j') ∈ E
        such that j' < j or m_j' = 1
        γ[j] = i
      end if
    end for
  else
    γ[−1] = 1
    for j = 0 to n do
      if m_j = 1 then
        find smallest i such that i ≠ γ[j]
        and i ≠ γ[j'] for any (j, j') ∈ E
        such that j' < j or m_j' = 0
        γ[j] = i
      end if
    end for
  end if
end if
return γ
```

Algorithm 12 Selected Colour Change extraction algorithm

Input: a graph $G(V, E)$, graph colouring function γ
Output: message M

```
n = |V| - 1
γ' = ColourGraph(G)
if γ[−1] = 0 then
  for j = 0 to n do
    if γ[j] = γ'[j] then
      m_j = 1
    else
      m_j = 0
    end if
  end for
end if
if γ[−1] = 1 then
  for j = 0 to n do
    if γ[j] = γ'[j] then
      m_j = 1
    else
      m_j = 0
    end if
  end for
end if
return M = m_0 m_1 ... m_n
```

3.1.8 Fingerprinting via Register Allocation

Qu *et al.* [143] proposed a modification to the QP algorithm for *fingerprinting*. Fingerprinting is a class of watermarking which embeds a unique identifier into each copy of the software (or music, or films, etc) which allows the origin of the stolen intellectual property to be identified.

The generic approach proposed is to generate n solutions to a graph colouring problem and assign a given solution to one customer only. This approach guarantees that each customer will be able to be uniquely identified by the register allocation generated by their unique inteference graph colouring.

3.1.9 QP Algorithms and Public-Key Cryptography

It is advisable to encrypt the watermark message before embedding to prevent an adversery from extracting a watermark, even if they know the extraction algorithm. For example, if an adversery obtains a program which they know contains a watermark embedding with the QPS algorithm they can simply run the QPS extraction algorithm and obtain the watermark; they could then claim that they inserted the watermark. Jian *et al.* [86] propose a technique based on a combination of RSA public-key encryption [148] and the QPI algorithm [173]. Simply, they suggest that the watermark bit string is encrypted before being embedded. If an adversery extracts the encrypted watermark they will not be able to decipher it, if the encryption is strong enough.

3.1.10 Conclusion

The QP algorithm has been shown to be unextractable [174] and an implementation of the QPS algorithm has shown that the algorithm has a low data-rate [75] and is highly suspceptible to semantics-preserving transformation attacks [124]. Any other register allocation based software watermarking algorithm will also be susceptible to semantics-preserving transformations. More specifically, any transformation which alters the interference graph or register allocation will easily remove a watermark.

Register allocation based algorithms maybe be stealthy [124], as they do not add any extra code, but we do not recommended them for protecting software due to their low-resilience to attacks. However, academic research continues in this area with the latest register allocation based approach [107] published this year.

We believe that further research should instead focus on dynamic software watermarking techniques which, in theory, should be resilient to semantics-preserving transformations; thus providing a robust technique for the protection of software.

3.2 Code Re-Ordering Watermarks

intro

3.2.1 Basic Block Re-Ordering

Davidson and Myhrvold [44] proposed one of the first software watermarking algorithms which encodes the watermark by basic block re-ordering. The embedding algorithm was described in a patent issued to Microsoft but the extraction algorithm was not discussed. Collberg *et al.* [126] proposed a method of watermark extraction and implemented the DM algorithm in Sandmark [26].

Definition 6. *Basic Block [4]: A basic block is a sequence of consecutive statements in which the flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.*

Collberg *et al.*'s extraction algorithm is an *informed* extraction algorithm; that is, it requires the original P and the watermarked program P^w to extract the watermark w. The embedding algorithm re-orders only unique basic blocks in all methods as there is no way of knowing which method(s) the watermark is stored in. This would result in the extraction of many watermarks; Collberg *et al.* overcome this in their implementation by prefixing and suffixing magic numbers to the watermark to guarantee recognition.

Definition 7. *Unique Basic Block [126]: A basic block is unique if and only if no other block in the graph contains the same instructions.*

Non-unique basic blocks can be made unique by inserting bogus code (such as *no op* instructions) until all the basic blocks are unique [27].

The first step in the DM algorithm is to convert the watermark into a number w; then the w^{th} permutation [93] of a set of basic blocks B is generated. The permutated basic blocks B' are re-linked to retain the original program semantics and B is replaced by B' to produce the watermarked program P'.

To extract a watermark the first step is to compare the ordering of the original basic blocks against the new ordering, to obtain the permutation number; this number is then converted back into the watermark number.

A program method containing n unique basic blocks can embed $[log_2 n!]$ watermark bits. The method should not contain exception handling code as this can impose an ordering of basic blocks which is difficult or impossible to alter [126].

Figure 3.16(a) shows a linearised control flow graph[1] [4] for the Java bubble sort program in listing 3.3. The program method contains 7 unique basic blocks (excluding *begin* and *end*) which means that this method can store 12 bits.

Correct?

Figure 3.16(b) shows the control flow graph after embedding the watermark 1011011010_2 (730_{10}). The linearised set of basic blocks for the control flow graph is $B = \{B_0, B_1, B_2, B_3, B_4, B_5, B_6\}$ of which the 730^{th} permutation is $B' = \{B_1, B_0, B_2, B_4, B_6, B_3, B_5\}$. The blocks in B' are relinked to retain the original control flow order by inserting additional *goto* statements where neccesary, for example at the end of B_0.

Hattanda *et al.* [78] evaluated the DM watermarking algorithm by watermarking several C programs and analysing metrics such as program size and program performance. In their implementation they found that the size increase of a watermarked program was between 9% and 24% while the performance was 86% to 102% of the original program. The 2% performance increase was due to the re-ordered code containing no redundant jump instructions, and that it showed more locality than the original - greater locality increases performance and increasing locality is a common optimisation performed by compilers [45].

does it? what is a partial permutation?

Hattanda *et al.* reported a data-rate of approximately 0.2% of program size (in bytes) based on their implementation that used a partial permutation scheme, which only used 6 basic blocks. Collberg *et al.*'s implementation was not constrained in this way and the data rate is dependent on the number of basic blocks in program methods.

[1] The code shown is Jimple [144] - a 3-address-code intermediate representation for Java bytecode, used by the Soot optimiser [164]

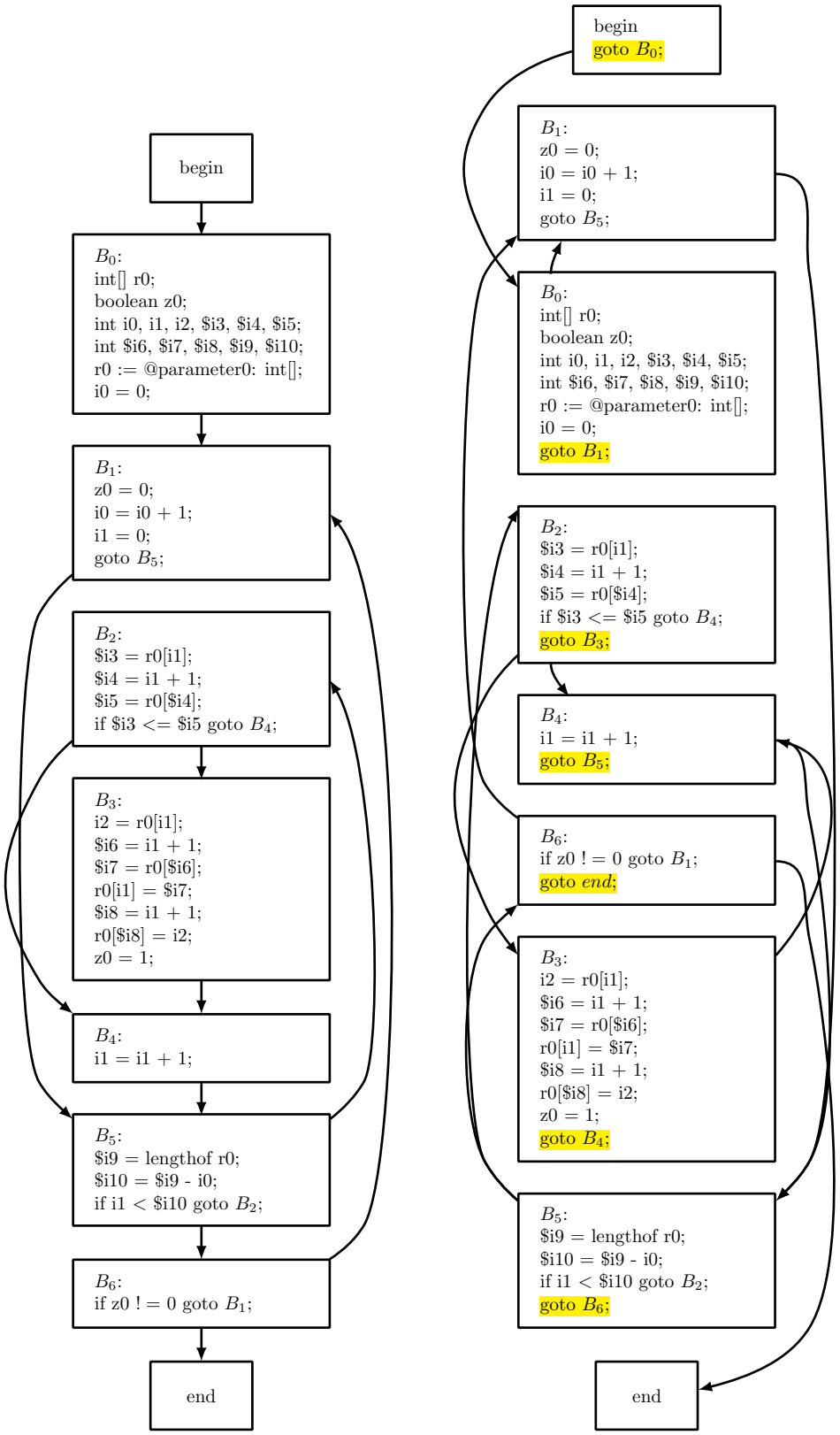
Listing 3.3: Bubble Sort in Java

```
public static void bubbleSort(int[] arr) {
    boolean swapped = true;
    int j = 0;
    int tmp;
    do {
      swapped = false;
      j++;
      for (int i = 0; i < arr.length - j; i++) {
          if (arr[i] > arr[i + 1]) {
            tmp = arr[i];
            arr[i] = arr[i + 1];
            arr[i + 1] = tmp;
            swapped = true;
          }
      }
    } while(swapped);
}
```

Algorithm 13 DM embedding algorithm

Input: a program P, a watermark number w
Output: a watermarked program P^w

G = CFG of a function in P
optimise and ensure unique blocks in G
linearise G as $B = \{B_0, B_1, \ldots B_n\}$
if $w \geq n!$ **then**
  abort('watermark too big')
else
  $B^w = w^{th}$ permutation of B
  add branches to B^w where necessary to ensure semantic equivalence with B
  $G^w = B^w$ replace B
  $P^w = G^w$ replace G
  return P^w
end if



(a) before wm (b) after wm

Figure 3.16: Linearised Control Flow Graphs for the Java Bubble Sort program, listing 3.3

The DM algorithm is highly unstealthy due to the fact that a normal compiler would not linearise the control flow graph as in DM watermarked programs [27]. A simple way to discover a DM watermark is to examine the ratio of *goto* statements to the total number of instructions - programs with the DM watermark show a high ratio compared to an unwatermarked program [126] (you can clearly see the difference in the number of *goto* statements in figure 3.16).

The biggest flaw with the DM watermark is that it is highly *fragile*; that is, it is not resilient to semantics-preserving transformations. For example, any transformation which re-ordered the basic blocks would eliminate the watermark [75].

Anckaert *et al.* [7] implemented and evaluated a version of the DM watermarking algorithm for machine code where groups of chains of basic blocks are re-ordered. They concluded that their watermarking algorithm is stealthier as it has a minimal affect on code locality.

Not sure exactly what this means.

Definition 8. *Basic Block Chain [8]: A set of basic blocks that must be placed consecutively.*

3.2.2 Equation Re-Ordering

Shirali-Shahrez *et al.* [153] proposed a software watermark scheme based the re-ordering of operations in mathematical equations. The idea involves re-ordering symmetric mathematical operations, such as addition, to preserve program semantics.

For example, equation 3.1 and equation 3.2 are equivalent and we could consider a 0 bit encoding for the first ordering, and a 1 bit encoding for the second ordering.

$$x = y + z \tag{3.1}$$

$$x = z + y \tag{3.2}$$

Not all operations are symmetric and the watermarking algorithm only re-orders *safe swappable* binary operations to ensure the watermarked equation is equivalent.

Definition 9. *Safe swappable operation [153]: An operation is safe swappable if it is symmetric and at least one of it's operands is constant.*

In order to produce a *blind* watermarking extraction algorithm an ordering is defined on the operands of a safe swappable operation:

1. If both operands are constant they are ordered according to their string representation.
2. If one operand is constant and the other is not then the constant operand would come first.

The sorted order of operands is retained to encode a 0 bit whereas the operands are reversed to encode a 1 bit. The extraction algorithm can then check to see if operands are in sorted order or not, to obtain 0 or 1.

Equation 3.3 contains 3 safe swappable operations: $5 \times y$, $x + 1$ and $2 \times (x + 1)$.

$$5 \times y + 2 \times (x + 1) \tag{3.3}$$

Of these 3 operations $x + 1$ is unordered according to the ordering definition; therefore the equation must be re-ordered as (3.4) before watermarking.

$$5 \times y + 2 \times (1 + x) \tag{3.4}$$

In order to encode a watermark we perform a pre-order traversal of the equation tree (see figure 3.17), swapping operations where necessary to encode bits. For example to encode the watermark 011_2 we leave the order of $5 \times y$ and change the order of $2 \times (1 + x)$ and $1 + x$, giving us equation 3.5.

$$5 \times y + (x + 1) \times 2 \tag{3.5}$$

The data-rate of this watermarking technique is related to the number of safe-swappable operations within a program. It is likely that there will be many, but the watermark will probably need to be split into pieces as most equations will only encode a small number of bits individually.

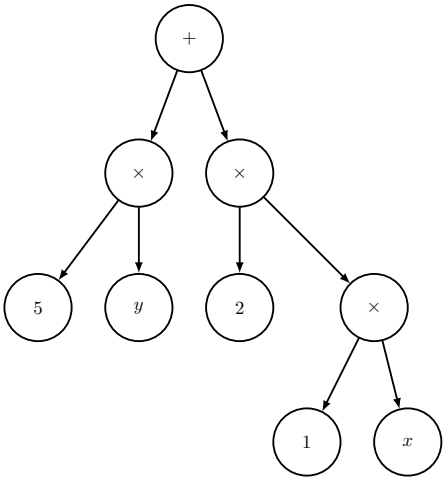


Figure 3.17: Equation tree for equation 3.4

Zonglu *et al.* [152] proposed a very similar technique using a re-ordering based on operand coefficients. Neither of these techniques cannot be applied to source-code as the compiler itself may re-order the operands and even when applied to bytecode or machine code this technique is highly susceptible to semantics-preserving transformations. Any re-ordering of the operands after watermarking will remove the watermark.

not sure about this one. add bogus function calls. requires re-ordering of functions. to extract look at the function ordering. goes under graph watermarking? opaque predicate algorithms?

3.2.3 Function Re-Ordering

Gupta and Pieprzyk [70] introduced a watermarking scheme for C/C++ by imposing an ordering on the mutually independent functions by introducing bogus dependencies.

3.2.4 Constant Pool Re-Ordering

Gong *et al.* [65] proposed a watermarking scheme, CPW, for Java based on the ordering of a class file's constant pool.

A Java compiler compiles Java source code into intermediate Java Byte Code in files ending with the extension *.class*.

Java class files are divided into 10 areas:

Magic Number
0xCAFEBABE

Version Numbers
The minor and major versions of the class file

Constant Pool
Pool of constants for the class

Access Flags
e.g. abstract, static, etc

This Class
The name of the current class

Super Class
The name of the super class

Interfaces
Any interfaces in the class

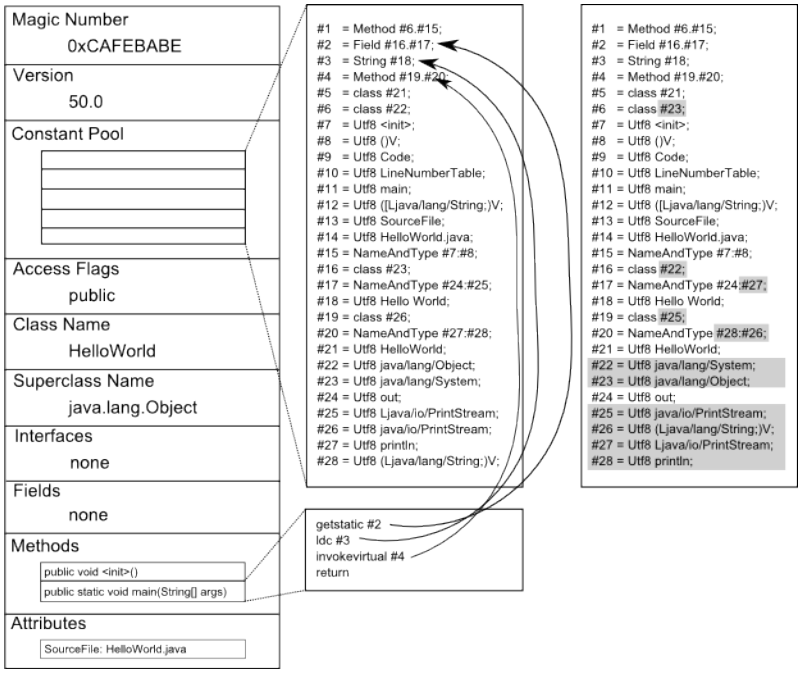


Figure 3.18: Conceptual Java class file diagram, showing constant pool before and after watermarking.

Fields
Any fields in the class

Methods
Any methods in the class

Attributes
Any attributes of the class

The constant pool is an array of variable length elements containing every constant used in the Java class [168]. Constants are referenced by an index throughout the bytecode. Some entries in the constant pool are *direct* references to a constant while other entries are references to other members in the constant pool - these are *indirect* constants. Each constant in the pool is preceded by a tag denoting it's type, for example *class*, *integer*, *Utf8* [108].

The CPW scheme involves re-ordering the *direct constants* corresponding to the W^{th} permutation of the direct constants where W is an integer watermark.

Figure 3.18 shows a conceptual diagram of listing 3.4; the constant pool is shown before and after watermarking. The set of direct constants D before watermarking is $D = \{7, 8, 9, 10, 11, 12, 13, 14, 18, 21, 22, 23, 24, 25, 26, 27,$

If we want to embed the watermark 1011011010_2 (730_{10}) we first obtain the 730^{th} permutation of D; this is $D' = \{7, 8, 9, 10, 11, 12, 13, 14, 18, 21, 23, 22, 24, 26, 28, 25, 27\}$. Figure 3.18 shows the constant pool after watermarking, on the right, where the direct constants have been re-numbered according to the permutation.

To extract the watermark we calculate the original ordering of the direct constants (in the same way as the original Java compiler did) and compare with the watermarked constant pool to find the permutation number W.

Gong *et al.* evaluated their algorithm by embedding watermarks into 4000 random class files downloading from the Internet and concluded that it has a good robustness but small capacity. However, the CPW algorithm is far from robust - any further re-ordering of the constant pool would destroy the watermark.

Listing 3.4: Hello World in Java

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println(             );
  }
}
```

3.2.5 Conclusion

We have presented a survey of software watermarking schemes based on code re-ordering. This family of watermarks are highly susceptible to semantics-preserving transformation attacks [75], and can be unstealthy. The DM watermark is highly unstealthy due to addition of large numbers to *goto* statements inserted to preserve the original control-flow after re-ordering.

Any further re-ordering of a program watermarked with any of the presented algorithms will likely destroy the watermark. In fact, even re-watermarking the program will likely destroy the watermark.

Academic research continues in this area, with the latest paper published last year [152]. We believe that further research should instead focus on dynamic software watermarking techniques which, in theory, should be resilient to semantics-preserving transformations; thus providing a robust technique for the protection of software.

3.3 Graph Watermarking

3.3.1 Encoding Watermarks in Graphs

Collberg et al. [33] describe several techniques for encoding watermark integers in graph structures. The algorithms in this paper rely on the fact that graph-generating code is difficult to analyse due to aliasing effects [64] which, in general, is known to be un-decidable [145].

An ideal class of watermarking graph should have the following properties [37]:

- ability to efficiently encode a watermark integer; and be efficiently decodable to a watermark integer
- a root node from which all other nodes are reachable
- a high data-rate
- a low outdegree to resemble common data structures such as lists and trees
- error correcting properties to allow detection after transformation attacks
- tamper-proofing abilities
- have some computationally feasible algorithms for graph isomorphism, for use during recognition

Graph Enumeration

The family of graph encoding is based on a branch of graph theory known as graphical enumerations [77]. An integer watermark n is encoded as the n^{th} enumeration of a given graph.

Directed Parent-Pointer Trees

subsubsubsetcion

This family of graphs contains a single edge between a vertex and it's parent. For example, there are 4 possible enumerations of DPPTs having four indistinguishable vertices [93], as shown in figure 3.19. We choose the n^{th} enumerated graph to embed the watermark number n.

how is it ordered?

Implementing a parent-pointer tree data-structure is space efficient because each node has just one pointer field referencing it's parent. However, the data-structure is fragile and an adversary could add a single node or edge to distort the watermark.

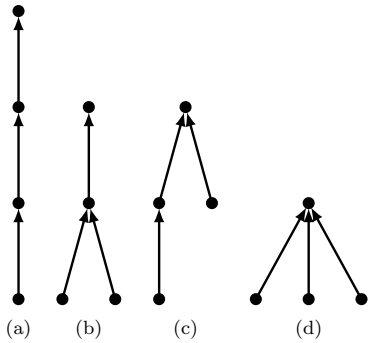


Figure 3.19: Enumerations of a directed graph with 4 indistinguishable vertices.

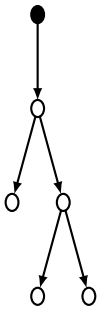


Figure 3.20: Planted Planar Cubic Tree

Planted Planar Cubic Trees

subsubsubsection

Planted planar cubic trees (PPCT) are binary trees where every interior vertex, except the root, has two children (see figure 3.20). These are easily enumerated by using a Catalan recurrence [18, 95]. Figure 3.22 shows PPCT enumerations with 1 to 4 leaves; the Catalan number $c(n)$ gives the number of unique trees for each n. An integer is encoded as one of the trees, for example, a 0 could be encoded as any of the trees in the first column.

The Catalan number is given by the formula:

$$c(n) = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \text{ for } n \geq 0 \tag{3.6}$$

PPCTs can be made more resilient to attacks by a) marking each leaf with a self-pointer, and b) creating an outer cycle from the root to itself through all the leaves [27] (see figure 3.21). This allows single edge and node insertions to be detected; however, multiple changes cannot be detected or corrected. The PPCT graphs have a lower bit-rate than other graph families but are more resilient to attacks.

Radix Graphs

Radix graphs add an extra pointer field in each vertex of a circular linked list of length k to encode a base-k digit. It is possible to encode watermark digits where a self-pointer represents 0, a pointer to the next node 1, and so on. Figure 3.23 shows the radix-5 expansion of 365_{10} (2430_5) encoded in a linked list structure.

These graphs give the highest data-rate for encoding watermarks however they are fragile as the watermark could easily be distorted. We can add redundancy to these watermarks by restricting the indegree to two and outdegree to two and using a permutation graph [33].

Several papers [21, 139, 171, 170, 87, 172] describe a technique which combines radix graphs with the error-correcting properties of PPCTs by converting a radix graph into a PPCT-like structure.

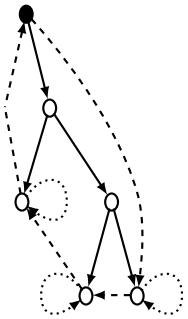


Figure 3.21: Enhanced Planted Planar Cubic Tree, with leaf self-pointers (dotted) and an outer cycle (dashed).

