Decompiling Java

James Hamilton

May 6, 2009

Contents

Abstract

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 [51].

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 [89] . The Java Virtual Machine Specification [65] 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 [65] and updated for Java 1.6 in JSR202 [29].

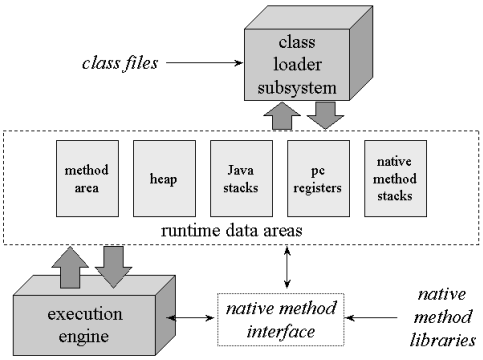

Figure 1.1: Java Virtual Machine internal architecture [89]

An advantage of the virtual machine architecture is portability - any machine that implements the Java Virtual Machine Specification [65] is able to execute Java bytecode hence the slogan "Write once, run anywhere" [60]. Java bytecode is not strictly linked to the Java language and there are many compilers, and other tools, available which produce Java bytecode [86, 49] such as the Jikes compiler, Eclipse Java Development Tools or Jasmin bytecode assem-

bler. 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 [49] which, combined, provides an attractive platform for other languages.

Compilation is the act of transforming a high-level language, such as C or Java, into a low-level language such as machine code or bytecode. In contrast, decompilation is the reverse - the act of transforming a low-level language into a high-level language [30]. We are concerned with the latter, though in actual fact the two tasks are similar: both parse a source language and transform it into another language.

The task of decompilation is much harder than compilation as a compiled program loses