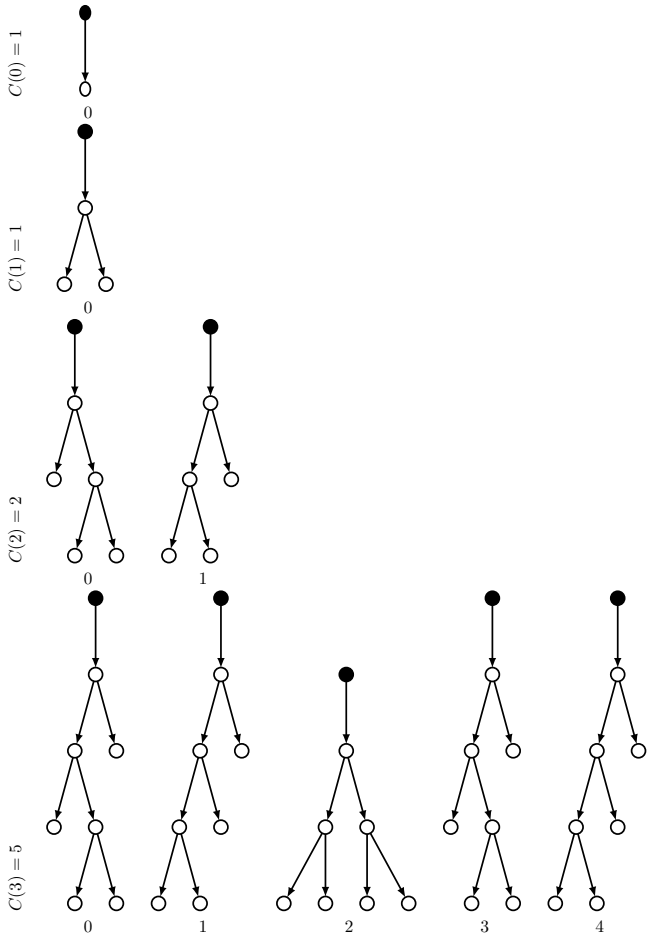


Figure 3.22: Enumerations of PPCTs with 1 to 4 leaves, showing the index below.

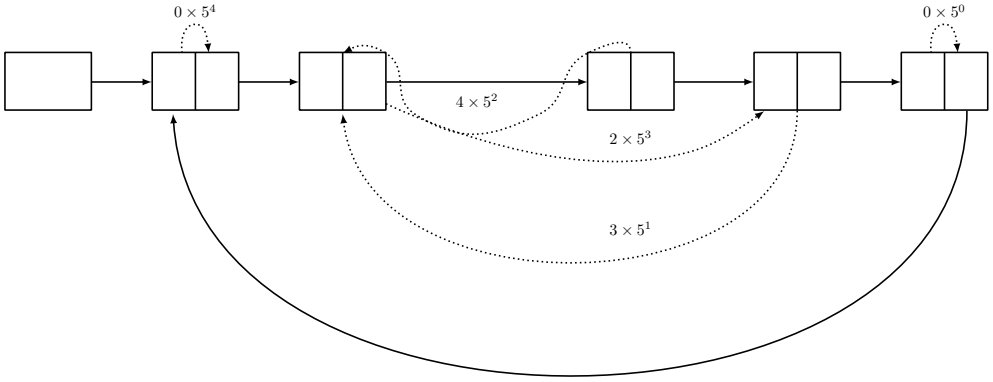


Figure 3.23: Radix-5 expansion of the watermark 365. The spine is black and edges representing radix-k are dotted.

Permutation Graphs

different from wikipedia & Spinrad [158]

Permutation based graphs (as defined by Collberg et al. [33]) use the same basic singly linked circular list structure as the radix graphs but have error-correcting properties. In this encoding scheme a permutation $P = \{p_1, p_2, \ldots, p_n\}$ is derived from the watermark integer n; the permutation is then encoded in the graph by adding edges between vertices i and p_i.

Collberg and Nagra [27] give algorithms for encoding and decoding an integer as a permutation, shown here as algorithm 14 and 15. For example, the integer 97_{10} is encoded as the permutation $\{3, 1, 0, 2, 4\}$ according to algorithm 14. We then encode this in a graph as shown in figure 3.24.

If an adversary changes one of the non-spine pointers to point to a different vertex we will discover an error because the permutation graph encodes a unique permutation - each vertex must have indegree two. However, if the adversary swaps two pointers we will lose the watermark.

Algorithm 14 int2perm [27]

Input: integer V, length of permutation len
Output: permutation encoding of V

$perm = (0, 1, 2, \ldots, len - 1)$
for $r = 2$; $r < len$; $r++$ **do**
  swap perm$[r-1]$ and perm$[V \bmod r]$
  $V = V \div r$
end for
return perm

Reducible Permutation Graphs

Reducible permutation graphs (RPG) [167, 166] are very similar to permutation graphs but they closely resemble control-flow graphs as they are reducible-flow graphs [81, 80].

Definition 10. *Reducible flow graph [4]: A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, often called* forward *edges and* back *edges, with the following two properties:*

1. *The forward edges from an acyclic graph in which every node can be reached from the intial node of G.*

2. *The back edges consist only of edges whose heads dominate their tails*

Algorithm 15 perm2int [27]

Input: a permutation *perm*
Output: encoded integer V

$V = 0$
$f = 0$
for $r = length(perm)$; $r \geq 2$; $r--$ **do**
  for $s = 0$; $s < r$; $s++$ **do**
    if $perm[s] == r - 1$ **then**
      $f = s$
      break
    end if
  end for
  swap perm$[r-1]$ and perm$[f]$
  $V = f + r \times V$
end for
return V

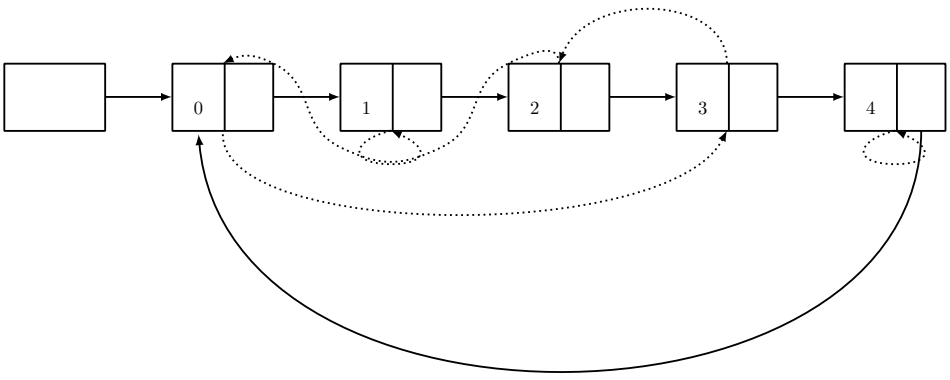


Figure 3.24: Permutation graph encoding the integer 97_{10} using algorithm 14 to generate the permutation $\{3, 1, 0, 2, 4\}$.

Listing 3.5: Example Java Method - Sum

```
public static void sum(int[] numbers) {
  int total = 0;

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

  if(total < 0) {
    System.out.println(                              );
  }else{
    System.out.println(            + total);
  }
}
```

Listing 3.6: Example Static Graph Watermark Method - Sum

```
public static boolean wm(int i) {
  if(i < 0)
    return true;
  else
    return false;
}
```

The reducibility of this family of graphs means that they resemble control-flow graphs constructed from programming constructs such as *if*, *while* etc. [27].

RPGs, like CFGs, contain a unique entry node and a unique exit node, a preamble which contains zero or more nodes from which all other nodes can be reached and a body which encodes a watermarking using a self-inverting permutation [22].

Definition 11. *Inverse permutation [22]: an inverse permutation of* $\{p_0, p_1, \ldots, p_n\}$ *is the permutation* $\{q_0, q_1, \ldots, q_n\}$ *where* $q_{p_i} = p_{q_i} = i$.

hmm...

Definition 12. *Self-inverting permutation [22]: a permutation that is it's own inverse, i.e.* $P_{p_i} = i$.

For example, the following permutation $Q = \{0, 5, 2, 10, 4, 1, 9, 7, 8, 6, 3\}$ is a self-inverting permutation of $P = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

This family of graphs is resistant to edge-flip attacks, where an attacker inverts the condition of conditional jumps in a program.

draw graph example

3.3.2 Static Graph Watermarking

Venkatesan et al. [167] proposed the first static graph watermarking scheme, Graph Theoretic Watermarking (*GTW*), which encodes a value in the topology of a program's control-flow graph [4]. The idea was later patented by Venkatesan and Vazirani [166] for Microsoft. The basic concept is to encode a watermark value in a reducible permutation graph and convert it into a control flow graph; it is then merged with the program control flow graph by adding control flow edges between the two.

Figure 3.25(a) shows the control flow graph for the Java method in listing 3.5 and figure 3.25(b) shows our watermark graph - this graph encodes our watermark (it's not important, for this example, what the watermark is; just that 3.25(b) encodes our watermark).

The watermark graph in figure 3.25(b) is also the control flow graph for the Java method in listing 3.6 - we embed our watermark graph in a program by converting a graph watermark into a control flow graph using programming constructs from the programming language we are using.

The watermark control flow graph can be integrated with the original program control flow graph as shown in figure 3.25(c) and code listing 3.7.

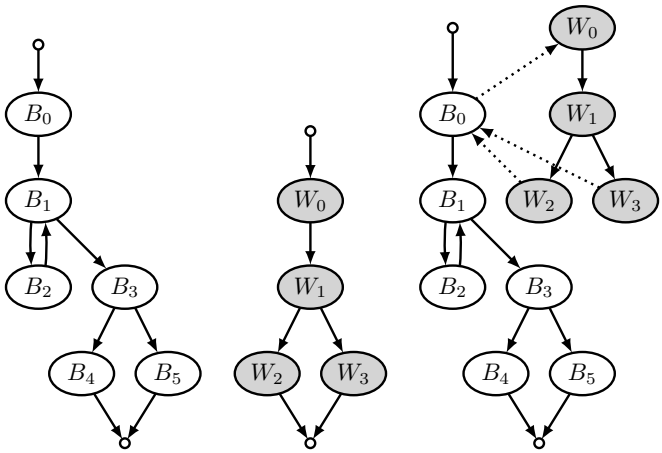
The algorithm adds bogus control flow edges between random pairs of vertices in the program CFG and watermark CFG in order to protect against static analysis attacks looking for sparse-cuts [11] in the control-flow graph. A sparse-cut would indicate a possible joining point of the original program

Listing 3.7: Example Static Graph Watermark Java Method - Sum

```
public static void sum(int[] numbers) {
  int total = 0;
  wm(total);
  for(int i = 0; i < numbers.length; i++) {
    total += numbers[i];
  }

  if(total < 0) {
    System.out.println(                              );
  }else{
    System.out.println(            + total);
  }
}

public static boolean wm(int i) {
  if(i < 0)
    return true;
  else
    return false;
}
```



(a) P (b) W (c) P_W

Figure 3.25: Graph theoretic watermarking

Listing 3.8: Example Dynamic Graph Watermarked Java Method - Sum

```
class Node { public List<Node> children = new ArrayList<Node>(); }

public static void sum(int[] numbers) {
  int total = 0;

  if(numbers.length > 5) {
    Node root = new Node();
    Node n1 = new Node();
    root.children.add(n1);
    Node n2 = new Node();
    Node n3 = new Node();
    n1.children.add(n2);
    n1.children.add(n3);
  }

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

  if(total < 0) {
    System.out.println(                              );
  }else{
    System.out.println(            + total);
  }
}
```

CFG and the watermark CFG where the attacker could split the program with as few edges broken as possible. For example, we could insert a call to the `sum`, or any other method in the original program, in the watermark method. The *GTW* algorithm also inserts bogus method calls in between parts of the original program so that the location of bogus calls cannot be used to point out the location of the watermark.

In order to recognise a *GTW* we must know which basic blocks, from the program's CFG, are part of the watermark graph; we must therefore mark the watermark basic blocks in some way in order to identify them. We could, for example, re-order instructions such that they are in lexicographic order, or insert bogus instructions to identify a watermark basic block [27]. However, these techniques are not resilient to semantics-preserving transformations and an attacker could remove the marks so that we cannot mark our watermark blocks.

Collberg et al. [24] implemented a version GTW_{SM} of *GTW* in Sandmark [26] in order to evaluate the algorithm. They measured the size and time overhead of watermarking and evaluated the algorithm against a variety of attacks. They also introduce two methods (Partial Sum splitting and Generalised Chinese Remainder Theorem [102] splitting) for splitting a watermark integer into redundant pieces so that a large integer can be stored in several smaller CFGs. They found that stealth is a big problem; for example, the basic blocks of the generated watermark method consisted of 20% arithmetic instructions compared to just 1% for standard Java methods [35]. Watermarks of up to 150 bits increased program size by between 40% and 75%, while performance decreased by between 0% and 36% [24].

3.3.3 Dynamic Graph Watermarking

Collberg and Thomborson [30] proposed the first dynamic graph based watermarking scheme *CT* to overcome problems with static watermarking schemes; most notably, static watermarks are highly fragile and therefore susceptible to semantics-preserving transformation attacks [75].

Dynamic graph watermarking schemes are similar to static graph watermarking except the graph structures are built at run-time. *CT* can use any of the previously described graph encoding schemes to store the watermark.

Figure 3.8 demonstrates how we could, trivially, embed our example watermark graph from figure 3.25(b) dynamically into the example Java program from listing 3.5. We only execute the watermark code when the `numbers` array is longer than 5 - this serves as our secret input. We would inspect the Java heap to retrieve our watermark when the program is executed with the secret input.

The first implementation [135] of the CT algorithm CT_{JW}, implemented in a system called JavaWizz [134], was for Java bytecode using PPCT graphs to encode the watermark integer. A class is chosen

Listing 3.9: CT watermark implementation.

```
class A {

  A() {
    ....
  }
  ....
}

class A1 extends A {

  A1() {
    super();
    // graph building
    // code goes here
  }

}
```

from the program to be watermarked and converted into a *node class* by adding additional fields which contain references to other nodes.

The graph building code is including in the program by sub-classing a class and putting the graph building code in it's constructor. For example, in listing 3.9 the class $A1$ is a sub-class of A; to build the graph an execution of *new* $A()$ only has to be replaced by *new* $A1()$.

Palsberg et al. [135] discuss a simple implementation which only resists attacks against dead-code removal; they suggest that other software protection techniques are applied after the program has been watermarked. Dependencies are added between the watermark generating code and the original program by replacing a statement S with a statement of the form $if(x \neq y)\ S$ where x and y are distinct nodes in the watermark graph.

In order to retrieve the watermark, the watermarked program is executed and the Java heap is accessed by dumping an image using the `-Xhprof` heap profiling JVM option [133].

The following steps are performed to retrieve the watermark:

1. Extracting potential node classes
2. Exracting potential node objects
3. Determining potential edges
4. Searching for the watermark graph

In general, searching for the graph would be an NP-complete problem however we know that the graph is a PPCT graph of a certain size and can prune away unnecessary nodes. Palsberg et al. [135] evaluated their implementation and found that, although it is possibly to retrieve a watermark, it is time-consuming — taking from 0.6 minutes up to 8.9 minutes on their set of test programs. It was shown that the CT algorithm is a good watermarking scheme because it is stealthy and resilient to semantics-preserving transformation attacks but it must be combined with other software protection techniques such as obfuscation [31] and tamper-proofing [36].

Collberg and Thomborson [37] implemented a version of the CT algorithm, CT_{SM}, in Sandmark [26] and compared it to JavaWiz [134]. The CT_{SM} implementation differs from the JavaWiz in the following ways:

- CT_{SM} requires a key to encode and decode the watermark, requiring user annotation of the program to be watermarked
- CT_{SM} can encode arbitrary strings whereas CT_{JW} requires an integer watermark
- CT_{JW} uses PPCTs to encode the watermark, whereas CT_{SM} offers the choice of Radix, PPCT, Permutation or Reducible Permutation graphs; also offering a cycled graph option to protect against node-splitting attacks.

- CT_{JW}'s graph building code is concentrated in one location whereas CT_{SM} embeds code fragments at sereval user-specified locations
- CT_{JW} uses an existing class for graph nodes whereas CT_{SW} generates its own
- CT_{JW} uses the heap profiling option of the JVM whereas CT_{SM} uses the Java Debug Inteferace (JDI) API [161]

The CT_{SW} algorithm embedding algorithm consists of the following steps:

Annotation
In this first step the programmer must insert calls to a method `mark()` which indicates to the watermarking system locations in which graph building code can be inserted. This manual annotation step allows a programmer to carefully choose the best locations for code insertion, maximising the stealthiness of the embedded watermark.

Tracing
After the code has been annotated the program is run with a secret input during which one or more annotation points will be encountered. Some of these encountered locations are used to store the graph building code.

Embedding
The watermark is encoded in a graph structure (strings watermarks are first converted to integers) and is converted into Java bytecode that builds the graph. The graph node class is generated from a similar class already in the program, or if no suitable class is found, classes from the Java library, such as `LinkedList` could be used. The graph is partitioned into several, equal sized, sub-graphs so that the code is more stealthy and the code is inserted at the marked locations discovered in the tracing step.

During extraction the program is run with the secret input; this will invoke the graph building code at the locations marked by the programmer. The graph structure will be on the Java heap and the Java debugging interface is used to retrieve it. The JDI is used to add breakpoints to every constructor in the program to build a circular linked buffer of the last 1000 objects allocated. The list is then searched, in reverse, for the watermark graph.

Collberg and Thomborson [37] evaluate CT_{SM} and conclude that it is efficient and resilient to semantics-preserving transformations. The greatest vulnerability is it's limited stealthiness and it is suggest that future work investigates combining a locally stealthy but not steganographically stealthy technique, such as describe by Nagra and Thomborson [130], with the CT algorithm would be helpful.

3.3.4 Attacks against graph watermarks

Collberg et al. [37] describe 4 types of attacks against dynamic graph watermarking schemes:

1. adding extra graph edges
2. adding extra nodes to the graph
3. renaming and re-ordering instance variables
4. splitting nodes into several linked parts

However, an adversary will probably not know the location of the watermark in a large program and as such would have to apply transformations across the program resulting in a large performance decrease. The use of RPG or PPCT graphs prevent renaming and re-ordering attacks because these graph encodings do not rely on the order of the nodes.

The biggest problem is the addition of bogus nodes to a graph for which tamper-proofing techniques are required. Collberg et al. [37] suggest the use of Java reflection may help. However, they did not implement this because they conclude that the code would be unstealthy as this kind of code is unusual in most programs.

Luo et al. [109] describe an algorithm based on CT using the Chinese Remainder Theorem [102] to split the watermark .

Is this just plagiarised from collberg?

3.3.5 Tamper-proofing by Constant Encoding

He [79], and later Thomborson et al. [163], developed a tamper-proofing technique for dynamic graph watermark called constant encoding. Constant encoding encodes some of the constants in a program into a tree structure similar to the watermark and decodes them at run-time. An attacker cannot distinguish between the watermark graph and the constant encoding graphs so they cannot change any graph structure as they may introduce bugs into the program.

Figure 3.26: A possible graph encoding of the constant 0.

For example, assuming that the constant 0 can be encoded as the graph in figure 3.26, we can replace the constant 0 in the example Java program from listing 3.5 with a method call which decodes the graph back into the constant 0, as shown in listing 3.10.

Listing 3.10: Example Constant Encoding Java Method - Sum

```
class Node { public List<Node> children = new ArrayList<Node>(); }

public static void sum(int[] numbers) {

  Node cg = buildConstantGraph();

  int total = decode(cg);

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

  if(total < 0) {
    System.out.println(                              );
  }else{
    System.out.println(            + total);
  }
}
```

An attacker that manipulates the constant graph will change the encoding; so the decode function will return a different number and the program will be semantically incorrect.

A weakness of this system is that an attacker may be able to discover that certain program functions always return the same value, for every execution. He [79] suggests introducing dependencies, to make the code harder to analyse, by providing an input variable to the decoding function. A further suggestion, for future work, is to change the constant tree at runtime making the code harder to analyse.

Jian-qi et al. [85] propose using the constant encoding functions within the parameters of opaque predicates Collberg et al. [32] so that if an attacker manipulates a constant graph control will pass to the wrong clause of an *if* statement using the opaque predicate.

Thomborson et al. [163] combine the watermarking and tamper-proofing techniques further by using sub-graphs of the watermark graph to encode constants, rather than separate graphs. Thomborson et al. [163] also formally define the problem of tamper-proofing and show that commonly occurring constants in computer programs can be replaced automatically and that a long series of dynamic analyses would be required to remove the watermark. However, it is not clear if their constant encoding technique is resilient against pattern-matching attacks – a question which is left for future work.

Khiyal et al. [91] suggest splitting constants so that large constants can be encoded in small trees. They evaluated the constant encoding technique by comparing watermarked and tamper-proofed programs for efficiency, size and resilience. They found that the size of a program does increase when tamper-proofed and the execution time is slower; however, the tamper-proofing technique should be used with obfuscation to further protect a tamper-proofed program.

Table 3.5: Mutually replaceable instructions with their value

iadd	000
isub	001
imul	010
idiv	011
irem	100
iand	101
ior	110
ixor	111

3.3.6 Conclusion

We have presented a survey of software watermarking schemes based on graph encoding using both static and dynamic embedding techniques. Static techniques, as always, are highly susceptible to semantics preserving transformation attacks and are therefore easily removed by an adversary.

Dynamic graph watermarking, however, is resilient to most semantics-preserving transformations if the right graph encoding is chosen. However, a choice between variable degrees of high stealth, resilience and bit-rate has to be made, as no algorithm has been developed which combines the best of all these properties.

Further research should continue into dnyamic graph watermarking, especially on improving the CT algorithm from attacks. We intend to investigate the use of program slicing [169] to attack dynamic watermarks and improve resilience to subtractive attacks.

3.4 Code Replacement

Some of the first patented software watermarking algorithms [83, 151] were based around the idea of code replacement; that is, they replaced a pre-determined portion of code and/or data in a program with the watermark value. These early techniques were susceptible to attacks, such as collusion attacks – where the attacker compares two or more copies of a watermarked program to identify the location of the watermark.

Monden et al. [119, 120, 121] describe a technique, MON, for watermarking Java programs by swapping bytecode instructions within dummy methods. The dummy methods used by MON are created either manually or automatically, and method calls are protected by opaque predicates [32] to ensure they are not executed. The original implementation of MON is available as jMark [118]. jMark requires the dummy method to be created and specified during embedding.

The basic idea is to assign bit values to certain Java bytecode instructions and replace the existing instructions with the encoding bits which correspond to the watermark value. As the dummy method is not executed there is no semantic restrictions on the replacements but the watermarked method must be semantically correct, in order to pass the Java bytecode verifier [168].

Two methods are used to replace bytecode in the dummy method: replacement of numerical operands and replacement of opcodes. The replacement of numerical operands ensures syntactic correctness is preserved, however in order to increase the bit-rate of the dummy method opcodes can be replaced too.

The replacement of opcodes requires the definition of mutually replaceable opcode groups, for example `ifnull` and `ifnonnull` are mutually replaceable but `iconst_0` and `ifnull` are not. Table 3.5 shows an example of mutually replaceable instructions, along with their assigned value. So the watermark 001011011010_2 can be encoded in the opcode sequence `isub`, `idiv`, `idiv`, `imul` - the watermarked method could look like listing 3.11 (using the instructions in table 3.5).

Listing 3.11: Watermarked using the instruction encoding from table 3.5

```
iconst_1
iconst_2
isub
idiv
idiv
imul
```

Monden et al. [121] evaluate jMark by watermarking some Java classfiles and attacking them using and obfuscator and decompiler. They conclude that their system is resilient to most attacks, however, this is due to the chosen systems for evaluation. For example, the chosen obfuscator does not replace any of the watermark instructions but this is a trivial semantics-preserving transformation which could easily remove the watermark.

It is not possible for the recogniser to know which method has the watermark embedded so jMark decodes all of the methods and displays all possible embedded watermarks.

Myles et al. [126] implemented a version, MON_{SM}, in Sandmark [26] and evaluated compared it to the Davidson/Myhrvold watermarking scheme [44].

MON_{SM} places a magic character at the beginning of an embedded watermark to allow identification of the watermarked method, unlike jMark which decodes all methods.

MON_{SM} differs from the jMark implementation as it automatically generates a dummy method, so is completely automatic. However, it is difficult to generate code which is similar to the original program and it may be discoverable by a statistical analysis of the bytecode. It may be possible to generate code which is statistically similar to the existing program code so that the watermarked method is more stealthy.

Myles et al. [126] found that the MON_{SM} algorithm is highly credible, if the program is unaltered, but not very stealthy due to the automatic generation of the dummy method. They also found that the MON technique is not resilient to attacks, including distortive, additive, subtractive and collusive attacks.

Fukushima and Sakurai [61], Fukushima et al. [62] combine the MON watermarking technique with obfuscation to provide protection against collusion attacks. The idea is to obfuscate each copy of a program differently, so that comparison of programs will show several differences – not just the watermark location.

Thaker [162] introduces a similar scheme which embeds watermarks by swapping semantically-equivalent x86 assembly instructions. The approach is applied to existing code and produces a semantically-equivalent watermarked program.

Another tool, Hydan [52, 53, 51] similarly uses the replacement of semantically-equivalent instructions for the purpose of steganographically embedding secret information in x86 binary programs. Steganography does not require high robustness, compared to watermarking, but the two techniques are very similar. Anckaert et al. [7, 8] further discuss Hydan's technique for steganography, comparing it to other techniques for hiding information in computer programs.

Pervez et al. [138] describe a MON-like system which acts on Java source, instead of bytecode. For example, their dictionary for encoding includes assigning 000 to an if-statement, 001 to if-else, etc.

3.4.1 Spread Spectrum Watermarking

Stern et al. [159] introduce robust object watermarking ROW, based on a spread-spectrum technique previously used for multimedia watermarking [39]. This technique differs from many other techniques because it views the code as a whole statistical object, rather than a sequence of instructions. The technique is more resilient against collusion attacks because the watermark is spread out over the program, rather than being in one location (such as in [83, 151]). The approach modifies the frequencies of groups of instructions in order to watermark the code (though other statistical properties of the program could be used). Stern et al. [159] implemented their technique for x86 assembly language and later Hachez [71], and separately Collberg and Sahoo [28], implemented the technique for Java bytecode.

The first step in this algorithm is to extract a vector c from the program code, known as the *extracted vector*. A set of instructions S of n ordered groups of machine instructions is defined. For each group of instructions i, the frequency c_i is computed (by dividing the number of occurrences of the group by the size of the code). The extracted vector is the entity that is watermarked [159].

A watermark vector w of the same length as c is randomly generated and the program code is iteratively modified such that each iteration produces a vector which is closer to $c+w$; eventually giving us a new vector c' that is equal to $c + w$. Each modification to the code must preserve the semantic behaviour; this is achieved by using a codebook of semantically equivalent instructions and semantics-preserving code transformations.

The recognition algorithm confirms the existence or non-existence of a watermark, above a detection threshold.

The watermark should not be embedded in every method in a program to make it harder to detect [71]. There is then the problem of identifying which methods are watermarked, during extraction. Hachez [71] use a heuristic algorithm using attributes of methods such as the maximum stack size, the number of exceptions, size of code, etc – they found that 95% of watermarks were able to be recovered, even in heavily obfuscated code.

The transformation, to coerce the c to c', used by Hachez [71] is a basic re-numbering of local variables. For example, the instructions in the following sequence:

Listing 3.12: 'Example Pseudo-Assembly code'

```
iload_0
iload_1
iadd
istore_2
iload_2
iload_1
imul
istore_1
```

could be replaced by:

Listing 3.13: 'Example Pseudo-Assembly code'

```
iload_1
iload_2
iadd
istore_0
iload_0
iload_2
imul
istore_2
```

Both are semantically equivalent but local variable slots are re-numbered. Using this technique, the frequency of a group of instructions such as `{iload_1,iload_2}` can be increased or decreased. The re-numbering of local variables is similar to register allocation based watermarking algorithms and is highly suseceptible to semantics-preserving transformations which affect register allocation [76].

Collberg and Sahoo [28] implemented a version of graph theoretic watermarking ROW_{SM} in Sandmark [26]. ROW_{SM} uses different techniques to change the frequency of instructions including code cloning by method overloading or over-riding and semantically-equivilent code-replacement.

For example, a method `void P()` can be overloaded by a method `void P(int x)`; this new method is never actually executed. However, this then becomes a problem of protecting the new method from dead-code analysis attacks, possibly using opaque predicates [32].

A code-book of semantically equivalent instructions is built manually including:

$$X = 0 \Longrightarrow X = 0/Y \quad (where\ Y\ is \neq 0) \tag{3.7}$$

$$\ldots X \ldots \Longrightarrow X = X - 1; X = X + 1; \ldots X \ldots \tag{3.8}$$

The purpose of these transformations is to increase the frequency of instruction groups, in order to change the vector. Applying any of the transformations will have the unwanted effect of adding additional instruction groups. For example, consider the instruction group `{isub,istore_x}` – increasing it's frequency will likely involve adding additional instructions.

The transformations must be [27]:

1. be small
2. involve commonly occuring instruction groups
3. not trivially undoable

Finding the perfect transformation is difficult and none of the existing implementations are perfect as many of the transformations are easily undoable by trivial obfuscations [27]. Collberg and Sahoo [28] conclude that their implementation of ROW, while the technique is robust against some obfuscations, is susceptible to many transformation attacks such as decompilation-recompilation and additive attacks.

Curran et al. [40] describe a spread-spectrum technique using a vector derived from the call graph depth of a program. Methods, in a program, are altered such that they call themselves at the correct depth, such as in listing 3.14.

Listing 3.14: Call depth manipulation for spread-spectrum watermarking

```
private int depth = 0;

public int add(int a, int b) {
    if(depth == 0) {
       depth++;
       return add(a, b);
    }else{
       /* original method code */
    }
}
```

In this scheme, the watermark is embedded by "storing distinct points of the call depth of a Java program run on a particular input, then subtracting their mean", determining such calls using a debugger. However, this scheme is not resilient to attacks, as an attacker can easily add their own method calls to the program; thus disrupting watermark detection.

Ai et al. [5] attempt to improve on the original algorithm by introducing a collusion-attack resistant variation. They achieve this by keeping the code-book of transformations secret, implementing their system for Microsoft's Common Intermediate Language.

3.4.2 Conclusion

We have presented a survey of software watermarking schemes based on code replacement, from the basic, early patents [83, 151] to the latest spread-spectrum techniques [5].

Static techniques, as always, are highly susceptible to semantics preserving transformation attacks and are therefore easily removed by an adversary. Even spread-spectrum techniques, which detect a watermark above a certain threshold, are only resilient against minor attacks.

Further research should look at dynamic watermarking as static watermarking schemes are not robust enough for intellectual property protection.

3.5 Abstract Interpretation

3.6 Threads

3.7 Execution Path

3.8 Slicing Based

Chapter 4

Evaluation of Static Watermarking Algorithms

We evaluate the existing static watermarking software by watermarking 60 *jar* files with all available watermark algorithms and then apply a distortive attack to each watermarked program, by obfuscating and optimising. After all the programs have been transformed we attempt to extract the watermarks from the programs. We expect that many watermarks will be lost during the transformations and attempt to find which transformations most affect the watermarks.

4.1 The Watermarkers

We are testing 14 different static watermarking algorithms from 3 different watermarking systems: Sandmark, Allatori and DashO. The latter two are commercial systems, while the former is an academic open-source framework (table 4.1).

4.1.1 Sandmark

SandMark [26] is a tool developed by Christian Collberg *et al.* at the University of Arizona for research into software watermarking, tamper-proofing, and code obfuscation of Java bytecode. The project is open-source and both binaries and source-code can be download from the SandMark homepage [26].

4.1.2 Allatori

Allatori [155] is a commercial Java obfuscator complete with a watermarking system created by Smardec [154]. The company claim that 'if it is necessary for you to protect your software, if you want to reduce its size and to speed up its work, Allatori obfuscator is your choice' [155].

4.1.3 DashO

DashO [**?**] is a commerical Java security solution, including obfuscator, watermarking and encrypter - similar to Allatori. DashO is made by PreEmptive Solutions [**?**] who claim that 'DashO provides advanced Java obfuscation and optimization for your application'.

name	type	version	year
Sandmark	open-source	3.4.0	2004
Allatori	commercial	2.8	2009
DashO	commercial	6.3.3	2010

Table 4.1: The Watermarking Systems

4.2 The Watermark Algorithms

Sandmark contains 12[1] static Java bytecode watermarking algorithms [25]:

1. **Add Expression** adds a bogus addition expression containing the watermark to a class-file.
2. **Add Initialization** adds bogus local variables to a method in a class-file.
3. **Add Method and Field** splits a watermark in two - one half stored in the name of a bogus field, the other half store in the name of a bogus method. The new method accesses the field, while a randomly chosen method calls the new method to make it seem like they are part of the program.
4. **Add Switch** embeds the watermark in the case values of a switch statement, inserted at the beginning of a randomly chosen method.
5. **Davidson/Myhrvold** [44] embeds the watermark by re-ordering basic blocks in a suitable method.
6. **Graph Theoretic Watermark** [167] embeds the watermark in a control-flow graph, which is added to the original program.
7. **Monden** [121] embeds the watermark by replacing opcodes in a dummy method, generated by Sandmark.
8. **Qu/Potkonjak** [142, 141, 124] embeds the watermark in local variable assignments by adding constraints to the interference graphs.
9. **Register Types** embeds a watermark by introducing local variables of certain Java standard library types.
10. **Static Arboit** [9, 125] embeds a method by encoding the watermark in an opaque predicate and then appending the predicate to a selected branch.
11. **Stern (Robust Object Watermarking)** [159, 28] embeds the watermark as a statistical object by creating a frequency vector representation of the code.
12. **String Constant** is a simple watermarking algorithm which simply embeds the watermark string into the constant pool of a class-file.

13) **Dash-O Pro** renames classes and inserts some extra static code in each of the class-files.

14) **Allatori** embeds watermarks as a sequence of *push* and *pop* operations inserted into multiple class-file methods.

4.3 The Obfuscation Algorithms

Sandmark contains a variety of semantics preserving obfuscation algorithms which we will use to evaluate the watermarking systems. We also use Proguard [101] to optimise the test programs, as another form of obfuscation. In total, there are 37 different transformations to be applied.

4.4 The Jar files

All the jar files that we use in the tests are plugins for the open-source text editor jEdit [**?**]. These files are fairly small (average 30KB) but represent a collection of real-world Java software[2] (see table 4.4). The range of plugins represent a variety of code, and were all written by different programmers but as they are plugins they share some characteristics. For example, some classes may subclass jEdit's abstract plugin classes to use jEdit's plugin API. All the test files were obtained by installing jEdit and then using the built-in plugin manager to download the plugin jar files. The average number of classes per jar is 11, while the average number of methods per jar is 66.

[1] A 13th static algorithm is included, but not counted here - Stenagoraphy. This algorithm stores a watermark within PNG files in the program jar file.

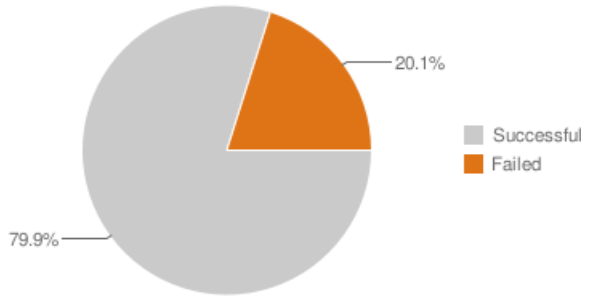
[2] we found that larger files cause problems with Sandmark's obfuscator resulting in crashes and/or extremely long embed times

4.5 Results

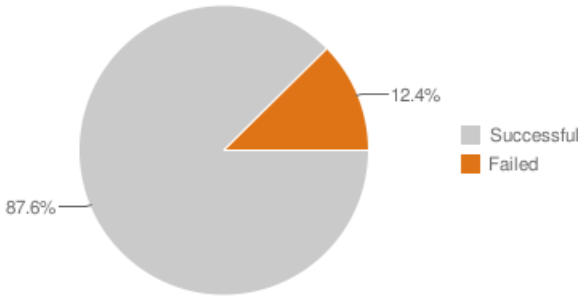
4.5.1 Watermarking

After embedding watermarks we obtained 671 out of an expected 840 watermarked jars. Some watermark algorithms failed to embed the specified watermark, due to error or incompatible program jar. For example, Qu/Potkonjak could only embed watermarks in 1 of the programs because the class files were too small for the watermark. Allatori, String Constant and Add Expression managed to correctly embed watermarks in all 60 test programs - they were embedded and recognised correctly. Only 79.9% of the expected watermarked jar files were actually produced (see figure 4.1).

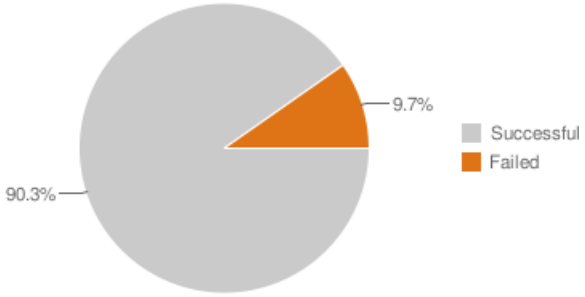
Out of the 671 watermarked jar files only 588 contained watermarks which were successfully recognised before the transformation attacks were applied. This means only 87.6% of the watermarks in the watermarked jar files produced were actually recognised (see figure 4.1).



(a) Watermarks Embeds



(b) Watermarks Recognitions



(c) Obfuscations

Figure 4.1: Watermark and Obfuscation success. Out of the 840 expected watermarked jars, only 671 were produced by the watermarkers (a), while only 588 of these were correctly recognised (b). Out of the 26,169 expected attacked watermarked jars only 23,626 were produced (c).

4.5.2 Obfuscation

We obfuscated the 671 jar files with 36 obfuscations, 1 optimisation and 2 obfuscation combinations which should have resulted in 26,169 attacked watermarked jars. Some algorithms failed to output some jars so we actually obtained 23,626 attacked watermarked jars using 39 semantics preserving transformations. We believe this is due to bugs in the implementation rather than a fundamental problem with the algorithms. This means only 90.3% of the expected attacked watermarked jar files were actually produced (see figure 4.1).

4.5.3 Recognition

The result of recognising the watermarks in the obfuscated jar files are shown in table 4.3. The number of successful recognitions before transformations is shown in the first column, while the remaining columns show the number of successful recognitions after transformations.

A number of zeros can be seen throughout the table indicating that no watermarks was recognised with that combination of the watermark and transformation. These are the combinations of watermark and transformation that we are interested.

4.5.4 Analysis

By examining the table we can see that Proguard Optimizer produces the best results overall - with a low number of recognitions for all watermarkers, except String Constant. We can also see that some of the other transformations remove some of the other watermarks completely. We therefore used a combination of well performing watermarks to remove more watermarks overall (see table 4.2).

Transformation	**Combo 1**	**Combo 2**
Array Folder		
Array Splitter		
Block Marker		
Constant Pool Reorderer		✓
Dynamic Inliner		
FalseRefactor		
Integer Array Splitter		
Interleave Methods		✓
Overload Names	✓	✓
ParamAlias		
Rename Registers	✓	✓
Split Classes		✓
String Encoder		✓
Class Splitter		✓
Field Assignment		
Method Merger		
Objectify		
Publicize Fields		
Simple Opaque Predicates		✓
Static Method Bodies		
Bludgeon Signatures		
Boolean Splitter		✓
Branch Inverter		
Duplicate Registers		
Insert Opaque Predicates		✓
Irreducibility		✓
Merge Local Integers		✓
Opaque Branch Insertion		✓
Promote Primitive Registers	✓	✓
Promote Primitive Types		
Random Dead Code		
Reorder Instructions		
Reorder Parameters		
Transparent Branch Insertion		
Variable Reassigner		✓
Inliner		✓
Proguard Optimize	✓	✓

Table 4.2: The combinations of transformations used for Combo 1 and Combo 2.

The results of running this combination of transformations are shown at the end of table 4.3, in the 'Combo 1' column. This removes many of the watermarks, leaving just 71 remaining and some watermark algorithms with no remaining watermarks. We then generated 'Combo 2' by selecting transformations which contained files in 'Combo 1' but which had the watermark removed.'Combo 2' removed some more

of the remaining watermarks resulting in just 53 files containing watermarks and the Add Switch, Davidson/Myhrvold, Monden and Allatori watermark algorithms completely defeated, compared to 'Combo 1'.

There are still 52 watermarks recognisable after Combo 2 using the 'String Constant' watermark algorithm but these can easily be removed. The String Constant algorithm creates a new, unused entry in a class-file's constant pool containing the watermark value. We can easily remove unused constant pool items with a simple static analysis and therefore remove the 52 String Constant watermarks.

The last remaining watermarked file contains an 'Add Method and Field' watermark. This jar file caused the obfuscators to crash and therefore could not be obfuscated. We believe that this happened due to bugs in obfuscation implementations rather than a fundamental problem with the algorithms. We therefore suggest that this remaining watermark could be removed if the obfuscation implementations were corrected.

A watermarking system can fail in two ways:

1. it fails to embed the watermark.
2. the watermark is easy to remove.

A good watermarking system is one where embedding succeeds often and the watermark is not often removed. Our results show that the static watermarking systems performed badly at embedding and watermarks were easily removed.

4.6 Conclusion

We confirmed that none of the 14 static watermark algorithms are resilient to semantics preserving transformations. A combination of transformations removed all but 52 'String Constant' watermarks and 1 'Add Method and Field' watermark from the test files. 52 of the remaining watermarks can be destroyed by removing (or overwriting) unused constants in a class-file's constant pool. The last watermarked file was rejected by some of the obfuscations and we assume that the watermark in this file would be removed if the bugs in the obfuscations were fixed.

The two commercial watermarkers (Allatori, DashO) available do nothing to identify the owner of our test files as the watermarks are easily removed by our transformation attacks. The fact that both Smardec and PreEmptive Solutions provide the watermarking functionality as part of their software on the basis of code protection is misleading. Although the watermarking functions are just one part of Allatori and DashO, it has also been previously shown that Allatori's string encryption technology is flawed [88]. Additionally, a review of a previous version of DashO, in 2008, concluded that 'the obfuscation mechanisms were not very strong when compared to its peer obfuscators' [136].

Software watermarking must be supplemented with other forms of protection [157], such as obfuscations or tamper-proofing techniques [36], in order to better protect a program from copyright infringement and decompilation.

We have shown that static watermarks are insufficient to prove ownership of software due to their lack of resilience to semantics preserving transformations.

	Original	Array Folder	Array Splitter	Block Marker	Constant Pool Reorderer	Dynamic Inliner	FalseRefactor	Integer Array Splitter	Interleave Methods	Overload Names	ParamAlias	Rename Registers	Split Classes	String Encoder	Class Splitter	Field Assignment	Method Merger	Objectify	Publicize Fields	Simple Opaque Predicates	Static Method Bodies	Bludgeon Signatures	Boolean Splitter	Branch Inverter	Duplicate Registers	Insert Opaque Predicates	Irreducibility	Merge Local Integers	Opaque Branch Insertion	Promote Primitive Registers	Promote Primitive Types	Random Dead Code	Reorder Instructions	Reorder Parameters	Transparent Branch Insertion	Variable Reassigner	Inliner	Proguard Optimize	Combo 1	Combo 2
Add Expression	60	60	60	60	57	60	60	60	57	60	60	0	28	60	60	60	60	60	60	60	24	60	60	60	60	60	59	56	60	0	7	47	60	60	59	1	10	2	0	0
Add Initialization	56	56	56	56	54	56	56	56	55	56	56	56	56	56	44	56	56	56	56	55	55	56	55	56	5	56	56	0	0	0	7	56	10	51	48	56	56	1	0	0
Add Method and Field	35	35	35	35	33	30	35	35	7	6	34	35	29	35	23	32	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	35	27	35	6	1
Add Switch	59	59	59	59	55	59	59	59	59	59	59	59	59	59	58	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	59	1	1	0
Davidson/Myhrvold	15	15	12	14	13	15	15	15	12	15	12	15	7	8	15	15	15	13	15	15	15	11	12	13	8	7	15	11	6	4	4	14	13	13	9	2	13	8	3	0
Graph Theoretic Watermark	47	47	47	47	45	45	47	47	29	47	47	47	46	47	47	47	47	47	47	1	47	47	47	47	47	33	0	47	1	0	5	47	47	47	2	47	45	0	0	0
Monden	58	56	58	56	55	58	58	58	28	58	58	58	58	58	58	58	57	58	58	7	58	58	48	55	56	31	44	25	32	44	46	56	58	58	58	58	56	7	5	0
Qu/Potkonjak	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Register Types	51	51	51	51	49	49	51	51	50	9	51	0	15	51	32	51	51	6	51	51	5	51	51	51	51	51	51	51	51	51	51	51	51	51	51	1	8	1	0	0
Static Arboit	19	19	19	19	18	12	19	19	3	19	19	19	19	19	19	19	1	19	19	0	19	19	19	19	19	3	19	19	0	0	2	19	19	19	10	19	2	2	0	0
Stern	46	43	46	45	44	44	45	46	39	45	46	45	45	46	45	45	45	45	45	1	45	45	46	45	45	10	26	16	42	0	5	22	46	45	45	45	42	2	0	0
String Constant	60	60	60	60	57	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	60	55	52
Dash-O Pro	22	4	11	0	8	8	0	2	6	4	0	0	4	11	6	0	0	2	0	1	0	2	9	0	0	10	9	0	0	0	2	0	9	2	0	1	7	22	0	0
Allatori	60	60	60	60	57	54	60	60	59	60	60	59	60	60	60	60	60	60	60	60	60	59	60	60	60	60	60	60	56	60	60	60	60	59	59	60	58	1	1	0

Table 4.3: Evaluation results - along the top is the name of the transformation performed and along the left is the name of the watermark system.

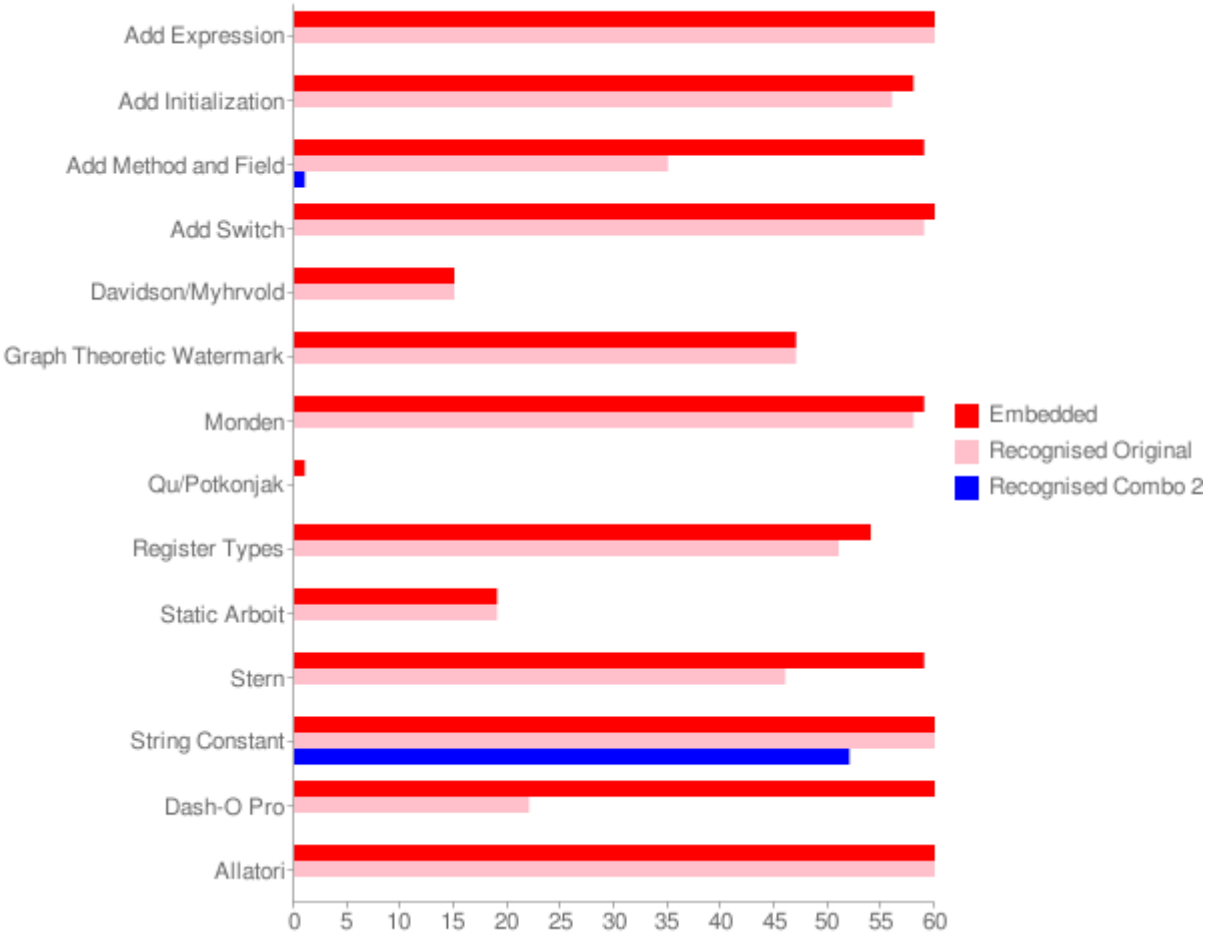


Figure 4.2: The number of files in which watermarks were correctly embedded and recognised. Not all the embedded watermarks were correctly recognised before transformation. This is due to the fact that some watermarking algorithms require a minimum size class-file to store the watermark. This led to some watermarks being embedding incorrectly or a subset of the original watermark being embedded. The number of recognitions before the files were attacked is much higher than after applying the 'Combo 2' combination of transformations. In fact, all but 53 watermarks were destroyed.

Filename	Size (KB)	Classes	Methods	Fields	Locals
Accents	18.4	6	33	16	96
Activator	21.1	17	86	47	212
Ancestor	4.9	5	16	9	48
AxisHelper	12.9	7	39	33	90
Background	11.0	6	35	26	100
BufferLocal	13.1	5	31	20	105
BufferSelector	15.2	9	44	29	110
CheckStylePlugin	4.7	3	19	9	47
CodeLint	12.4	3	19	11	82
CommentFolder	3.3	2	4	3	15
CommonControls	271.6	62	436	218	1189
ConfigurableFoldHandler	24.0	15	83	53	223
ContextHelp	16.7	2	21	48	87
ContextMenu	20.3	11	64	32	136
DBTerminal	23.6	22	101	56	203
Dict	10.9	6	40	31	89
GroovyScriptEnginePlugin	3.6	1	5	1	5
HelperLauncher	7.3	4	23	10	63
HexEdit	22.7	27	137	35	321
Hyperlinks	17.0	18	74	34	194
IncludesParser	22.3	15	51	43	123
InformSideKick	24.2	10	78	82	252
JFuguePlugin	24.7	12	51	12	84
JNAPlugin	1.9	1	3	0	3
JVMStats	4.8	4	11	16	35
JalopyPlugin	22.2	22	70	13	117
JavaFold	5.2	3	10	9	50
JavaInsight	26.6	10	52	20	237
JavaScriptShell	27.6	6	38	7	103
JavascriptScriptEnginePlugin	3.6	1	5	1	5
JcrontabPlugin	19.1	11	52	35	133
JinniConsole	7.9	5	37	13	86
LineGuides	15.3	8	57	24	156
LispPaste	8.4	7	25	20	68
MacOSX	8.9	4	29	8	94
MetalColor	9.0	4	36	40	68
MibSideKick	8.5	6	23	9	60
MouseSnap	4.6	1	8	3	19
MyDoggyPlugin	24.1	17	94	46	237
Nested	15.9	12	47	23	132
NetComponents	17.8	21	137	49	336
Optional	15.4	11	68	29	226
Outline	4.6	4	13	7	33
PerlSideKick	7.2	2	4	10	15
ProjectViewer	712.1	169	1103	523	3004
PrologConsole	17.6	3	22	7	60
RETest	16.9	6	54	31	121
RecentBufferSwitcher	10.6	6	31	9	87
RecursiveOpen	8.3	4	17	9	48
Rename	5.0	6	18	14	49
SaxonAdapter	7.6	3	20	13	67
SaxonPlugin	26.3	1	1	0	1
ScriptEnginePlugin	21.2	7	54	21	166
SendBuffer	5.5	2	7	11	34
ShortcutDisplay	10.9	9	39	18	85
Sudoku	16.8	17	62	50	221
SuperScript	27.6	14	70	39	201
SwitchBuffer	22.3	17	66	43	171
TableLayout-20050920	10.1	5	70	47	296
TomcatSwitch	17.9	7	60	45	159
Average	30.0	11	66	35	180

Table 4.4: Test file statistics.

Chapter 5

Current Progress and Thesis Plan

5.1 PhD Road Map

Figure 5.1, page 112 shows a road map for completion of my PhD thesis.

5.2 Thesis Plan

Introduction Introduce software piracy, decompilation, software watermarking, Java, slicing, etc

Empirical Evaluation of Java Decompilers A study of Java decompilers detailing strengths and weaknesses, showing the need for software protection techniques. Based on chapter 2 and IEEE SCAM2009 paper [74] .

A Survey of Software Watermarking A survey of static and dynamic software watermarking, based on chapter 3.

Empirical Evaluation of Static Watermarking Algorithms Based on chapter 4 and WCECS2010 paper [75]. Evaluating algorithms implemented in Sandmark [26].

Empirical Evaluation of Dynamic Watermarking Algorithms Similar to previous chapter but focusing on dynamic watermarking algorithms.

Investigation of Subtractive Attacks Using Slicing The main contribution of the thesis will be this chapter. It will contain methods of using program slicing [169] to remove dynamic software watermarks while preserving the original program semantics.

Investigating Slicing Resistant Watermarks Investigation concerning the development of slicing resistant watermarking algorithms.

Conclusion and Future Work Suggestions for further work might include thoughts concerning further development of IP techniques for Java.

5.3 Program Slicing

Program slicing [169] is a program transformation technique which computes a set of program statements, known as a *program slice*, that affect a point of interest known as the *slicing criterion*. A program slice can be used for debugging, allowing a programmer to focus on a subset of the program. For example, a programmer can select a certain variable as the slicing criterion and produce a slice which contains all the code that affects the variable.

Definition 13. *Program Slice [169]: An independent program guaranteed to faithfully represent the original program within the domain of the specified subset of behaviour.*

Definition 14. *Slicing Criterion [169]: A slicing criterion is specified as a program statement and a set of variables which specify the behaviour of interest.*

Program slicing can be used to perform a *subtractive attack* against a watermarked program; that is, if we know the location of the watermark we can slice the program at that point leaving a semantically correct unwatermarked program.

Previously, slicing has been suggested as a method of attacking code obfuscations and resilient obfuscations have been designed based on program slicing [110, 46]. Research towards my PhD will include using slicing in a similar way to remove and suggest improvements for software watermarking techniques.

5.4 Finding Watermarks

It may be possible to find watermarks by analysing the execution of a program. For example, if a watermark is protected by an opaque predicate we can analyse many executions of the program to determine which branches are not taken. Previous research has studied such program invariants [57] and Daikon [58] is an example of an automatic invariant detector.

Definition 15. *Opaque Constructs [32]: A variable V is* opaque *at a point p in a program, if V has a property q at p which is known at obfuscation time. A predicate P is opaque at p if its outcome is known at obfuscation time.*

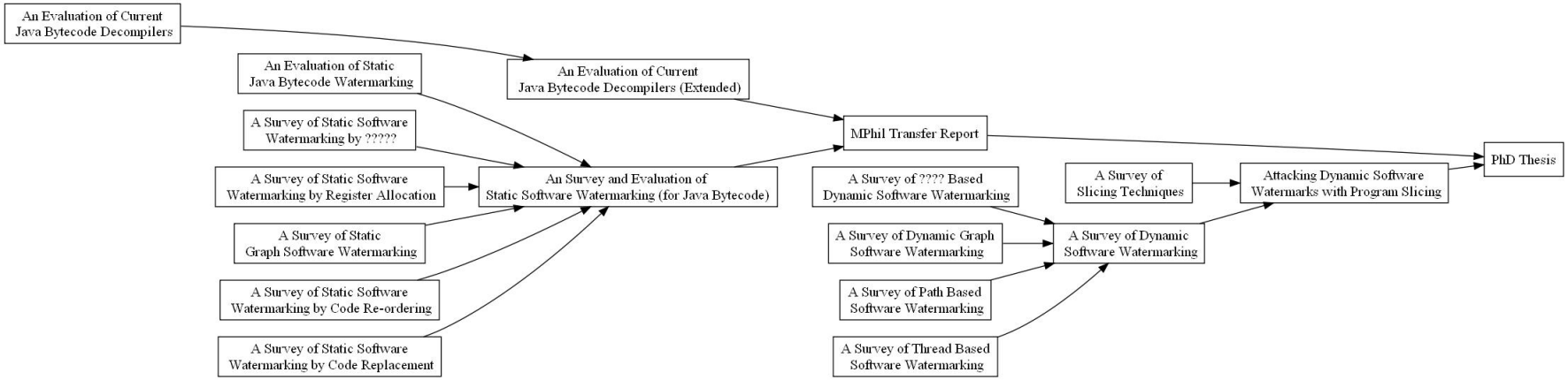


Figure 5.1: PhD Road Map

Appendix A

Decompiled Program Listings

Listing A.1: Test Result: Dava - Exceptions test program

```
import java.io.PrintStream;

class Exceptions2
{

    public static void main(String[]  r0)
    {

        Exceptions2 $r1;
        $r1 = new Exceptions2();
    }

    public void Exceptions2()
    {


        System.out.println(   );
    }
}
```

Listing A.2: Test Result: Dava - Casting test program

```
import java.io.PrintStream;

public class Casting
{

    public static void main(String[]  r0)
    {

        char c0;
        for (c0 =          ; c0 <          ; c0 = (char) (c0 + 1))
        {
            System.out.println((new StringBuilder()).append(        ).append(c0).append(
                               ).append(c0).toString());
        }
    }
}
```

Listing A.3: Test Result: Dava - Args test program

```
import java.io.PrintStream;

class Args
{

    public static void main(String[]  r0)
    {

        byte b0;
        b0 = (byte) (byte) 0;
        System.out.println(b0);
    }
```

Listing A.4: Test Result: Dava - ControlFlow test program

```
import java.io.PrintStream;

public class ControlFlow
{

    public static int foo(int  i0, int  i1)
    {

        int $i2;
        label_0:
        while (true)
        {
            try
            {
                if (i0 < i1)
                {
                    $i2 = i1;
                    i1++;
                    i0 = $i2 / i0;
                    continue label_0;
                }
            }
            catch (RuntimeException $r1)
            {
                i0 = 10;
                continue label_0;
            }

            return i1;
        }
    }

    public static void main(String[]  r0)
    {


        System.out.println(ControlFlow.foo(1, 2));
    }
}
```

Listing A.9: Test Result: Dava - Extract 1 from connectfour test program

```
boolean $z4, z5;
....
if (($z4 & $z5) != 0) {
    r1.all = 1;
}
```

```
}
```

Listing A.5: Test Result: Dava - Sable test program

```
public class Sable
{

    public static void f(short  s0)
    {

        Rectangle r0;
        boolean z0;
        Drawable r1;
        Circle r2;
        label_0:
        {
            while (s0 <= 10)
            {
                r2 = new Circle(s0);
                z0 = r2.isFat();
                r1 = r2;
                break label_0;
            }

            r0 = new Rectangle(s0, s0);
            z0 = r0.isFat();
            r1 = r0;
        } //end label_0:


        label_1:
        {
            while (z0)
            {
                break label_1;
            }

            r1.draw();
        } //end label_1:

    }

    public static void main(String[]  r0)
    {


        Sable.f(11);
    }
}
```

Listing A.6: Test Result: Dava - Usa test program

```
public class Usa
{
    public String name;

    private final void this()
    {


        name =          ;
    }

    public Usa()
    {

        this.this();
    }
}
```

Listing A.7: Test Result: Dava - TryFinally test program. Variable $r5 was originally $r4.

```
import java.io.PrintStream;

public class TryFinally
{

    public static void main(String[]  r0)
    {

        Throwable r2;
        try
        {
            System.out.println(     );
        }
        catch (Throwable $r4)
        {
            label_0:
            while (true)
            {
                try
                {
                    r2 = $r4;
                }
                catch (Throwable $r4)
                {
                    continue label_0;
                }

                System.out.println(         );
                throw r2;
            }
        }

        System.out.println(         );
    }
}
```

Listing A.18: Test Result: Jad - Extract 1 from connectfour test program

```
        if(_loop_index == _index_max)
            break; /* Loop/switch isn't completed */
         _src_tmp[_loop_index];
           goto _L1
_L3:
         _trg_tmp[_loop_index];
         _trg_tmp[_loop_index];
_L1:
         0;
         if(true) goto _L3; else goto _L2
_L2:
         JVM INSTR arraylength .length;
         0;
         JVM INSTR swap ;
         System.arraycopy();
         _loop_index++;
         if(true) goto _L5; else goto _L4
_L4:
         return _result;
```

Listing A.8: Test Result: Dava -Optimised test program

```
public class Optimised
{

    public static void f(short  s0)
    {

        Rectangle r0;
        boolean z0;
        Drawable r1;
        Circle r2;
        label_0:
        {
            while (s0 <= 10)
            {
                r2 = new Circle(s0);
                z0 = r2.isFat();
                r1 = r2;
                break label_0;
            }

            r0 = new Rectangle(s0, s0);
            z0 = r0.isFat();
            r1 = r0;
        } //end label_0:


        label_1:
        {
            while (z0)
            {
                break label_1;
            }

            r1.draw();
        } //end label_1:

    }

    public static void main(String[]  r0)
    {


        Optimised.f(11);
    }
}
```

Listing A.10: Test Result: Jad - Sable test program

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)
// Source File Name:   Sable.java


public class Sable
{

    public Sable()
    {
    }

    public static void f(short word0)
    {
        Object obj;
        boolean flag;
        if(word0 > 10)
        {
            Rectangle rectangle = new Rectangle(word0, word0);
            flag = rectangle.isFat();
            obj = rectangle;
        } else
        {
            Circle circle = new Circle(word0);
            flag = circle.isFat();
            obj = circle;
        }
        if(!flag)
            ((Drawable) (obj)).draw();
    }

    public static void main(String args[])
    {
        f((short)11);
    }
}
```

Listing A.11: Test Result: Jad - ControlFlow test program

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)
// Source File Name:   ControlFlow.java

import java.io.PrintStream;

public class ControlFlow
{

    public ControlFlow()
    {
    }

    public int foo(int i, int j)
    {
        do
            try
            {
                for(; i < j; i = j++ / i);
                break;
            }
            catch(RuntimeException runtimeexception)
            {
                System.out.println(runtimeexception);
                i = 10;
            }
        while(true);
        return j;
    }
}
```

Listing A.12: Test Result: Jad - Casting test program

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)

import java.io.PrintStream;

public class Casting
{

    public Casting()
    {
    }

    public static void main(String args[])
    {
        for(char c =     ; c < 128; c++)
            System.out.println((new StringBuilder()).append(        ).append(c).append(
                    ).append(c).toString());

    }
}
```

Listing A.13: Test Result: jad - Usa test program

```
// Decompiled by DJ v3.9.9.91 Copyright 2005 Atanas Neshkov  Date: 11/01/2009 15:15:53
// Home Page : http://members.fortunecity.com/neshkov/dj.html  - Check often for new version!
// Decompiler options: packimports(3)
// Source File Name:   Usa.java

import java.io.PrintStream;

public class Usa
{
    public class England
    {
        public class Ireland
        {

            public void print_names()
            {
                System.out.println(name);
            }

            public String name;
            final England this$1;

            public Ireland()
            {
                this$1 = England.this;
                super();
                name =         ;
            }
        }


        public String name;
        final Usa this$0;

        public England()
        {
            this$0 = Usa.this;
            super();
            name =         ;
        }
    }


    public Usa()
    {
        name =          ;
    }

    public String name;
}
```

Listing A.14: Test Result: Jad - Exceptions test program

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)
// Source File Name:   Exceptions2.jasmin

import java.io.PrintStream;

class Exceptions
{

    Exceptions()
    {
    }

    public static void main(String args[])
    {
        new Exceptions();
    }

    public void Exceptions()
    {
        System.out.println(   );
        System.out.println(   );
        try
        {
            System.out.println(   );
            System.out.println(   );
        }
        // Misplaced declaration of an exception variable
        catch(Exceptions this)
        {
            System.out.println(   );
        }
_L2:
        System.out.println(   );
        return;
        this;
        System.out.println(   );
        if(true) goto _L2; else goto _L1
_L1:
    }
}
```

Listing A.15: Test Result: Jad - Optimised test program

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)
// Source File Name:   Optimised.java


public class Optimised
{

    public Optimised()
    {
    }

    public static void f(short arg0)
    {
        Object obj;
        if(arg0 > 10)
        {
            obj = JVM INSTR new #43  <Class Rectangle>;
            ((Rectangle) (obj)).Rectangle(arg0, arg0);
            arg0 = ((Rectangle) (obj)).isFat();
            obj = obj;
        } else
        {
            obj = JVM INSTR new #19  <Class Circle>;
            ((Circle) (obj)).Circle(arg0);
            arg0 = ((Circle) (obj)).isFat();
            obj = obj;
        }
        if(arg0 == 0)
            ((Drawable) (obj)).draw();
    }

    public static void main(String arg0[])
    {
        f((short)11);
    }
}
```

Listing A.16: Test Result: Jad - TryFinally test program

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)
// Source File Name:   TryFinally.java
//Overlapped try statements detected. Not all exception handlers will be resolved in the method
    main
//Couldn't fully decompile method main
//Couldn't resolve all exception handlers in method main

import java.io.PrintStream;

public class TryFinally
{

    public TryFinally()
    {
    }

    public static void main(String args[])
    {
        System.out.println(     );
        System.out.println(         );
        break MISSING_BLOCK_LABEL_30;
        Exception exception;
        exception;
        System.out.println(         );
        throw exception;
    }
}
```

Listing A.17: Test Result: Jad - Args test program

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)
// Source File Name:   Args.jasmin

import java.io.PrintStream;

class Args
{

    Args()
    {
    }

    public static void main(String args[])
    {
        args = 0;
        System.out.println(args);
    }
}
```

Listing A.19: Test Result: Java Decompiler - Casting test program

```
import java.io.PrintStream;

public class Casting
{
  public static void main(String[] paramArrayOfString)
  {
    for (char c =       ; c <    ; c = (char)(c +       ))
      System.out.println(         + c +               + c);
  }
}
```

Listing A.20: Test Result: Java Decompiler - InnerClass test program

```
import java.io.PrintStream;
public class Usa {
  public String name;
  private final void jdMethod_this() {
    this.name =          ;
  }
  public Usa() {
    jdMethod_this();
  }
  public class England { public String name;
    private final void jdMethod_this() { this.name =         ; }
    public England() {
      jdMethod_this();
    }
    public class Ireland {
      public String name;
      public void print_names() { System.out.println(this.name); }
      private final void jdMethod_this() {
        this.name =         ;
      }
      public Ireland() {
        jdMethod_this();
      }
    }
  }
}
```

Listing A.26: Test Result: Java Decompiler - Extract 1 from connectfour test program

```
standard.access_string tmp10_7 = new jgnat/adalib/standard$access_string;
tmp10_7.<init>();
```

Listing A.27: Test Result: Java Decompiler - Extract 2 from connectfour test program

```
public static void _elabb() throws {
```

Listing A.35: Test Result: jdec - Extract 1 from connectfour test program

```
int[] JdecGenerated173 = new int[]{    _aggr =(JdecGenerated173);71,143,214,286,357,429,500};
```

Listing A.21: Test Result: Java Decompiler - Sable test program

```
public class Sable {
  public static void f(short paramShort) {
    Object localObject;
    if (paramShort > 10) {
    localObject = new Rectangle;
      ((Rectangle)localObject).<init>(paramShort, paramShort);
      paramShort = ((Rectangle)localObject).isFat();
      localObject = localObject;
    } else {
      localObject = new Circle;
      ((Circle)localObject).<init>(paramShort);
      paramShort = ((Circle)localObject).isFat();
      localObject = localObject;
    }
    if (paramShort != 0)
      return;
    ((Drawable)localObject).draw();
  }

  public static void main(String[] paramArrayOfString) {
    f(11);
  }
}
```

Listing A.22: Test Result: Java Decompiler - ControlFlow test program

```
public class ControlFlow {
  public static int foo(int paramInt1, int paramInt2) {
    while (true) {
      try {
        while (paramInt1 < paramInt2)
          paramInt1 = paramInt2++ / paramInt1;
      } catch (RuntimeException localRuntimeException) {
        paramInt1 = 10;
        break label35:
      }
      label35: break;
    }
    return paramInt2;
  }

  public static void main(String[] paramArrayOfString) {
    System.out.println(foo(1, 2));
  }
}
```

Listing A.23: Test Result: Java Decompiler - Exceptions test program

```
import java.io.PrintStream;

class Exceptions {
  public static void main(String[] paramArrayOfString) {
    new Exceptions();
  }

  public void Exceptions() {
    System.out.println(   );
    try {
      System.out.println(   );
    } catch (java.lang.RuntimeException this) {
      try {
        System.out.println(   );
        System.out.println(   );
        label32: System.out.println(   );
        return;
      } catch (java.lang.Exception this) {
        System.out.println(   );
        break label32:
        this = this;
        System.out.println(   );
      }
    }
  }
}
```

Listing A.24: Test Result: Java Decompiler - Optmised test program

```
public class Optimised
{
  public static void f(short arg0)
  {
    Object localObject;
    if (arg0 > 10)
    {
      localObject = new Rectangle;
      ((Rectangle)localObject).<init>(arg0, arg0);
      arg0 = ((Rectangle)localObject).isFat();
      localObject = localObject;
    }
    else
    {
      localObject = new Circle;
      ((Circle)localObject).<init>(arg0);
      arg0 = ((Circle)localObject).isFat();
      localObject = localObject;
    }
    if (arg0 == 0)
      ((Drawable)localObject).draw();
  }

  public static void main(String[] arg0)
  {
    f(11);
  }
}
```

Listing A.25: Test Result: Java Decompiler - Args test program

```
import java.io.PrintStream;

class Args
{
  public static void main(String[] paramArrayOfString)
  {
    paramArrayOfString = 0;
    System.out.println(paramArrayOfString);
  }
}
```

Listing A.28: Test Result: jdec - Casting test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File           For more details.

/**** List of All Imported Classes ***/

import java.io.PrintStream;
import java.lang.Object;
import java.lang.StringBuilder;
import java.lang.System;

// End of Import

public  class  Casting

{


//  CLASS: Casting:
 public     Casting( )
 {
    super();
    return;

 }

//  CLASS: Casting:
 public  static    void main(  String [] aString1)
 {
    char char1= 0;
    char1=0;
    while(true)
    {
      if(char1 >= 128)
     {
        break;
      }
      if(char1 < 128)
      {
        StringBuilder JdecGenerated14 = new StringBuilder();
        System.out.println(JdecGenerated14.append(        ).append(char1).append(             ).
            append(char1).toString());
        char1=(char)((char1 + 1));
        continue  ;

      }

    }
    return;

 }


}
```

Listing A.29: Test Result: jdec - Usa test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File           For more details.

/**** List of All Imported Classes ***/

import java.lang.Object;

// End of Import

public class Usa {

 /***
 **Class Fields
 ***/
 public  String name ;

//  CLASS: Usa:
 public Usa() {
    super();
    this.name =         ;
    return;
 }
}
```

Listing A.36: Test Result: jdec - Extract 3 from connectfour test program

```
 public  static    int[][] connectfour$Tboard_array_typeB_deep_copy(  int [][] _target,  int
     _trgstart,  int [][] _source,  int _srccount,  int _srcstart  int [][] _result  int
     _loop_index  int _trgindex  int _srcindex  int [][] _trg_tmp  int [][] _src_tmp  int
     _loop_index  int _index_max) { ... }

 public static void initialize_board(  int [][] board  int _loop_param  int _loop_limit  int
     _loop_param  int _loop_limit  Throwable _exc_var) { ... }

 public static void place_disk(  int [][] board,  int column,  Int row,  int who  Throwable
     _exc_var) { ... }

 public static void check_won(  int [][] board,  int who,  Int won  int _loop_param  int
     _loop_limit  int _loop_param  int _loop_limit  Throwable _exc_var) { ... }

 public static void check_tie(  int [][] board,  Int is_tie  int _loop_param  int _loop_limit
     Throwable _exc_var) { ... }

 public static void computer_turn(  int [][] board,  Int column  standard$access_string []
     value_typeT  standard$access_string _alloc_tmp  byte [] _str_literal  byte [] _str_literal
     byte [] _str_literal  byte [] _str_literal  byte [] _str_literal  byte [] _str_literal
     __AR_connectfour$computer_turn __AR  int [][] new_board  int [] evaluations  int []
     moves_to_unknown  int count_unknowns  int value  int max_value  int best_move  int
     _loop_param  int _loop_limit  int _loop_param  int _loop_limit  int _loop_param  int
     _loop_limit  Int _out_tmp  int R41b  int _loop_param  int _loop_limit  Throwable _exc_var) {
      ... }
```

Listing A.30: Test Result: jdec - Sable test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File           For more details.
// class file compiled with jikes, javac wouldn't decompile
/**** List of All Imported Classes ***/

import java.lang.Object;

// End of Import

public class Sable {
//  CLASS: Sable:
 public Sable() {
    super();
    return;
 }

//  CLASS: Sable:
 public static void f(short short2) {
    Circle aCircle1= null;
    Rectangle aRectangle1= null;
    Rectangle aRectangle2= null;

    int int1= 0;
    if(short2 > 10) {
      Rectangle JdecGenerated8 = new Rectangle(short2,short2);
      aRectangle1=JdecGenerated8;
      int1=aRectangle1.isFat();
      aRectangle2=aRectangle1;
    }else{
      Circle JdecGenerated29 = new Circle(short2);
      aCircle1=JdecGenerated29;
      int1=aCircle1.isFat();
      aCircle2 =aCircle1;
    }
    if(int1==0) {
      aCircle2.draw();
      return ;
    }
 }

//  CLASS: Sable:
 public static void main(String[] aString1) {
    f((short)11);
    return;
 }
}
```

Listing A.31: Test Result: jdec - ControlFlow test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File           For more details.

/**** List of All Imported Classes ***/

import java.lang.Object;
import java.lang.RuntimeException;

// End of Import

public class ControlFlow {

//  CLASS: ControlFlow:
 public ControlFlow() {
    super();
    return;
 }

//  CLASS: ControlFlow:
 public  int foo(int int4,  int int5) {
    int int4= 0;
    while(true) {
      try {
    if(int4 < int5) {
      int4 = (int5++) / (int4);
      continue  ;
    }else{

        }catch(RuntimeException  aRuntimeException1) {

          }
        int4=10;
        continue  ;

      }

    }
    return int5;
 }
}
```

Listing A.32: Test Result: jdec - Exceptions test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File           For more details.

/**** List of All Imported Classes ***/

import java.io.PrintStream;
import java.lang.Object;
import java.lang.RuntimeException;
import java.lang.System;

// End of Import

class  Exceptions2

{


//  CLASS: Exceptions2:
 Exceptions2( )
 {
    super();
    return;

 }

//  CLASS: Exceptions2:
 public  static    void main(  String [] aString1)
 {

    Exceptions2 JdecGenerated2 = new Exceptions2();
    return;

 }

//  CLASS: Exceptions2:
 public void Exceptions2() {
    System.out.println(   );
    try {
      System.out.println(   );
      try {
        System.out.println(   );
        return;
      }catch(RuntimeException  this) {
        System.out.println(   );
      }
    }
 }


}
```

Listing A.33: Test Result: jdec - Optimised test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File           For more details.

/**** List of All Imported Classes ***/

import java.lang.Object;

// End of Import

public  class  Optimised {
//  CLASS: Optimised:
 public     Optimised( )
 {
    super();
    return;
 }
//  CLASS: Optimised:
 public  static    void f()
 {
    Circle Var_1= null;
    Object Var_1= null;
    Rectangle Var_1= null;

    if(arg0 > 10)
    {
      Rectangle JdecGenerated8 = new Rectangle(      Var_1=JdecGenerated8;
arg0,arg0);
      arg0=this.isFat();
      Var_1=this;
    }
    else
    {
      Circle JdecGenerated28 = new Circle(      Var_1=JdecGenerated28;
arg0);
      arg0=this.isFat();
      Var_1=this;
    }
    if(arg0==0)
    {
      this.draw();
      return ;
    }
 }
//  CLASS: Optimised:
 public  static    void main()  {
    f(11);
    return;
 }
}
```

Listing A.34: Test Result: jdec - Args test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File           For more details.

/**** List of All Imported Classes ***/

import java.io.PrintStream;
import java.lang.Object;
import java.lang.System;

// End of Import

class  Args

{


//  CLASS: Args:
 Args( )
 {
    super();
    return;

 }

//  CLASS: Args:
 public  static    void main(  java.lang.String [] aString1)
 {
    java.lang.String [] aString1= 0;
    aString1=0;
    System.out.println(aString1);
    return;

 }


}
```

Listing A.37: Test Result: jdec - Extract 4 from connectfour test program

```
if(board[(_loop_param- 1)][((_loop_param + 1)- 1)] != who) {

}else{

}
```

Listing A.38: Test Result: JODE - TryFinally test program

```
/* TryFinally - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */

public class TryFinally
{
    public static void main(String[] strings) {
  try {
      System.out.println(     );
  } catch (Object object) {
      System.out.println(         );
      throw object;
  }
  System.out.println(         );
    }
}
```

Listing A.50: Test Result: jReversePro - Extract 1 from connectfour test program

```
else if (k == 0) {
}
else if (0 & 1 != 0) {
}
else if (j != 1) {
}
else if (k != 1) {
}
else if (j != 2) {
}
else if (1 | 0 & 1 != 0) {
}
else if (1 & (o ^ 1) != 0) {
}
else if (k != 1) {
}
else if (1 & (o ^ 1) != 0) {
}
```

Listing A.51: Test Result: jReversePro - Extract 2 from connectfour test program

```
for (;n <= m;) {
      else if (iArr[n - 1] == 0) {
      }
      else if (0 & 1 != 0) {
      }
}
```

Listing A.53: Test Result: Mocha - Extract 1 from connectfour test program

```
public connectfour() throws {
    $this.<init>();
}
```

Listing A.39: Test Result: JODE - ControlFlow test program

```
/* ControlFlow - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */

public class ControlFlow
{
    public int foo(int i, int i_0_) {
  for (;;) {
      try {
    for (/**/; i < i_0_; i = i_0_++ / i) {
        /* empty */
    }
    break;
      } catch (RuntimeException runtimeexception) {
    i = 10;
      }
  }
  return i_0_;
    }

}
```

Listing A.40: Test Result: JODE - Exceptions test program

```
Exception while decompiling:jode.AssertError: Exception handlers ranges are intersecting: [16,
    32] and [8, 24].
  at jode.flow.TransformExceptionHandlers.checkTryCatchOrder(TransformExceptionHandlers.java:914)
  at jode.flow.TransformExceptionHandlers.analyze(TransformExceptionHandlers.java:928)
  at jode.decompiler.MethodAnalyzer.analyzeCode(MethodAnalyzer.java:577)
  at jode.decompiler.MethodAnalyzer.analyze(MethodAnalyzer.java:652)
  at jode.decompiler.ClassAnalyzer.analyze(ClassAnalyzer.java:352)
  at jode.decompiler.ClassAnalyzer.dumpJavaFile(ClassAnalyzer.java:624)
  at jode.decompiler.Decompiler.decompile(Decompiler.java:192)
  at jode.swingui.Main.run(Main.java:204)
  at java.lang.Thread.run(Thread.java:619)
```

Listing A.41: Test Result: JODE - Optmised test program

```
/* Optimised - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */

public class Optimised
{
    public static void f(short arg0) {
  boolean bool;
  Drawable drawable;
  if (arg0 > 10) {
      Rectangle rectangle = new Rectangle;
      ((UNCONSTRUCTED)rectangle).Rectangle(arg0, arg0);
      bool = rectangle.isFat();
      drawable = rectangle;
  } else {
      Circle circle = new Circle;
      ((UNCONSTRUCTED)circle).Circle(arg0);
      bool = circle.isFat();
      drawable = circle;
  }
  if (!bool)
      drawable.draw();
    }

    public static void main(String[] arg0) {
  f((short) 11);
    }
}
```

Listing A.42: Test Result: JODE - connectfour test program

```
Exception while decompiling:java.lang.IllegalArgumentException: stack length differs
  at jode.flow.VariableStack.merge(VariableStack.java:77)
  at jode.flow.LoopBlock.mergeContinueStack(LoopBlock.java:454)
  at jode.flow.LoopBlock.mapStackToLocal(LoopBlock.java:440)
  at jode.flow.SequentialBlock.mapStackToLocal(SequentialBlock.java:73)
  at jode.flow.SequentialBlock.mapStackToLocal(SequentialBlock.java:73)
  at jode.flow.FlowBlock.mapStackToLocal(FlowBlock.java:1531)
  at jode.flow.FlowBlock.mapStackToLocal(FlowBlock.java:1515)
  at jode.decompiler.MethodAnalyzer.analyzeCode(MethodAnalyzer.java:580)
  at jode.decompiler.MethodAnalyzer.analyze(MethodAnalyzer.java:652)
  at jode.decompiler.ClassAnalyzer.analyze(ClassAnalyzer.java:355)
  at jode.decompiler.ClassAnalyzer.dumpJavaFile(ClassAnalyzer.java:624)
  at jode.decompiler.Decompiler.decompile(Decompiler.java:192)
  at jode.swingui.Main.run(Main.java:204)
  at java.lang.Thread.run(Thread.java:619)
```

Listing A.43: Test Result: jReversePro - Fibo test program

```
Exception in thread        java.lang.IndexOutOfBoundsException: Index: 2304, Size: 68
  at java.util.ArrayList.RangeCheck(ArrayList.java:547)
  at java.util.ArrayList.get(ArrayList.java:322)
  at jreversepro.reflect.JConstantPool.getCpValue(JConstantPool.java:381)
  at jreversepro.reflect.JConstantPool.getUtf8String(JConstantPool.java:457)
  at jreversepro.parser.JClassParser.readMethodAttributes(JClassParser.java:677)
  at jreversepro.parser.JClassParser.readMethods(JClassParser.java:659)
  at jreversepro.parser.JClassParser.parse(JClassParser.java:199)
  at jreversepro.parser.JClassParser.parse(JClassParser.java:166)
  at jreversepro.parser.JClassParser.parse(JClassParser.java:119)
  at jreversepro.revengine.JSerializer.loadClass(JSerializer.java:80)
  at jreversepro.JCmdMain.loadClass(JCmdMain.java:241)
  at jreversepro.JCmdMain.process(JCmdMain.java:186)
  at jreversepro.JCmdMain.main(JCmdMain.java:159)
```

Listing A.54: Test Result: Mocha - Extract 2 from connectfour test program

```
public static void initialize_board(int board[][]) throws {
    int _loop_param;
    int _loop_limit;
    int _loop_param;
    int _loop_limit;
    expression 1
    _loop_limit = 6;
    pop _loop_param
    if (_loop_param > _loop_limit)
        expression 1
    _loop_limit = 7;
    pop _loop_param
    for (; _loop_param <= _loop_limit; _loop_param++)
        board[_loop_param - 1][_loop_param - 1] = 0;
    _loop_param++;
}
```

Listing A.44: Test Result: jReversePro - Casting test program

```
// JReversePro v 1.4.1 Sun Feb 01 14:23:58 GMT 2009
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions;See the File           for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSiON: 50.0
// SOURCEFILE: null

import java.io.PrintStream;


public class Casting{


    public Casting()
    {
         ;
         return;
    }


    public static void main(String[] stringArr)
    {
         int i = 0;
         for (;i < 128;) {
              System.out.println(new StringBuilder().append(        ).append(i).append(
                                 ).append(i).toString());
              char j = (char)(i + 1);
         }
         return;
    }

}
```

Listing A.45: Test Result: jReversePro - Usa test program

```
// JReversePro v 1.4.1 Sun Feb 01 14:25:55 GMT 2009
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions;See the File           for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSiON: 50.0
// SOURCEFILE: Usa.java


public class Usa{

  public String name;

    public Usa()
    {
         ;
         name =          ;
         return;
    }

}
```

Listing A.46: Test Result: jReversePro - TryFinally test program

```
// JReversePro v 1.4.1 Sun Feb 01 14:21:55 GMT 2009
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions;See the File           for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSiON: 50.0
// SOURCEFILE: null

import java.io.PrintStream;


public class TryFinally{


    public TryFinally()
    {
         ;
         return;
    }


    public static void main(String[] stringArr)
    {
         System.out.println(     );
         System.out.println(         );
         System.out.println(         );
         return;
    }

}
```

Listing A.47: Test Result: jReversePro - Exceptions test program

```
jreversepro.revengine.RevEngineException: Block overlap   false 16 16  35 TYPE_TRY    with
    true 8 8  27 TYPE_TRY
  at jreversepro.runtime.JRunTimeContext.pushControlEntry(JRunTimeContext.java:302)
  at jreversepro.runtime.JRunTimeContext.getBeginStmt(JRunTimeContext.java:209)
  at jreversepro.revengine.JDecompiler.genSource(JDecompiler.java:476)
  at jreversepro.revengine.JDecompiler.genCode(JDecompiler.java:215)
  at jreversepro.reflect.JClassInfo.processMethods(JClassInfo.java:406)
  at jreversepro.reflect.JClassInfo.reverseEngineer(JClassInfo.java:592)
  at jreversepro.JCmdMain.process(JCmdMain.java:323)
  at jreversepro.JCmdMain.process(JCmdMain.java:198)
  at jreversepro.JCmdMain.main(JCmdMain.java:159)
```

Listing A.48: Test Result: jReversePro - Optmised test program

```
// JReversePro v 1.4.1 Tue Feb 17 18:33:46 GMT 2009
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions;See the File           for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSiON: 46.0
// SOURCEFILE: Optimised.java


public class Optimised{


    public Optimised()
    {
        ;
        return;
    }


    public static void f(short i)
    {
        Drawable drawable;
        if (i > 10) {
            Rectangle rectangle = new Rectangle;
            rectangle(i , i);
            i = rectangle.isFat();
            rectangle = rectangle;
        }
        else {
            drawable = new Circle;
            drawable(i);
            i = drawable.isFat();
            drawable = drawable;
        }
        if (i == 0)
            drawable.draw();

        return;
    }


    public static void main(String[] stringArr)
    {
        Optimised.f(11);
        return;
    }

}
```

Listing A.49: Test Result: jReversePro - Args test program

```
// JReversePro v 1.4.1 Sun Feb 01 14:22:50 GMT 2009
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions;See the File           for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSiON: 45.3
// SOURCEFILE: Args.jasmin

import java.io.PrintStream;


class Args{


    Args()
    {
         ;
         return;
    }


    public static void main(String[] stringArr)
    {
         stringArr = 0;
         System.out.println(stringArr);
         return;
    }

}
```

Listing A.52: Test Result: Mocha - Args test program

```
/* Decompiled by Mocha from Args.class */
/* Originally compiled from Args.jasmin */

import java.io.PrintStream;

synchronized class Args
{
    Args()
    {
    }

    public static void main(String astring[])
    {
        astring = 0;
        System.out.println(astring);
    }
}
```

Listing A.55: Test Result: Mocha - Extract 3 from connectfour test program

```
if (who != 1) goto 82 else 10;
```

Listing A.56: Test Result: SourceAgain - Casting test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

import java.io.PrintStream;

public class Casting {

    public static void main(String[] as)
    {
        char c = 0;

        for( c = (char) 0; c < 128; c = (char) (c + 1) )
            System.out.println( new StringBuilder().append(          ).append( c ).append(
                                ).append( c ).toString() );
    }
}
```

Listing A.57: Test Result: SourceAgain - Usa test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

public class Usa {

    public String name =          ;
}
```

Listing A.63: Test Result: SourceAgain - Extract 1 from connectfour test program

```
public connectfour() {
    $this();
}
```

Listing A.64: Test Result: SourceAgain - Extract 2 from connectfour test program

```
public static void initialize_board(int[][] board) {
    int _loop_limit = 6;
    int _loop_param = 0;

    for( _loop_param = 1; _loop_param <= _loop_limit; ++_loop_param ) {
        int _loop_limit = 7;
        int _loop_param = 0;

        for( _loop_param = 1; _loop_param <= _loop_limit; ++_loop_param )
           board[_loop_param - 1][_loop_param - 1] = 0;
    }
}
```

Listing A.58: Test Result: SourceAgain - Sable test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

public class Sable {
Warning #2008: Inheritance relationship can not be determined because Circle can not be found.
Warning #2008: Inheritance relationship can not be determined because Rectangle can not be found.
Warning #2008: Inheritance relationship can not be determined because Drawable can not be found.

    public static void f(short si)
    {
        Object obj = null;
        boolean bool = false;

        if( si > 10 )
        {
            Object obj1 = new Rectangle( si, si );

            bool = ((Rectangle) obj1).isFat();
            obj = obj1;
        }
        else
        {
            Object obj2 = new Circle( si );

            bool = ((Circle) obj2).isFat();
            obj = obj2;
        }
        if( !bool )
            ((Drawable) obj).draw();
    }

    public static void main(String[] as)
    {
        f( (short) 11 );
    }
}
```

Listing A.59: Test Result: SourceAgain - TryFinally test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

import java.io.PrintStream;

public class TryFinally {

    public static void main(String[] as)
    {
        try
        {
            System.out.println(       );
        }
        finally
        {
            try
            {
            }
            finally
            {
                System.out.println(           );
                throw obj;
            }
        }
        System.out.println(           );
    }
}
```

Listing A.60: Test Result: SourceAgain - ControlFlow test program

```
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

public class ControlFlow {

    public int foo(int i, int j)
    {
        for( ;; )
        {
            try
            {
                if( i < j )
                {
                    i = j++ / i;
                    continue;
                }
            }
            catch( RuntimeException runtimeexception1 )
            {
                i = 10;
                continue;
            }
            return j;
        }
    }
}
```

Listing A.61: Test Result: SourceAgain - Exceptions test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

import java.io.PrintStream;

class Exceptions {

    public void Exceptions()
    {
        System.out.println(     );
label_15:
        {
            try
            {
                System.out.println(     );
                try
                {
                    System.out.println(     );
                    break label_15;
                }
                catch( RuntimeException runtimeexception1 )
                {
                    System.out.println(     );
                }
            }
            catch( RuntimeException runtimeexception2 )
            {
                System.out.println(     );
            }
            System.out.println(     );
            return;
        }
        System.out.println(     );
    }

    public static void main()
    {
        Exceptions exceptions1 = new Exceptions();

    }
}
```

Listing A.62: Test Result: SourceAgain - Optmised test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

public class Optimised {
Warning #2008: Inheritance relationship can not be determined because Circle can not be found.
Warning #2008: Inheritance relationship can not be determined because Rectangle can not be found.
Warning #2008: Inheritance relationship can not be determined because Drawable can not be found.

    public static void f(short arg0)
    {
        Object obj = null;
        boolean bool = false;

        if( arg0 > 10 )
        {
            obj = new Rectangle;
            (Rectangle) obj( arg0, arg0 );
            bool = ((Rectangle) obj).isFat();
            obj = obj;
        }
        else
        {
            obj = new Circle;
            (Circle) obj( bool );
            bool = ((Circle) obj).isFat();
            obj = obj;
        }
        if( !bool )
            ((Drawable) obj).draw();
    }

    public static void main(String[] arg0)
    {
        f( (short) 11 );
    }
}
```

Appendix B

Bytecode Analysis Tools

ASM http://asm.ow2.org/

ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or dynamically generate classes, directly in binary form. Provided common transformations and analysis algorithms allow to easily assemble custom complex transformations and code analysis tools.

Soot: a Java Optimization Framework

Soot is a Java optimization framework. It provides four intermediate representations for analyzing and transforming Java bytecode:

1. Baf: a streamlined representation of bytecode which is simple to manipulate. 2. Jimple: a typed 3-address intermediate representation suitable for optimization. 3. Shimple: an SSA variation of Jimple. 4. Grimp: an aggregated version of Jimple suitable for decompilation and code inspection.

Soot can be used as a stand alone tool to optimize or inspect class files, as well as a framework to develop optimizations or transformations on Java bytecode.

Indus http://indus.projects.cis.ksu.edu/

Indus is an effort to provide a collection of program analyses and transformations implemented in Java to customize and adapt Java programs. It is intended to serve as an umbrella for

* static analyses such as points-to analysis, escape analysis, and dependence analyses, * transformations such as program slicing and program specialization via partial evaluation, and * any software module that delivers the analyses/transformations into a particular application such as Bandera or platform such as Eclipse.

Appendix C

Bytecode Generation/Manipulation Tools

A Java virtual machine executes Java bytecode in class files conforming to the class file specification which is part of the Java Virtual Machine Specification [108] and updated for Java 1.6 in JSR202 [16]. This open specification allows tools other than Sun's Java compiler to generate Java bytecode.

This further complicates decompilation as the source of Java bytecode (see Figure **??**, page 38) is not necessarily a Java compiler. Arbitrary bytecode can contain instruction sequences for which there is no valid Java source due to the more powerful nature and less-restrictions in Java bytecode. For example there are no arbitrary control flow instructions in Java but there are in Java bytecode.

C.1 Java → Java Bytecode Compilers

A Java → Java Bytecode compiler must take as input Java source code defined by the Java Language Specification and output Java Bytecode defined by the Java Virtual Machine Specification [108, 16]. There are several such compilers available besides Sun's javac.

javac Sun's Java compiler, javac [**?**], is the original Java compiler from the creators of Java. It is now part of the open-source OpenJDK [**?**].

GNU Java Compiler GCJ [**?**] is an open-source Java compiler which compiles Java to bytecode or native machine code. GCJ can also compile Java bytecode to machine code.

Jikes Jikes [**?**] is a Java compiler that originated at IBM but since 2007 development has transferred into an open-source project hosted at sourceforge.net. Jikes is written in C++ and strictly adheres to the Java Language Specification [67] moreso than javac - for example extraneous semi-colons after code blocks are valid with javac but not Jikes. The last version available via sourceforge.net is 1.22 which was released October 3, 2004. This release does not support many of the Java 1.5, has no support for Java 1.6.

Java Compiler Kit JKit [**?**] is a compiler designed with the aim of teaching compilation theory and implementation and to aid research into programming languages and compilers. It is designed to easily allow for prototyping of Java extensions or implementation of new languages which compile to Java bytecode.

Eclipse JDT The Eclipse Java Development Tools [**?**] form the basis of the Eclipse Java IDE and includes an incremental Java compiler. The Eclipse JDT adheres to the Java Language Specification [67] more closely than javac [49].

Janino Janino [**?**] is an embedded compiler, which compiles blocks of Java source, rather than a stand-alone compiler. It is intended for runtime compilation of expressions or Java server pages and also can be used for static analysis and code manipulation.

Kopi Java Compiler The Kopi Java Compiler [**?**] is part of the larger open-source Kopi Suite which includes other tools for the generation and editing of Java class files.

JastAdd Extensible Java Compiler The JastAdd Extensible Java Compiler [**?** 50] is a Java compiler built with the JastAdd compiler compiler system [**?**]. As its name suggest it is designed to be easily extensible and is built in a modular fashion. Java 5 features have been implemented as modular extensions to a Java 1.4 compiler base to demonstrate modularity and extensibility that is possible using JastAdd. It compares well with other compilers and runs within a factor of three compared to javac.

The compilers listed here are strictly Java compilers which adhere (or closely adhere) to the Java Language Specification [67] and Java Virtual Machine Specification [108, 16] but there are many compilers which work with a superset or subset of the Java language which we aren't interested in at this stage [**?**]. Not all compilers listed support the latest version of Java (currently 1.6).

C.2 Java Bytecode Assemblers

A Java bytecode assembler takes written bytecode instructions (usually in the form of mnemonics or a simple language) and produces a Java class file. A Java assembler may take care of such things as constant pool generation, use of local variable names and labels. There is no standard ASCII description language for Java bytecode provided by Sun, and different tools use different syntax to describe Java bytecode for the assembler input. Other systems are software libraries which provide APIs for generating and manipulating class files in Java.

ASM ASM [**?**] is an open-source all purpose bytecode manipulation and analysis framework which can be used to generate or manipulate Java class files. It can be used to analyse and transform bytecode programs [98, 15] and is used as a component in many other Java tools [**?**]. It is a Java library and uses Java objects to represent class files and their attributes. In order to generate a class file from scratch a Java application must be written which uses the ASM libraries to create a class file and bytecode attributes to it. Figure C.1, page 148 shows a Java program using the ASM libraries to generate a Hello World Java program.

BCEL Byte Code Engineering Library [**?**] is a library for manipulating and generating class files. BCEL was originally created by Markus Dham [43] and is now an open-source project hosted by The Apache Software Foundation. The latest version is 5.2 which was released in June 2006 and no further work has been done on BCEL since 2006. BCEL is very similar to ASM as they are both libraries for Java which can be used to generate and manipulate bytecode using Java objects.

The class *org.apache.bcel.util.BCELifier* can be used to transform any Java class file into a Java source file which, when compiled and executed, will use the BCEL library to generate the original class file.

Listing C.2, page 149 shows the 'BCELified' Hello World Java source.

Jasmin Jasmin [**?**] is a Java bytecode assembler initially written in 1996 as companion to the book 'Java Virtual Machine' [112] which is now out-of-print. The original authors no longer maintain the program and it is now hosted as an open-source project on sourceforge.net but, as of 2006, is no longer maintained[1]. The Soot framework uses a modified Jasmin assembler [164]. An extension to the Jasmin language known as JasminXT has been defined as part of the tinapoc project [**?**] - a set of reverse engineering tools for Java bytecode in early development stages[2].

Jasmin takes as input a human readable bytecode representation, similar to the output of Sun's javap disassembler, and outputs a Java class file corresponding to the written bytecode instructions.

Constants are written inline and Jasmin takes care of the creation of the class files constant pool.

The standard 'Hello World' program in Java would be represented in Jasmin source as figure C.3, page 150.

[1] New developers were requested on 2008-01-28 via the project forum but the last release was in 2006

[2] the latest version is 0.4-alpha, released Feb 05 2006

Listing C.1: Hello World in ASM source (derived from ASM examples package).

```
import org.objectweb.asm.*;
import org.objectweb.asm.commons.*;

public class ASM {
  public static void main(String[] args) {
    ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_MAXS);
    cw.visit(V1_1,ACC_PUBLIC,              ,null,                    ,null);

    // creates a GeneratorAdapter for the (implicit) constructor
    Method m = Method.getMethod(                  );
    GeneratorAdapter mg = new GeneratorAdapter(ACC_PUBLIC,
            m,
            null,
            null,
            cw);
    mg.loadThis();
    mg.invokeConstructor(Type.getType(Object.class), m);
    mg.returnValue();
    mg.endMethod();

    // creates a GeneratorAdapter for the         method
    m = Method.getMethod(                        );
    mg = new GeneratorAdapter(ACC_PUBLIC + ACC_STATIC, m, null, null, cw);
    mg.getStatic(Type.getType(System.class),
                 ,
            Type.getType(PrintStream.class));
    mg.push(                );
    mg.invokeVirtual(Type.getType(PrintStream.class),
            Method.getMethod(                        ));
    mg.returnValue();
    mg.endMethod();

    cw.visitEnd();

    code = cw.toByteArray();

    FileOutputStream output = new FileOutputStream(                    );
    output.write(code);
    output.close();
  }
}
```

Listing C.2: Hello World in BCEL source.

```
import org.apache.bcel.generic.*;
import org.apache.bcel.classfile.*;
import org.apache.bcel.*;
import java.io.*;

public class Example implements Constants {

  // Instruction factory class contains methods to create
  // bytecode instruction objects.
  private InstructionFactory factory;

  // ConstantPoolGen holds the class' constant pool information
  private ConstantPoolGen    cp;

  // ClassGen represents the class file
  private ClassGen           cg;

  public Example() {
    // generate a new class ( public class HelloWorld extends Object )
    cg = new ClassGen(            ,                    ,
                    , ACC_PUBLIC | ACC_SUPER, new String[] {  });
    // create a constant pool
    cp = cg.getConstantPool();
    // initialise the instruction factory
    factory = new InstructionFactory(cg, cp);
  }

  public void createClassFile(OutputStream out) throws IOException {
    createMethod_Constructor();
    createMethod_Main();
    cg.getJavaClass().dump(out);
  }



  private void createMethod_Constructor() {
    InstructionList il = new InstructionList();

    MethodGen method = new MethodGen(ACC_PUBLIC, Type.VOID,
    Type.NO_ARGS, new String[] {  },
            ,             , il, cp);

    InstructionHandle ih_0 = il.append(factory.createLoad(Type.OBJECT, 0));

    il.append(factory.createInvoke(                    ,          ,
    Type.VOID, Type.NO_ARGS, Constants.INVOKESPECIAL));

    InstructionHandle ih_4 = il.append(factory.createReturn(Type.VOID));

    method.setMaxStack();

    method.setMaxLocals();

    cg.addMethod(method.getMethod());

    il.dispose();
  }

  private void createMethod_Main() {
    InstructionList il = new InstructionList();

    // generate method public static void main(String[] args)
    MethodGen method = new MethodGen(ACC_PUBLIC | ACC_STATIC,
  Type.VOID, new Type[] { new ArrayType(Type.STRING, 1) },
  new String[] {        },       ,             , il, cp);

    // getstatic System.out
    InstructionHandle ih_0 = il.append(
```

Listing C.3: Hello World in Jasmin source. Comments begin with semi-colon (;).

```
;the class modifier and name
.class public HelloWorld
;the super class
.super java/lang/Object


; standard initializer
; a compiler generates a constructor even if one is not defined in Java

; method modifiers, name (<init> is the special name for a constructor),
; parameters (none), and return type (V is the bytecode symbol for void).
treated
.method public <init>()V
   aload_0   ; push
   invokenonvirtual java/lang/Object/<init>()V ; invoke super constructor.
   return
.end methodfigure}

; main method

; method modifiers (public + static), name (main),
; parameters (String[] args) and return type (void)

.method public static main([Ljava/lang/String;)V
  .limit stack 2 ; limit methods stack height to 2
  .limit locals 1 ; limit number of local variables to 1 (args)

  ; push System.out onto stack
  getstatic java/lang/System/out Ljava/io/PrintStream;
  ; push the constant                onto stack
  ldc
  ; invoke System.out.println(              )
  invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

  return
.end method
```

---

serp TODO http://serp.sourceforge.net/ still active?

Cojen Cojen [**?**] is a bytecode generation library with a primary goal of making raw Java classfile generation easy. The library contains classes which equate to bytecode instructions. Cojen is a fork of the no longer maintained Tea Trove bytecode library [**?**].

Cojen is another Java library, very similar to ASM and BCEL, but seems simpler - see the Hello World example in listing C.4, page 151.

Jamaica Jamaica, The JVM Macro Assembler, is an easy to learn assembly language for JVM programming [**?**]. It takes as input a hybrid Java and Java bytecode source file as input and outputs a class file. Class, interface and method signatures are written in standard Java source while the method body is written in a bytecode format similar to Jasmin. Jamaica bytecode instruction format is slighlty easier to understand than Jasmin.

For example Jamaica uses the Java standard fullstop (.) to for package hierarchies and resolve package names (e.g. it is only neccesary to write PrintStream and not java.io.PrintStream). Variables, fields and labels all use names rather than virtual machine indices unlike Jasmin.

Jamaica also supports a number of macros to make bytecode programming easier, which automatically generate common bytecode instructions. For example, listing C.6, page 152 shows how the

Listing C.4: Hello World in Cojen source.

```
public class CojenTest {

  public static void main(String[] args) throws Exception {
    ClassFile classFile = new ClassFile(              );
    FileOutputStream outputFile = new FileOutputStream(                    );

    classFile.addDefaultConstructor();

    TypeDesc[] params = new TypeDesc[] {TypeDesc.STRING.toArrayType()};
    MethodInfo mainMethod = classFile.addMethod(Modifiers.PUBLIC_STATIC,
                    , null, params);
    CodeBuilder b = new CodeBuilder(mainMethod);

    TypeDesc printStream = TypeDesc.forClass(                      );

    b.loadStaticField(                    ,       , printStream);
    b.loadConstant(              );

    params = new TypeDesc[] {TypeDesc.STRING};
    b.invokeVirtual(printStream,             , null, params);

    b.returnVoid();

    classFile.writeTo(outputFile);
    outputFile.close();
  }

}
```

three lines of bytecode in listing C.5 can be condensed into one line using the *%println* macro. Listing C.5, page 152 shows the Hello World program using Jamaica.

Listing C.5: Hello World in Jamaica source.

```
public class HelloJamaica {

  public static void main(String[] args) {
    getstatic System.out PrintStream
    ldc
        invokevirtual PrintStream.println(String)void
  }

}
```

Listing C.6: Hello World in Jamaica source using a macro.

```
public class HelloJamaica {

  public static void main(String[] args) {
    %println
  }

}
```

The most recently active in development bytecode assembly system is ASM, while Jasmin and BCEL are widely used and mature systems. ASM, BCEL and Cojen are all Java libraries for the generation and manipulation of bytecode via the use of Java objects whereas Jasmin compiles human readable bytecode instructions into classfiles.

The use of Java objects for the manipulation of bytecode files will be more familar to Java programmers but still needs a thorough understanding of the bytecode language.

Althought out-of-print 'Java Virtual Machine' [112] is an excellent reference for the Jasmin software. Jamaica code is more readable than Jasmin code but not as similar to actual bytecode due to its use of macros which can make the code easier to read. Tinapoc [**?**] includes a new version 2.0 of Jasmin and an extension of the language known as JasminXT.

C.2.1 Other language → Java Bytecode Compilers

Althought the JVM was initially designed with only Java in mind there are many other languages aimed at the Java platform. Unlike Common Intermediate Language virtual machines the Java Virtual Machine was designed from the beginning to execute only Java bytecode generated from Java source. Most of the language compilers for other languages include libraries which help to execute features of the specific language on the JVM.

A few languages are listed here for a comprehensive, up-to-date list see [**?**].

Groovy Groovy is 'an agile dynamic language for the Java Platform'. It is a scripting language which builds on the strengths of Java and the JVM aimed at Java developers and others familar with scripting languages. Groovy includes an interperater and a bytecode compiler. It produces bytecode which uses a Java library for Groovy functions.

A simple Hello World program consisting of the line

```
println "Hello World"
```

produces a 700 line bytecode program[3] and requires the inclusion of Groovy's library classes.

Groovy is in mature and still in development and it is also going standardization process via a Java Community Process under JSR241[4].

Rhino Rhino is an open-source implementation of JavaScript written in Java which also includes a compiler to Java bytecode. The compiler produces bytecode which requires the use of the Rhino runtime library.

Scala The Scala programming language integrates the features of object orientated programming with those of functional programming. There exists an interperater and compiler written in Java and a compiler which compiles Scala code to bytecode to be executed on a JVM.

JGNAT JGNAT is an open-source Ada compiler which compiles Ada source into Java bytecode.

Jython Jython is a Java implementation of the Python programming langauge. Version 2.2.1 and below contain a Python to Java bytecode compiler but this is no longer maintained and is dropped from the latest beta[5] release.

[3] The output of javap -verbose HelloWorld was 700 lines
[4] `http://www.jcp.org/en/jsr/detail?id=241`
[5] 2.5b0, 31/10/2008

C.2.2 Bytecode Optimisers

A bytecode optimiser takes as input a Java class file, performs optimising transformations and outputs a semantically equivalent Java class file. For example an optimiser might perform peephole optimisation on the bytecode by replacing a set of instructions with a smaller set of equivalent instructions.

As an example the following bytecode

```
iconst_1
iconst_2
iadd
```

could be replaced by

```
iconst_3
```

This is known as constant folding.

Soot Soot is a Java optimisation framework created by the Sable Research Group at McGill University. It can transform class files into one of four intermediate representations of Java (Baf, Jimple, Shimple, Grimp) which are suitable for transformation and analysis operations. Baf is closest to bytecode, whereas Grimp is closest to Java source code.

ProGuard ProGuard [**?**] is an open-source Java class file shrinker, optimizer, obfuscator, and preverifier.

JODE JODE [**?**] is an open-source decompiler and optimiser for Java bytecode. It is able to decompiler Java 1.3 source and also contains an optimiser which can perform optimising transformations on class files.

Zelix Classmaster Zelix Classmaster [**?**] is a commerical obfuscator and optimiser sold by Zelix Pty Ltd.

C.2.3 Bytecode Obfuscators

A bytecode obfuscator takes a Java class file as input, performs some obfuscating transformations and outputs a semantically equivalent class file.

An simple obfuscation might be to randomise constants in a classfile constant pool. Performing name randomisation is easier on a class file than a Java file because all the constants as stored in the constant pool and are referenced by their index throughout the bytecode.

Consider the following class Randomise and the extract from it's constant pool (Listing C.7), which contains a private method named *sayHello*. The name of this method gives away the purpose of the method and would be useful to an attacker in understanding the source code after decompiling.

Listing C.7: Randomise Class.

```
public class Randomise {

  public static void main(String[] args) {
    sayHello();
  }

  private static void sayHello() {
    System.out.println(             );
  }

}

1: CONSTANT_Methodref - class_index: 7  name_and_type_index: 17
2: CONSTANT_Methodref - class_index: 6  name_and_type_index: 18
3: CONSTANT_Fieldref -  class_index: 19 name_and_type_index: 20
4: CONSTANT_String: Hello World
.................
14: CONSTANT_Utf8: sayHello
15: CONSTANT_Utf8: SourceFile
16: CONSTANT_Utf8: Randomise.java
17: CONSTANT_NameAndType - name_index: 8  descriptor_index: 9
18: CONSTANT_NameAndType - name_index: 14 descriptor_index: 9
.................
31: CONSTANT_Utf8: (Ljava/lang/String;)V
```

The name *sayHello* is stored in the constant pool at index 14 and every place that name is mentioned in the Java source is replace by a reference to the constant pool. We can change the string at constant pool index 14 to some random name to confuse attackers of the class file.

ProGuard ProGuard [**?**] is an open-source Java class file shrinker, optimizer, obfuscator, and preverifier.

Zelix Classmaster Zelix Classmaster [**?**] is a commerical obfuscator and optimiser sold by Zelix Pty Ltd.

Appendix D

Java Class File Format

Class file primitives types: TODO

A Java compiler compiles Java source code into intermediate Java Byte Code in files ending with the extension *.class*. Listing D.2, page 158 shows a hex dump of the Java class file for the hello world program (Listing D.1, page 158).

Java class files are divided into 10 areas:

Magic Number 0xCAFEBABE

Version Numbers The minor and major versions of the class file

Constant Pool Pool of constants for the class

Access Flags e.g. abstract, static, etc

This Class The name of the current class

Super Class The name of the super class

Interfaces Any interfaces in the class

Fields Any fields in the class

Methods Any methods in the class

Attributes Any attributes of the class

Magic Number

The first four bytes of a class file is the magic number 0xCAFEBABE (3405691582 in decimal) which identifies the file as a Java class file to the Java Virtual Machine. The choice of magic number was taken by James Gosling who used a similar hex number 0xCAFEDEAD to describe a local cafe they frequented and which was used as the magic number for a an object file format. When a new magic number was needed for the Java class file a search for words other than DEAD resulted in the use of BABE [**?**].

Version Numbers

The next four bytes are the major and minor version numbers of the compiler used. The minor version of listing D.2, page 158 is 0x0000 and the major version is 0x0031 (decimal 49). Versions of Java Virtual Machines are J2SE 6.0=50, J2SE 5.0=49, JDK 1.4=48, JDK 1.3=47, JDK 1.2=46, JDK 1.1=45.

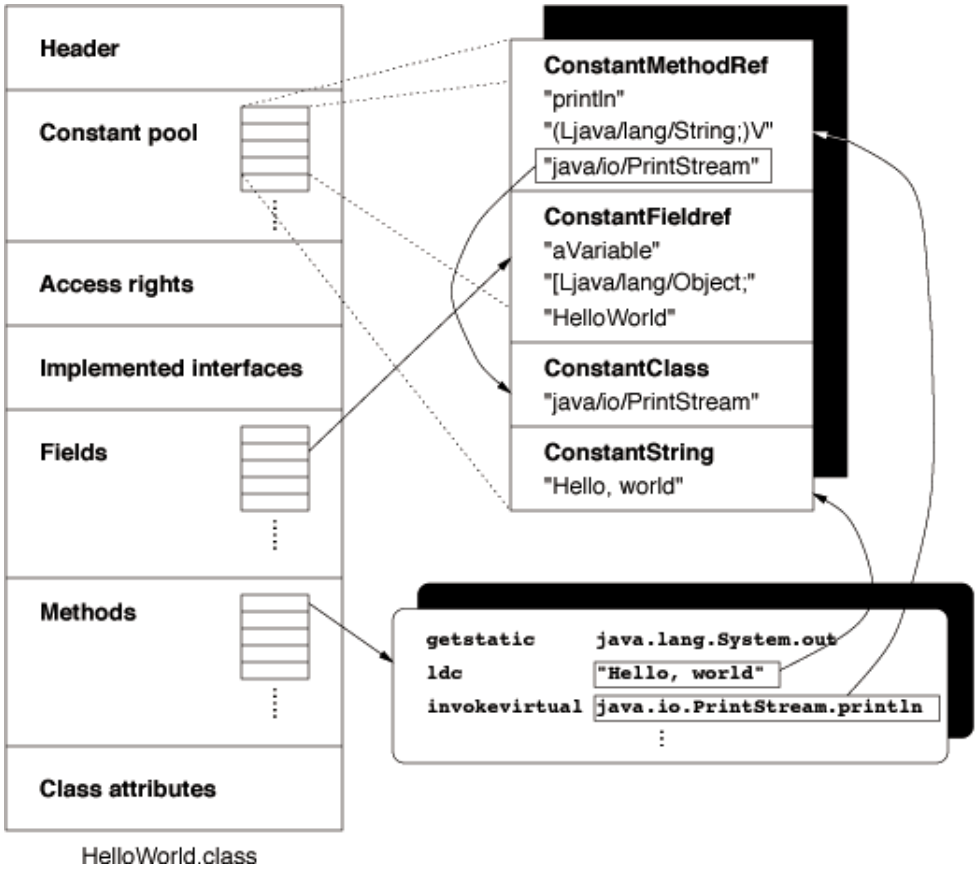


Figure D.1: Conceptual diagram of a Java class file. Source: [**?**]

Constant Pool

The next two bytes of a class file are the constant pool count. The constant pool is an array of variable length elements containing every constant and variable name used in the Java class. Constants are referenced by their constant pool index through the bytecode. Each constant in the pool is preceded by a tag denoting it's type (TODO: INSERT TABLE OF TYPES). The count of the constant pool is one greater than the actual number of entries as the constant pool count is included itself in the count. A lot of the tags in the constant pool are symbolic references to other members of the constant pool. Symbolic references are resolved at runtime.

The constant pool count bytes are 0x00, 0x1D (decimal 29) in listing D.2, page 158 meaning that there are 28 constants and/or variables used in the program.

Access Flags

The first two bytes after the constant pool are access flags which show whether the file is a class or interface and whether it is final, abstract, and/or public. Access flags are or'd together to generate a modifier containing all the access flags.

TODO: INSERT TABLE FROM PAGE 35, decompiling Java.

This Class

This class contains an reference to an index in the constant pool containing the name of the class defined in the file.

Listing D.1: Hello World Java Program

```
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println(             );
  }
}
```

Listing D.2: Hex Dump of Hello World Java Program

```
CAFEBABE00000031001D0A0006000F09001000110800120A
00130014070015070016010006 3C696E69743E0100032829
56010004436F646501000F4C696E654E756D626572546162
6C650100046D61696E010016285B4C6A6176612F6C616E67
2F537472696E673B295601000A536F7572636546696C6501
000F48656C6C6F576F726C642E6A6176610C000700080700
170C0018001901000B48656C6C6F20576F726C6407001A0C
001B001C01000A48656C6C6F576F726C640100106A617661
2F6C616E672F4F626A6563740100106A6176612F6C616E67
2F53797374656D0100036F75740100154C6A6176612F696F
2F5072696E7453747265616D3B0100136A6176612F696F2F
5072696E7453747265616D010007072696E746C6E010015
284C6A6176612F6C616E672F537472696E673B2956002100
0500060000000000020001000700080001000900000001D00
010001000000052AB70001B100000001000A000000060001
000000010009000B000C000100090000002500020001 0000
0009B200021203B60004B100000001000A0000000A000200
0000030008000400010000D00000002000E
```

Super Class

This class contains an reference to an index in the constant pool containing the name of the parent of the class defined in the file.

Interfaces

The next two bytes of a class file are the interfaces count. The interfaces then follow as references to indexes in the constant pool.

Fields

The next two bytes of a class file are the field count followed by the fields declared in the class file. Each field is composed of access flags, name type, description and attributes. The access flags are or'd together to produce a one byte modifier while the type, description and attributes are references to indexes in the constant pool.

Methods

The next two bytes of a class file are the method count followed by the methods declared in the class file. Each method is composed of access flags, name type, description and attributes.

Attributes

The next two bytes of a class file are the attribute count followed by attributes of the class file which include the name of the source file, information about inner classes. Other attributes can be stored here too.

Listing D.3: Mnemonic Byte Code Dump of Hello World Java Program

```
Class: HelloWorld
Superclass: java/lang/Object
Source File: HelloWorld.java
Access Flags: {public super synchronized }
cf->major_version: 49
cf->constant_pool_count: 29
cf->methods_count: 2
cf->attributes_count: 1
Constant Pool:
  1: CONSTANT_Methodref - class_index: 6  name_and_type_index: 15
  2: CONSTANT_Fieldref -  class_index: 16 name_and_type_index: 17
  3: CONSTANT_String: Hello World
  4: CONSTANT_Methodref - class_index: 19 name_and_type_index: 20
  5: CONSTANT_Class: Index 21, Name HelloWorld
  6: CONSTANT_Class: Index 22, Name java/lang/Object
  7: CONSTANT_Utf8: <init>
  8: CONSTANT_Utf8: ()V
  9: CONSTANT_Utf8: Code
  10: CONSTANT_Utf8: LineNumberTable
  11: CONSTANT_Utf8: main
  12: CONSTANT_Utf8: ([Ljava/lang/String;)V
  13: CONSTANT_Utf8: SourceFile
  14: CONSTANT_Utf8: HelloWorld.java
  15: CONSTANT_NameAndType - name_index: 7  descriptor_index: 8
  16: CONSTANT_Class: Index 23, Name java/lang/System
  17: CONSTANT_NameAndType - name_index: 24 descriptor_index: 25
  18: CONSTANT_Utf8: Hello World
  19: CONSTANT_Class: Index 26, Name java/io/PrintStream
  20: CONSTANT_NameAndType - name_index: 27 descriptor_index: 28
  21: CONSTANT_Utf8: HelloWorld
  22: CONSTANT_Utf8: java/lang/Object
  23: CONSTANT_Utf8: java/lang/System
  24: CONSTANT_Utf8: out
  25: CONSTANT_Utf8: Ljava/io/PrintStream;
  26: CONSTANT_Utf8: java/io/PrintStream
  27: CONSTANT_Utf8: println
  28: CONSTANT_Utf8: (Ljava/lang/String;)V

public super synchronized class HelloWorld extends java/lang/Object

Method public  <init> () -> void

0 aload_0
1 invokenonvirtual #1 <Method java/lang/Object.<init> ()V>
4 return

Method public static  main (java/lang/String []) -> void

0 getstatic #2 <Field java/lang/System.out Ljava/io/PrintStream;>
3 ldc #3 <String               >
5 invokevirtual #4 <Method java/io/PrintStream.println (Ljava/lang/String;)V>
8 return
```

Listing D.4: Simple use of Java if statement

```
public class if_simple {

  public static void main(String[] args) {
    if(args[0] ==       ) {
      System.out.println(       );
    }else{
      System.out.println(         );
    }
  }

}
```

Listing D.5: Mnemonic Byte Code Dump of Java Simple If program

```
public class if_simple extends java.lang.Object{
public if_simple();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>
```

Listing D.6: Simple use of Java loop statements

```
public class loop_simple {

  public static void main(String[] args) {
    for(int i = 0; i < 100; i++) {
      System.out.println(i);
    }

    int i = 0;
    while(i < 100) {
      System.out.println(i);
      i++;
    }
  }

}
```

Listing D.7: Mnemonic Byte Code Dump of Simple use of Java loop statements

```
public class loop_simple extends java.lang.Object{
public loop_simple();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>
```

Listing D.8: Mnemonic Byte Code Dump of c2j Java translation of ?? - short C program with *goto* statements

```
public class test extends java.lang.Object{
public test();
  Code:
   0: aload_0
   1: invokespecial #1; //Method java/lang/Object."<init>
```

<init>

Appendix E

Bytecode Instruction Set

```
00 (0x00) nop
01 (0x01) aconst_null

02 (0x02) iconst_m1
03 (0x03) iconst_0
04 (0x04) iconst_1
05 (0x05) iconst_2
06 (0x06) iconst_3
07 (0x07) iconst_4
08 (0x08) iconst_5

09 (0x09) lconst_0
10 (0x0a) lconst_1

11 (0x0b) fconst_0
12 (0x0c) fconst_1
13 (0x0d) fconst_2

14 (0x0e) dconst_0
15 (0x0f) dconst_1

16 (0x10) bipush

17 (0x11) sipush

18 (0x12) ldc
19 (0x13) ldc_w
20 (0x14) ldc2_w

21 (0x15) iload
22 (0x16) lload

23 (0x17) fload

24 (0x18) dload

25 (0x19) aload

26 (0x1a) iload_0
27 (0x1b) iload_1
28 (0x1c) iload_2
29 (0x1d) iload_3

30 (0x1e) lload_0
31 (0x1f) lload_1
32 (0x20) lload_2
33 (0x21) lload_3

34 (0x22) fload_0
35 (0x23) fload_1
36 (0x24) fload_2
37 (0x25) fload_3

38 (0x26) dload_0
39 (0x27) dload_1
40 (0x28) dload_2
41 (0x29) dload_3

42 (0x2a) aload_0
43 (0x2b) aload_1
44 (0x2c) aload_2
45 (0x2d) aload_3

46 (0x2e) iaload

47 (0x2f) laload

48 (0x30) faload

49 (0x31) daload

50 (0x32) aaload

51 (0x33) baload

52 (0x34) caload

53 (0x35) saload

54 (0x36) istore

55 (0x37) lstore

56 (0x38) fstore

57 (0x39) dstore

58 (0x3a) astore

59 (0x3b) istore_0
60 (0x3c) istore_1
61 (0x3d) istore_2
62 (0x3e) istore_3

63 (0x3f) lstore_0
64 (0x40) lstore_1
65 (0x41) lstore_2
66 (0x42) lstore_3

67 (0x43) fstore_0
68 (0x44) fstore_1
69 (0x45) fstore_2
70 (0x46) fstore_3

71 (0x47) dstore_0
72 (0x48) dstore_1
73 (0x49) dstore_2
74 (0x4a) dstore_3

75 (0x4b) astore_0
76 (0x4c) astore_1
77 (0x4d) astore_2
78 (0x4e) astore_3

79 (0x4f) iastore

80 (0x50) lastore

81 (0x51) fastore

82 (0x52) dastore

83 (0x53) aastore

84 (0x54) bastore

85 (0x55) castore

86 (0x56) sastore

87 (0x57) pop
88 (0x58) pop2

089 (0x59) dup
090 (0x5a) dup_x1
091 (0x5b) dup_x2

092 (0x5c) dup2
093 (0x5d) dup2_x1
094 (0x5e) dup2_x2

095 (0x5f) swap

096 (0x60) iadd

097 (0x61) ladd

098 (0x62) fadd

099 (0x63) dadd
```

```
100 (0x64) isub
101 (0x65) lsub
102 (0x66) fsub
103 (0x67) dsub
104 (0x68) imul
105 (0x69) lmul
106 (0x6a) fmul
107 (0x6b) dmul
108 (0x6c) idiv
109 (0x6d) ldiv
110 (0x6e) fdiv
111 (0x6f) ddiv
112 (0x70) irem
113 (0x71) lrem
114 (0x72) frem
115 (0x73) drem
116 (0x74) ineg
117 (0x75) lneg
118 (0x76) fneg
119 (0x77) dneg
120 (0x78) ishl
121 (0x79) lshl
122 (0x7a) ishr
123 (0x7b) lshr
124 (0x7c) iushr
125 (0x7d) lushr
126 (0x7e) iand
127 (0x7f) land
128 (0x80) ior
129 (0x81) lor
130 (0x82) ixor
131 (0x83) lxor
132 (0x84) iinc
133 (0x85) i2l
134 (0x86) i2f
135 (0x87) i2d
136 (0x88) l2i
137 (0x89) l2f
138 (0x8a) l2d
139 (0x8b) f2i
140 (0x8c) f2l
141 (0x8d) f2d
142 (0x8e) d2i
143 (0x8f) d2l
144 (0x90) d2f
145 (0x91) i2b
146 (0x92) i2c
147 (0x93) i2s
148 (0x94) lcmp
149 (0x95) fcmpl
150 (0x96) fcmpg
151 (0x97) dcmpl
152 (0x98) dcmpg
153 (0x99) ifeq
154 (0x9a) ifne
155 (0x9b) iflt
156 (0x9c) ifge
157 (0x9d) ifgt
158 (0x9e) ifle
159 (0x9f) if_icmpeq
160 (0xa0) if_icmpne
161 (0xa1) if_icmplt
162 (0xa2) if_icmpge
163 (0xa3) if_icmpgt
164 (0xa4) if_icmple
165 (0xa5) if_acmpeq
166 (0xa6) if_acmpne
167 (0xa7) goto
168 (0xa8) jsr
169 (0xa9) ret
170 (0xaa) tableswitch
171 (0xab) lookupswitch
172 (0xac) ireturn
173 (0xad) lreturn
174 (0xae) freturn
175 (0xaf) dreturn
176 (0xb0) areturn
177 (0xb1) return
178 (0xb2) getstatic
179 (0xb3) putstatic
180 (0xb4) getfield
181 (0xb5) putfield
182 (0xb6) invokevirtual
183 (0xb7) invokespecial
184 (0xb8) invokestatic
185 (0xb9) invokeinterface
186 (0xba) unused
187 (0xbb) new
188 (0xbc) newarray
189 (0xbd) anewarray
190 (0xbe) arraylength
191 (0xbf) athrow
192 (0xc0) checkcast
193 (0xc1) instanceof
194 (0xc2) monitorenter
195 (0xc3) monitorexit
196 (0xc4) wide
197 (0xc5) multianewarray
198 (0xc6) ifnull
199 (0xc7) ifnonnull
200 (0xc8) goto_w
201 (0xc9) jsr_w
Reserved opcodes:
202 (0xca) breakpoint
254 (0xfe) impdep1
255 (0xff) impdep2
```

Appendix F

Stuff

F.1 Static Single Assignment Form

Static Single Assignment [6, 150, 41] is a program form where each variable is assigned exactly once. SSA essentially maps each user defined variable x into a set of variables $x_0, x_1, x_2, \ldots$ for each assignment to x.

Consider the following psuedo-code where the variable x has 3 values assigned to it

$$
\begin{aligned}
x &= 5 \\
x &= x \times 6 \\
x &= x + 1
\end{aligned}
$$

Converting this into SSA form results in the creation of new instances of variable x for each assignment operation

$$
\begin{aligned}
x_0 &= 5 \\
x_1 &= x_0 \times 6 \\
x_2 &= x_1 + 1
\end{aligned}
$$

The variable instances on path merges must ensure that they preserve the same data-flow by assigning the correct instance of the common variable using a function usually denoted with the ϕ symbol. The function $\phi(x, y)$ takes the value of x if control derived from the left arc or y if control derived from the right. In the example below the value of y is dependent on *condition* because the common variable x takes its value from either x_1 or x_2 depending on whether *condition* is true or false.

$$x_0 = 5$$

`if(`*condition*`)` {
$x_1 = x_0 \times 6$
`}else{`
$x_2 = x_1 + 1$
}

$$y = x_? + 3$$

Thus, in the following, we introduce a new instance x_3 which takes its value from either x_1 or x_2 determined by the ϕ function and dependent on *condition*.

$$x_0 = 5$$

`if(`*condition*`)` {
$x_1 = x_0 \times 6$
`}else{`

$$x_2 = x_1 + 1$$

}

$$x_3 = \phi(x_1, x_2)$$
$$y = x_3 + 3$$

The bytecode from listing 2.9, page 33 could be imagined in psuedo-bytecode as

```
0: iconst_0        //push 0 onto stack
1: istore_0_0      //pop integer from stack, store in local 0_0
2: ldc "hello"     //push String constant onto stack
3: astore_0_1      //pop object reference from stack, store in local 0_1
4: return
```

where each 'assignment' operation (**astore** and **istore** instructions) to local variable 0 creates a new instance of local variable 0.

An SSA transformation would create an intermediate language between bytecode and Java and is a good first step to decompilation of Java bytecode.

F.2 Stack-based instructions

Java bytecode is a stack-based language and flattening the stack-based instructions is one minor problem that machine code decompilers do not have [54]. This problem is solved by introducing variables to represent the stack positions.

F.2.1 Stack Height

Every opcode in the Java bytecode instruction set performs known operations on the stack. It is therefore possible to calculate the maximum stack height that a method uses. For example the instruction *iconst_2* pushes one integer onto the stack. After *iconst_2* is executed the stack therefore contains exactly one more item than before *iconst_2* was executed.

The bytecode below shows a list of instructions with the the type of stack operation and the size of the stack after each instruction is executed, showing a maximum stack height of two.

```
0: iconst_2   push           [*]
1: istore_0   pop            []
2: iconst_2   push           [*]
3: istore_1   pop            []
4: iload_0    push           [*]
5: iload_1    push           [**]
6: iadd       pop, pop, push [*]
7: istore_2   pop            []
8: return
```

F.2.2 Local Variables

The number of local variables can be calculated for a method by adding the number of method arguments to the number of extra local variables used in *load* and *store* operations within the method.

The following is an instance method with 2 paramaters

```
public void foobar(int foo, int bar):
 0: iconst_0
 1: istore_3
 2: iconst_3
 3: istore 4
 5: iload_1
 6: istore_2
 7: return
```

this means that the first three local variables are used by $this, foo$ and bar. There are two more more local variable slots used within the method (3 and 4) bringing the total number of local variables to 5.

F.2.3 Flattening the Stack

The follow static method foo has a maximum stack height of 2 and uses 3 local variables. It simply declares two variables and stores their sum in a third variable.

```
public static void foobar():
 0: iconst_2
 1: istore_0
 2: iconst_2
 3: istore_1
 4: iload_0
 5: iload_1
 6: iadd
 7: istore_2
 8: return
```

We declare two variables to represent the two possible positions on the stack (s_0 and s_1) and translate *store* and *load* operations into assignments by introducing 3 variables representing the local variable slots (v_0, v_1, v_2).

```
public static void foo():
 s0 = 2
 v0 = s0
 s0 = 2
 v1 = s0
 s0 = v0
 s1 = v1
 s0 = s0 + s1
 v2 = s0
```

F.3 JLS Inconsistancies

Due to inconsistencies in the Java Language Specification and the Java Virtual Machine Specification there are irregularities between some Java source files and their bytecode counterparts. The *try-finally* construct is a source of such problems. Listing F.1, page 169 shows Java source which, when compiled with Jikes to Java 1.4 compatible bytecode, results in being rejected by the JVM [160]. If compiled with javac 1.6 the code will execute though some entries in its StackMapTable are marked bogus (TODO: what does this mean?).

Both jad and Dava are unable to decompile the bytecode for listing F.1, page 169 (Figure F.2, 171).

Listing F.1: Legal Java source which produces illegal bytecode [160]

```
static int test(boolean b) {
  int i;

  L: {
    try {
      if (b) return 1;
      i = 2;
      if (b) break L;
    } finally {
      if (b) i = 3;
    }
      i = 4;
     }

  return i;
}
```

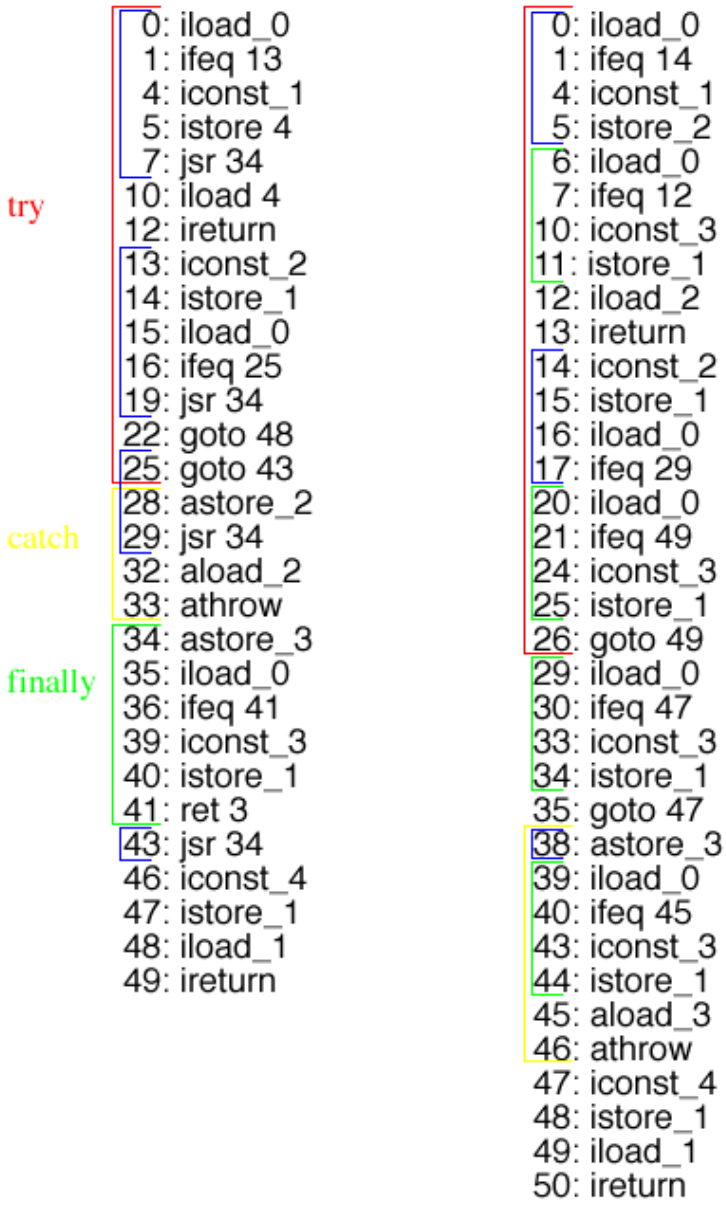


Figure F.1: Bytecode resulting from compilation of listing F.1, page 169. Left-hand-side compiled with jikes (Java 1.4), right-hand-side compiled with javac 1.6. Blue sections indicate exception table entries.

Listing F.2: Jad decompiler output for listing F.1, page 169 (compiled with javac 1.6)

```
static int test(boolean flag)
    {
        int i;
        if(!flag)
            break MISSING_BLOCK_LABEL_14;
        i = 1;
        byte byte0;
        if(flag)
            byte0 = 3;
        return i;
        byte byte1 = 2;
        if(flag)
        {
            if(flag)
                byte1 = 3;
            break MISSING_BLOCK_LABEL_49;
        }
        if(flag)
            byte1 = 3;
        break MISSING_BLOCK_LABEL_47;
        Exception exception;
        exception;
        if(flag)
            byte1 = 3;
        throw exception;
        byte1 = 4;
        return byte1;
    }
```

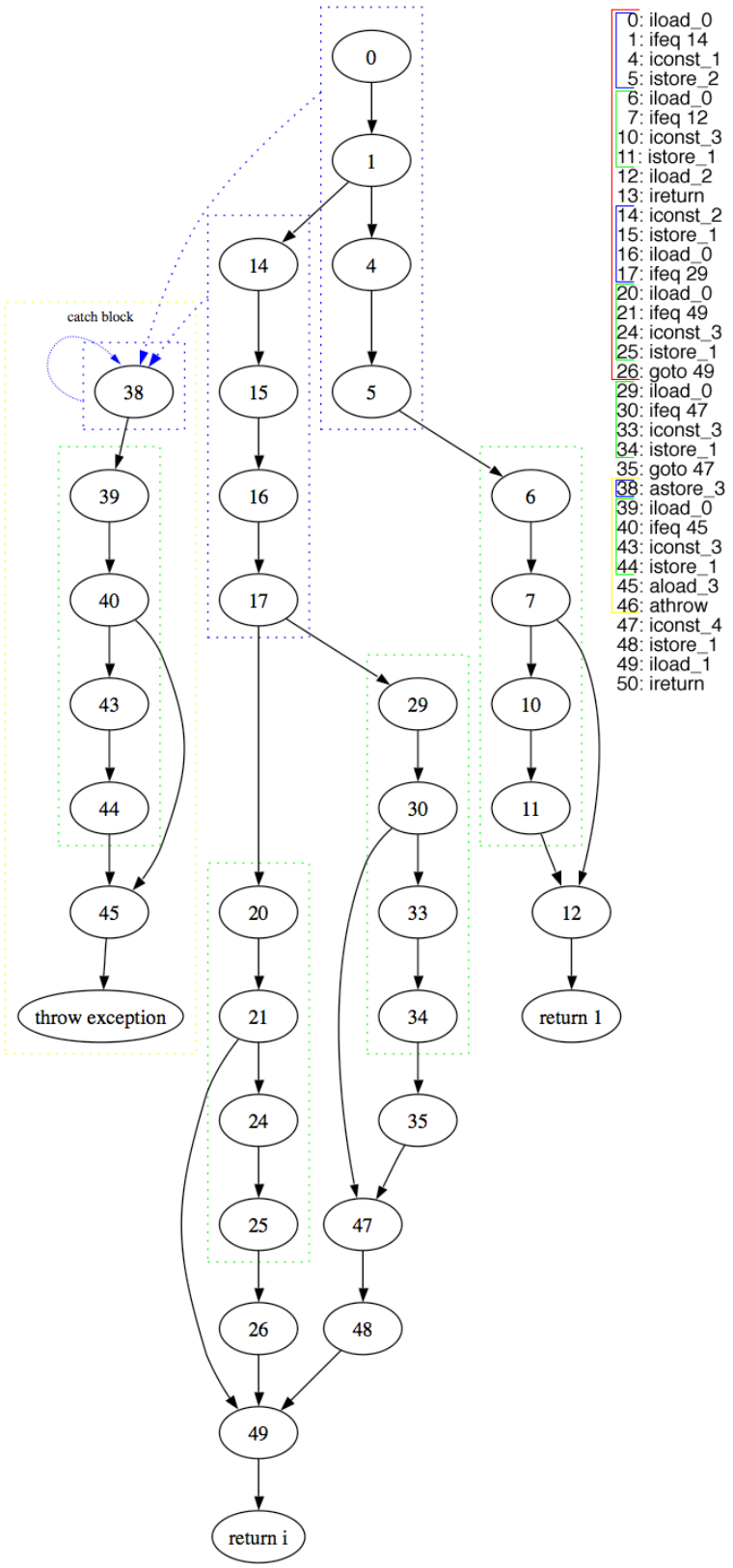


Figure F.2: Control flow graph for javac 1.6 generated bytecode of listing F.1, page 169.

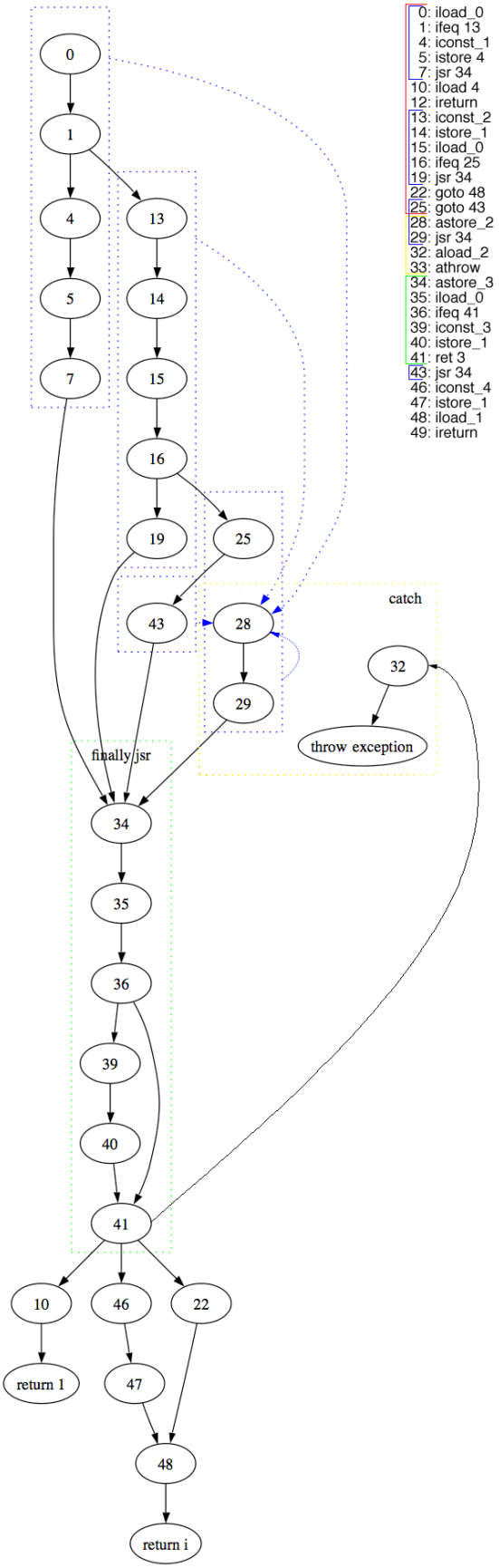


Figure F.3: Control flow graph for jikes (Java 1.4) generated bytecode of listing F.1, page 169.

Bibliography

[1] Mocha, the java decompiler, 1996. URL `http://www.brouhaha.com/~eric/computers/mocha.html`.

[2] SourceTec (Jasmine), 1997. URL `http://www.sothink.com/product/javadecompiler/index.htm`.

[3] JGNAT, 2009. URL `http://code.google.com/p/jgnat/`.

[4] Alfred V Aho, Monica S Lam, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, August 2006. ISBN 0321486811.

[5] Jieqing Ai, Xingming Sun, Yunhao Liu, Ingemar J. Cox, Guang Sun, and Yi Luo. A stern-based Collusion-Secure software watermarking algorithm and its implementation. In *Proceedings of the 2007 International Conference on Multimedia and Ubiquitous Engineering*, pages 813–818. IEEE Computer Society, 2007. ISBN 0-7695-2777-9.

[6] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 111, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73561.

[7] Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Covert communication through executables. In *Program Acceleration through Application and Architecture Driven Code Transformations: Symposium Proceedings*, pages 83–85, 2004.

[8] Bertrand Anckaert, Bjorn De Sutter, and Koen De Bosschere. Steganography for executables. *Information Security and Cryptology - ICISC 2004*, pages 425–439, 2005.

[9] Genevieve Arboit. A method for watermarking java programs via opaque predicates. In *The Fifth International Conference on Electronic Commerce Research (ICECR-5)*, 2002.

[10] Jrg Arndt. *Matters Computational: ideas, algorithms, source code*. Springer, 1st edition edition, October 2010. ISBN 978-3642147630. URL `http://www.jjj.de/fxt/#fxtbook`. Free electronic version online.

[11] Sanjeev Arora, Satish Rao, and Umesh Vazirani. Expander flows, geometric embeddings and graph partitioning. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 222–231, Chicago, IL, USA, 2004. ACM. ISBN 1-58113-852-0.

[12] Michael Batchelder and Laurie Hendren. Obfuscating java: the most pain for the least gain . Braga, Portugal, March 2007.

[13] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. Efficient local type inference. *SIGPLAN Not.*, 43(10):475492, 2008. ISSN 0362-1340. doi: 10.1145/1449955.1449802.

[14] Swaroop Belur and Kartik Bettadapura. Jdec: Java decompiler, 2008. URL `http://jdec.sourceforge.net/`. 2008.

[15] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a code manipulation tool to implement adaptable systems. Technical report, 2002.

[16] Alex Buckley, Eva Rose, Alessandro Coglio, Borland Software Corporation, Inc. Sun Microsystems, Inc. Tmax Soft, SavaJe Technologies, and Esmertec AG. JSR 202: JavaTM class file specification update, 2006. URL `http://jcp.org/en/jsr/detail?id=202`.

[17] Business Software Alliance. Sixth annual BSA and IDC global software piracy study. Technical Report 6, Business Software Alliance, 2008.

[18] Eugène Catalan. Note extraite d'une lettre adressée à l'éditeur. *Journal för die reine und angewandte Mathematik*, 27:192, 1844.

[19] W.L. Caudle. On inverse of compiling. *Sperry-UNIVAC*, April 1980. URL `http://www.program-transformation.org/view/Transform/OnInverseOfCompiling?skin=print.pattern`.

[20] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981. ISSN 0096-0551. doi: DOI:10.1016/0096-0551(81)90048-5. URL `http://www.sciencedirect.com/science/article/B6TYK-48V1WXD-1S/2/4d1760365aca10db3f7fc10d8a263ea8`.

[21] XiaoJiang Chen, DingYi Fang, JingBo Shen, Feng Chen, WenBo Wang, and Lu He. A dynamic graph watermark scheme of tamper resistance. In *Proceedings of the 2009 Fifth International Conference on Information Assurance and Security - Volume 01*, pages 3–6. IEEE Computer Society, 2009. ISBN 978-0-7695-3744-3.

[22] Maria Chroni and Stavros D. Nikolopoulos. Encoding watermark integers as self-inverting permutations. In *Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies*, pages 125–130, Sofia, Bulgaria, 2010. ACM. ISBN 978-1-4503-0243-2.

[23] C. Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994.

[24] C. Collberg, A. Huntwork, E. Carter, and G. Townsend. Graph theoretic software watermarks: Implementation, analysis, and attacks. In *Workshop on Information Hiding*, 2004.

[25] Christian Collberg. Sandmark algorithms. Technical report, University of Arizona, Department of Computer Science, July 2002.

[26] Christian Collberg. Sandmark, August 2004. URL `http://www.cs.arizona.edu/sandmark/`.

[27] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional, 2009. ISBN 0321549252, 9780321549259.

[28] Christian Collberg and Tapas Ranjan Sahoo. Software watermarking in the frequency domain: implementation, analysis, and attacks. *J. Comput. Secur.*, 13(5):721–755, 2005.

[29] Christian Collberg and Clark Thomborson. On the limits of software watermarking. Technical Report 164, August 1998.

[30] Christian Collberg and Clark Thomborson. Software watermarking: Models and dynamic embeddings. In *Principles of Programming Languages 1999, POPL'99*, January 1999.

[31] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, July 1997.

[32] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Principles of Programming Languages 1998, POPL'98*, January 1998.

[33] Christian Collberg, Stephen Kobourov, Edward Carter, and Clark Thomborson. Error-Correcting graphs for software watermarking. In *Proceedings of the 29th Workshop on Graph Theoretic Concepts in Computer Science*, pages 156–167, 2003.

[34] Christian Collberg, Clark Thomborson, and Gregg Townsend. Dynamic Graph-Based software watermarking. Technical report, Dept. of Computer Science, Univ. of Arizona, 2004.

[35] Christian Collberg, Andrew Huntwork, Edward Carter, Gregg Townsend, and Michael Stepp. More on graph theoretic software watermarks: Implementation, analysis, and attacks. *Inf. Softw. Technol.*, 51(1):56–67, 2009.

[36] Christian S. Collberg and Clark Thomborson. Watermarking, Tamper-Proofing, and obfuscation - tools for software protection. In *IEEE Transactions on Software Engineering*, volume 28, page 735746, August 2002.

[37] Christian S. Collberg, Clark Thomborson, and Gregg M. Townsend. Dynamic graph-based software fingerprinting. *ACM Trans. Program. Lang. Syst.*, 29(6):35, 2007.

[38] Patrick Cousot and Radhia Cousot. An abstract interpretation-based framework for software watermarking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 173–185, Venice, Italy, 2004. ACM. ISBN 1-58113-729-X.

[39] Ingemar J. Cox, Joe Kilian, Frank Thomson Leighton, and Talal Shamoon. A secure, robust watermark for multimedia. In *Proceedings of the First International Workshop on Information Hiding*, pages 185–206. Springer-Verlag, 1996. ISBN 3-540-61996-8.

[40] D. Curran, N.J. Hurley, and M. O. Cinneide. Securing java through software watermarking. In *Proceedings of the 2nd international conference on Principles and practice of programming in Java*, page 311324, 2003.

[41] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 2535, New York, NY, USA, 1989. ACM. ISBN 0-89791-294-2. doi: 10.1145/75277.75280.

[42] Barthlmy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. *SIGPLAN Not.*, 43(10):313328, 2008. ISSN 0362-1340. doi: 10.1145/1449955.1449790.

[43] Markus Dahm. Byte code engineering with the bcel api. Technical report, Freie University, Berlin, Institut fur Informatik, 2001.

[44] Robert Davidson and Nathan Myhrvold. Method and system for generating and auditing a signature for a computer program, June 1996. Microsoft Corporation, US Patent 5559884.

[45] Robert I Davidson, Nathan Myhrvold, Keith Randel Vogel, Gideon Andreas Yuval, Richard Shupak, and Norman Eugene Apperson. Method and system for improving the locality of memory references during execution of a computer program, September 2001.

[46] Stephen Drape, Anirban Majumdar, and Clark Thomborson. Slicing aided design of obfuscating transforms. *Computer and Information Science, ACIS International Conference on*, 0:1019–1024, 2007. doi: 10.1109/ICIS.2007.167.

[47] Emmanuel Dupuy. Java decompiler, 2009. URL `http://java.decompiler.free.fr/`.

[48] Dave Dyer. Java decompilers compared, July 1997. URL `http://www.javaworld.com/javaworld/jw-07-1997/jw-07-decompilers.html`.

[49] Bruce Eckel and Kirk Pepperdine. *JDT more correct than Javac.* 2006. Published: \urlhttp://www.theserverside.com/news/thread.tss?thread_id=38644.

[50] Torbjrn Ekman and Grel Hedin. The JastAdd extensible java compiler. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, page 118, 2007.

[51] Rakan El-Khalil. Hydan, 2004. URL `http://www.crazyboy.com/hydan/`.

[52] Rakan El-khalil and Angelos D. Keromytis. Hydan: Hiding information in program binaries. In *in International Conf. on Information and Communications Security, ICICS*. Springer-Verlag, 2004.

[53] Rakan El-Khalil and Angelos D. Keromytis. Hydan: Information hiding in program binaries. In *International Conference on Informaton and Communications Security*, 2004.

[54] Michael Van Emmerik. *Static Single Assignment for Decompilation.* PhD thesis, The University of Queensland, 2007.

[55] Mike Van Emmerik. Java decompiler tests. http://www.program-transformation.org/Transform/JavaDecompilerTests, February 2003. URL `http://www.program-transformation.org/Transform/JavaDecompilerTests`.

[56] Mike Van Emmerik and Trent Waddington. Using a decompiler for Real-World source recovery. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, page 2736, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2243-2.

[57] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants.* Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[58] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(13):3545, December 2007.

[59] Barry Fagin and Martin Carlisle. Connect Four(TM) game, written in ada, 2005. URL `http://webdiis.unizar.es/asignaturas/EDA/gnat/jgnat/connect_four/test.html`. 2005.

[60] Stephen N. Freund. The costs and benefits of java bytecode subroutines. In *In Formal Underpinnings of Java Workshop at OOPSLA*, 1998.

[61] Kazuhide Fukushima and Kouichi Sakurai. A software fingerprinting scheme for java using classfiles obfuscation, 2004.

[62] Kazuhide Fukushima, Toshihiro Tabata, Toshiaka Tanaka, and Kouichi Sakurai. A software fingerprinting scheme for java using class structure transformation. *Transactions of Information Processing Society of Japan*, 46(8):2042–2052, August 2005. ISSN 03875806.

[63] Etienne Gagnon, Laurie J. Hendren, and Guillaume Marceau. Efficient inference of static types for java bytecode. In *Static Analysis Symposium*, page 199219, 2000. URL `www.sable.mcgill.ca/publications`.

[64] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, St. Petersburg Beach, Florida, United States, 1996. ACM. ISBN 0-89791-769-3.

[65] Daofu Gong, Fenlin Liu, Bin Lu, Ping Wang, and Lan Ding. Hiding information in java class file. In *Proceedings of the 2008 International Symposium on Computer Science and Computational Technology - Volume 02*, pages 160–164. IEEE Computer Society, 2008. ISBN 978-0-7695-3498-5. doi: 10.1109/ISCSCT.2008.231.

[66] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, Boston, Mass., 2000. ISBN 0-201-31008-2.

[67] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005. ISBN 0321246780.

[68] K John Gough and Diane Corney. Implementing languages other than java on the java virtual machine. 2001.

[69] Vladimir Grishchenko and Johann Gyger. JadClipse, 2009. URL `http://jadclipse.sourceforge.net/wiki/index.php/Main_Page`.

[70] Gaurav Gupta and Josef Pieprzyk. Source code watermarking based on function dependency oriented sequencing. In *Proceedings of the 2008 International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 965–968. IEEE Computer Society, 2008. ISBN 978-0-7695-3278-3.

[71] Gael Hachez. *A Comparative Study of Software Protection Tools Suited for E-Commerce with Contributions to Software Watermarking and Smart Cards*. PhD thesis, Universite Catholique de Louvain, March 2003.

[72] Peter Haggar. Understanding bytecode makes you a better programmer. http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/, July 2001.

[73] Maurice H Halstead. *Elements of software science (Operating and programming systems series)*. Elsevier, 1977. ISBN 0444002057. Published: Hardcover.

[74] James Hamilton and Sebastian Danicic. An evaluation of current java bytecode decompilers. In *Ninth IEEE International Workshop on Source Code Analysis and Manipulation*, volume 0, pages 129–136, Edmonton, Alberta, Canada, 2009. IEEE Computer Society. doi: 10.1109/SCAM.2009.24.

[75] James Hamilton and Sebastian Danicic. An evaluation of static java bytecode watermarking. In *Proceedings of the International Conference on Computer Science and Applications (ICCSA'10)*, The World Congress on Engineering and Computer Science (WCECS'10), San Francisco, October 2010. ISBN 978-988-17012-0-6. To appear.

[76] James Hamilton and Sebastian Danicic. A survey of software watermarking by register allocation. Technical report, Goldsmiths, University of London, Department of Computing, August 2010.

[77] Frank Harary and Edgar M Palmer. *Graphical Enumeration*. Academic Press, New York,, 1973. ISBN 0123242452.

[78] Kazuhiro HATTANDA and Shuichi ICHIKAWA. The evaluation of davidsons digital signature scheme. *IEICE TRANS. FUNDAMENTALS*, E87A(1), January 2004.

[79] Yong He. *Tamperproofing a Software Watermark by Encoding Constants*. Masters thesis, University of Auckland, June 2002.

[80] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, 1974.

[81] Matthew S. Hecht and Jeffrey D. Ullman. Flow graph reducibility. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 238–250, Denver, Colorado, United States, 1972. ACM.

[82] Jochen Hoenicke. JODE, 2004. URL `http://jode.sourceforge.net/`. 2004.

[83] Keith Holmes. Computer software protection, February 1994.

[84] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):10311052, 1997. ISSN 0164-0925. doi: 10.1145/267959.269971.

[85] Zhu Jian-qi, Wang Ai-min, and Liu Yan-heng. Tamper-proofing software watermarking scheme based on constant encoding. *Education Technology and Computer Science, International Workshop on*, 1:129–132, 2010. doi: 10.1109/ETCS.2010.429.

[86] Zetao Jiang, Rubing Zhong, and Bina Zheng. A software watermarking method based on Public-Key cryptography and graph coloring. In *Genetic and Evolutionary Computing, 2009. WGEC '09. 3rd International Conference on*, pages 433–437, 2009. doi: 10.1109/WGEC.2009.76. URL `http://www.computer.org/portal/web/csdl/doi/10.1109/WGEC.2009.76`.

[87] Zhu Jianqi, Liu YanHeng, and Yin KeXin. A novel dynamic graph software watermark scheme. In *Proceedings of the 2009 First International Workshop on Education Technology and Computer Science - Volume 03*, pages 775–780. IEEE Computer Society, 2009. ISBN 978-0-7695-3557-9.

[88] k00dr. Cracking the allatori string encryption, February 2008. URL http://www.moparisthebest.com/smf/index.php?topic=238584.0.

[89] R. M Karp. Reducibility among combinatorial problems. In R. E Miller and J. W Thatcher, editors, *Complexity of Computer Computations*, page 85103. Plenum Press, 1972.

[90] Kearney, Sedlmeyer, Thompson, Gray, and Adler. Software complexity measurement. *Commun. ACM*, 29(11):10441050, 1986. ISSN 0001-0782. doi: 10.1145/7538.7540.

[91] Malik Sikandar Hayat Khiyal, Aihab Khan, Sehrish Amjad, and M. Shahid Khalil. Evaluating effectiveness of tamper proofing on dynamic graph software watermarks. *CoRR*, abs/1001.1974, 2010.

[92] Todd B. Knoblock and Jakob Rehof. Type elaboration and subtype completion for java bytecode. *ACM Trans. Program. Lang. Syst.*, 23(2):243272, 2001. ISSN 0164-0925. doi: 10.1145/383043.383045.

[93] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd ed.): Fundamental Algorithms*, volume 1. Addison Wesley Longman Publishing Co., Inc., 3 edition, 1997. ISBN 0-201-89683-4.

[94] Donald E Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*, volume 3. Addison-Wesley Professional, 2 edition, May 1998. ISBN 0201896850. Published: Hardcover.

[95] Thomas Koshy. *Catalan Numbers with Applications*. Oxford University Press, 2008. ISBN 9780195334548.

[96] Pavel Kouznetsov. Jad - the fast java decompiler, 2001. URL http://www.varaneckas.com/jad.

[97] Douglas Kramer, Bill Joy, and David Spenhoff. The java[tm] platform. Technical report, Sun Microsystems, May 1996. URL http://java.sun.com/docs/white/platform/javaplatformTOC.doc.html.

[98] Eugene Kuleshov. Using the ASM framework to implement common java bytecode transformation patterns. Vancouver, Canada, March 2007.

[99] Karthik Kumar. JReversePro - java decompiler / disassembler, 2005. URL http://jreversepro.blogspot.com.

[100] Mark D. Ladue. When java was one: Threats from hostile byte code. In *Proceedings of the 20th National Information Systems Security Conference*, 1997.

[101] Eric Lafortune et al. ProGuard, July 2009. URL http://proguard.sourceforge.net/.

[102] Lay Lam. *Fleeting footsteps: tracing the conception of arithmetic and algebra in ancient China*. World Scientific, Singapore, River Edge NJ, rev. ed. edition, 2004. ISBN 9789812386960. English translation of "The Mathematical Classic" by Sun Zi.

[103] Tri Van Le and Yvo Desmedt. Cryptanalysis of UCLA watermarking schemes for intellectual property protection. In *Revised Papers from the 5th International Workshop on Information Hiding*, pages 213–225. Springer-Verlag, 2003. ISBN 3-540-00421-1. doi: 10.1007/3-540-36415-3_14. URL http://dx.doi.org/10.1007/3-540-36415-3_14.

[104] Hakun Lee and Keiichi Kaneko. New approaches for software watermarking by register allocation. In *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 63–68. IEEE Computer Society, 2008. ISBN 978-0-7695-3263-9.

[105] Hakun Lee and Keiichi Kaneko. Two new algorithms for software watermarking by register allocation and their empirical evaluation. In *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*, pages 217–222. IEEE Computer Society, 2009. ISBN 978-0-7695-3596-8.

[106] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reason.*, 30 (3-4):235269, 2003. ISSN 0168-7433.

[107] Jun Li and Quan Liu. Design of a software watermarking algorithm based on register allocation. In *e-Business and Information System Security (EBISS), 2010 2nd International Conference on*, pages 1–4, 2010. doi: 10.1109/EBISS.2010.5473660. URL `http://dx.doi.org/10.1109/EBISS.2010.5473660`.

[108] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999. ISBN 0201432943. Published: Paperback.

[109] Yang-Xia Luo, Jian-Hua Cheng, and Ding-Yi Fang. Dynamic graph watermark algorithm based on the threshold scheme. In *Proceedings of the 2008 International Symposium on Information Science and Engieering - Volume 02*, pages 689–693. IEEE Computer Society, 2008. ISBN 978-0-7695-3494-7.

[110] Anirban Majumdar, Stephen J. Drape, and Clark Thomborson. Slicing obfuscations: design, correctness, and evaluation. In *DRM '07: Proceedings of the 2007 ACM workshop on Digital Rights Management*, page 7081, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-884-8. doi: 10.1145/1314276.1314290.

[111] Michael Masnick. A detailed explanation of how the BSA misleads with piracy stats | techdirt. http://www.techdirt.com/articles/20080718/1226541724.shtml, 2008. URL `http://www.techdirt.com/articles/20080718/1226541724.shtml`.

[112] Jon Meyer and Troy Downing. *The Java Virtual Machine*. pub-ORA, pub-ORA, 1997. ISBN 1-56592-194-1. URL `http://oreilly.com/catalog/9781565921948/`.

[113] Jonathan Meyer, Daniel Reynaud, Iouri Kharon, et al. Jasmin, 2004. URL `http://jasmin.sourceforge.net/`.

[114] Jerome Miecznikowski. *New algorithms for a Java decompiler and their implementation in Soot*. Masters thesis, McGill University, 2003.

[115] Jerome Miecznikowski and Laurie Hendren. Decompiling java using staged encapsulation. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 368, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1303-4.

[116] Jerome Miecznikowski and Laurie J. Hendren. Decompiling java bytecode: Problems, traps and pitfalls. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, page 111127, London, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4.

[117] Anshuman Mishra, Rajeev Kumar, and P. P. Chakrabarti. A method-based Whole-Program watermarking scheme for java class files. 2008.

[118] Akito Monden. Jmark, 2003. URL `http://se.aist-nara.ac.jp/jmark/`.

[119] Akito Monden, Hajimu Iida, et al. A watermarking method for computer programs. In *Proceedings of the 1998 Symposium on Cryptography and Information Security, SCIS'98*. Institute of Electronics, Information and Communication Engineers, January 1998. in Japanese.

[120] Akito Monden, Hajimu Iida, Ken ichi Matsumoto, Katsuro Inoue, and Koiji Torii. Watermarking java programs. In *International Symposium on Future Software Technology '99*, pages 119–124, October 1999.

[121] Akito Monden, Hajimu Iida, Ken ichi Matsumoto, Koji Torii, and Katsuro Inoue. A practical method for watermarking java programs. In *COMPSAC '00: 24th International Computer Software and Applications Conference*, page 191197, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0792-1.

[122] Alan Mycroft. Type-Based decompilation (or program reconstruction via type reconstruction). In *ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems*, page 208223, London, UK, 1999. Springer-Verlag. ISBN 3-540-65699-5.

[123] Ginger Myles. Using software watermarking to discourage piracy. *Crossroads - The ACM Student Magazine*, 2004. URL `http://www.acm.org/crossroads/xrds10-3/watermarking.html`.

[124] Ginger Myles and Christian Collberg. Software watermarking through register allocation: Implementation, analysis, and attacks. In *International Conference on Information Security and Cryptology*, volume 2971/2004 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-21376-5. doi: 10.1007/978-3-540-24691-6_21. URL `http://dx.doi.org/10.1007/978-3-540-24691-6_21`.

[125] Ginger Myles and Christian Collberg. Software watermarking via opaque predicates: Implementation, analysis, and attacks. In *ICECR-7*, 2004.

[126] Ginger Myles, Christian Collberg, Zachary Heidepriem, and Armand Navabi. The evaluation of two software watermarking algorithms. *Softw. Pract. Exper.*, 35(10):923938, 2005. ISSN 0038-0644. doi: 10.1002/spe.v35:10.

[127] Nomair A. Naeem. *Programmer-Friendly Decompiled Java*. Masters thesis, January 2007. URL `http://www.sable.mcgill.ca/publications/thesis/#nomairMastersThesis`.

[128] Nomair A. Naeem and Laurie Hendren. Programmer-Friendly decompiled java. In *ICPC '06: Proceedings of the 14th IEEE International Conference onProgram Comprehension (ICPC'06)*, page 327336. IEEE Computer Society, 2006. ISBN 0-7695-2601-2. doi: http://dx.doi.org/10.1109/ICPC.2006.20.

[129] Nomair A. Naeem, Michael Batchelder, and Laurie Hendren. Metrics for measuring the effectiveness of decompilers and obfuscators. In *ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension*, page 253258, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2860-0. doi: http://dx.doi.org/10.1109/ICPC.2007.27.

[130] Jasvir Nagra and Clark Thomborson. Threading software watermarks. In Jessica J. Fridrich, editor, *Information Hiding*, volume 3200 of *Lecture Notes in Computer Science*, pages 208–223. Springer, 2004. ISBN 3-540-24207-4.

[131] Jasvir Nagra, Clark Thomborson, and Christian Collberg. A functional taxonomy for software watermarking. In Michael J. Oudshoorn, editor, *Aust. Comput. Sci. Commun.*, pages 177–186, Melbourne, Australia, 2002. ACS.

[132] Godfrey Nolan. *Decompiling Java*. APress, 2004. ISBN 1590592654.

[133] Kelly O'Hair. HPROF: a Heap/CPU profiling tool in J2SE 5.0. *Sun Developer Network*, Developer Technical Articles & Tips, November 2004. URL `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html`.

[134] Jens Palsberg and Di Ma. Javawiz, 2000. URL `http://www.cs.purdue.edu/homes/madi/wm/`. No longer available.

[135] Jens Palsberg, S. Krishnaswamy, Minseok Kwon, D. Ma, Qiuyun Shao, and Y. Zhang. Experience with software watermarking. In *ACSAC*, pages 308–316, 2000.

[136] Tapasya Patki. DashO java obfuscator review. Technical report, University of Arizona, Department of Computer Science, September 2008. URL `http://www.cs.arizona.edu/~collberg/Teaching/620/2008/Assignments/tools/DashO/index.html`.

[137] Mike Pearson. Piracy: BSA's $53b global piracy tab grossly inflated, argue skeptics. http://www.ecommercetimes.com/story/67049.html?wlc=1252256745, 2009. URL `http://www.ecommercetimes.com/story/67049.html?wlc=1252256745`.

[138] Z. Pervez, Noor-ul-Qayyum, Y. Mahmood, and H.F. Ahmad. Semblance based disseminated software watermarking algorithm. In *Computer and Information Sciences, 2008. ISCIS '08. 23rd International Symposium on*, pages 1–4, 2008. doi: 10.1109/ISCIS.2008.4717945.

[139] Zhou Ping, Chen Xi, and Yang Xu-Guang. The software watermarking for tamper resistant radix dynamic graph coding. *Inform. Technol. J.*, 9:1236–1240, June 2010. doi: 10.3923/itj.2010.1236.1240.

[140] Todd A Proebsting and Scott A Watterson. Krakatoa: Decompilation in java (Does bytecode reveal source?). page 185197, 1997.

[141] Gang Qu and Miodrag Potkonjak. Analysis of watermarking techniques for graph coloring problem. In *Proceedings of the 1998 IEEE/ACM international conference on Computer-aided design*, pages 190–193, San Jose, California, United States, 1998. ACM. ISBN 1-58113-008-2.

[142] Gang Qu and Miodrag Potkonjak. Hiding signatures in graph coloring solutions. In *Information Hiding*, pages 348–367, 1999.

[143] Gang Qu and Miodrag Potkonjak. Fingerprinting intellectual property using constraint-addition. In *Design Automation Conference*, pages 587–592, 2000.

[144] Raja V Rai and Laurie J Hendren. Jimple: Simplifying java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, Montreal, Quebec, Canada, 1998.

[145] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, 1994.

[146] Lyle Ramshaw. Eliminating go to's while preserving program structure. *J. ACM*, 35(4):893920, 1988. ISSN 0004-5411. doi: 10.1145/48014.48021.

[147] Mayon Software Research. ClassCracker 3, 2005. URL `http://mayon.actewagl.net.au/`.

[148] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978. ISSN 0001-0782. doi: 10.1145/359340.359342.

[149] Eva Rose. Lightweight bytecode verification. 1998.

[150] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, page 1227, New York, NY, USA, 1988. ACM. ISBN 0-89791-252-7. doi: 10.1145/73560.73562.

[151] Peter R. Samson. Apparatus and method for serializing and validating copies of computer software, February 1994. URL `http://www.freepatentsonline.com/5287408.html`.

[152] Zonglu Sha, Hua Jiang, and Aicheng Xuan. Software watermarking algorithm by coefficients of equation. *Genetic and Evolutionary Computing, International Conference on*, 0:410–413, 2009. doi: 10.1109/WGEC.2009.18.

[153] M. Shirali-Shahreza and S. Shirali-Shahreza. Software watermarking by equation reordering. In *Information and Communication Technologies: From Theory to Applications, 2008. ICTTA 2008. 3rd International Conference on*, pages 1–4, 2008. doi: 10.1109/ICTTA.2008.4530357. URL `http://dx.doi.org/10.1109/ICTTA.2008.4530357`.

[154] Smardec. Software development and information technology offshore outsourcing company. http://www.smardec.com/, 2008. URL `http://www.smardec.com/`.

[155] Smardec. Allatori java obfuscator, September 2009. URL `http://www.allatori.com/`.

[156] Ahpah Software. SourceAgain, 2004. URL `http://www.ahpah.com/cgi-bin/suid/~pah/demo_license.cgi`. year = 2004.

[157] Jose Sogiros. Is protection software needed watermarking versus software security. http://bb-articles.com/watermarking-versus-software-security, March 2010. URL `http://bb-articles.com/watermarking-versus-software-security`.

[158] Jeremy Spinrad. *Efficient graph representations*. Number 19 in Fields Institute Monographs. American Mathematical Society, Providence R.I., 2003. ISBN 9780821828151.

[159] Julien Stern, Gael Hachez, Francois Koeune, and Jean-Jacques Quisquater. Robust object watermarking: Application to code. In *Information Hiding Workshop '99*, pages 368–378, 1999.

[160] R. F. Strk, J. Schmid, and E. Brger. *Java and the Java Virtual Machine: Definition, Verification and Validation*. Springer-Verlag, 2001.

[161] Sun Microsystems, Inc. Java debug interface, 2005. URL `http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/index.html`.

[162] Smita Thaker. *Software Watermarking via Assembly Code Transformations*. Masters thesis, San Jose State University, 2004.

[163] Clark Thomborson, Jasvir Nagra, Ram Somaraju, and Charles He. Tamper-proofing software watermarks. In *ACSW Frontiers '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, page 2736, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.

[164] Raja Valle-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[165] Various. The easter egg archive. http://www.eeggs.com/, 2009. URL `http://www.eeggs.com/`.

[166] Ramarathnam Venkatesan and Vijay Vazirani. Technique for producing through watermarking highly tamper-resistant executable code and resulting watermarked code so formed, May 2006. Microsoft Corporation, US Patent: 7051208.

[167] Ramarathnam Venkatesan, Vijay Vazirani, and Saurabh Sinha. A graph theoretic approach to software watermarking. In *Proceedings of the 4th International Workshop on Information Hiding*, 2001.

[168] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 1996. ISBN 0079132480.

[169] Mark Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, page 439449, Piscataway, NJ, USA, 1981. IEEE Press. ISBN 0-89791-146-6.

[170] Wang Yong and Yang Yixian. A software watermark database scheme based on PPCT. In *CIHW2004*, 2004.

[171] J. Zhu, Y. Liu, and K. Yin. A novel planar IPPCT tree structure and characteristics analysis. *Journal of Software*, 5(3):344, 2010.

[172] Jianqi Zhu, Kexin Yin, and Yanheng Liu. A novel DGW scheme based on 2D_PPCT and permutation. *Multimedia Information Networking and Security, International Conference on*, 2:109–113, 2009. doi: 10.1109/MINES.2009.177.

[173] William Zhu and Clark Thomborson. Algorithms to watermark software through register allocation. In *Digital Rights Management. Technologies, Issues, Challenges and Systems*, volume 3919 of *Lecture notes in computer science*, pages 180–191, Berlin, ALLEMAGNE, 2006. Springer. ISBN 978-3-540-35998-2. doi: 10.1007/11787952_14. URL `http://dx.doi.org/10.1007/11787952_14`.

[174] William Zhu and Clark Thomborson. Extraction in software watermarking. In Sviatoslav Voloshynovskiy, Jana Dittmann, and Jessica J. Fridrich, editors, *MM&Sec*, pages 175–181. ACM, 2006. ISBN 1-59593-493-6. URL `http://dblp.uni-trier.de/db/conf/mmsec/mmsec2006.html#ZhuT06`.

[175] William Zhu and Clark Thomborson. Recognition in software watermarking. In *Proceedings of the 4th ACM international workshop on Contents protection and security*, pages 29–36, Santa Barbara, California, USA, 2006. ACM. ISBN 1-59593-499-5.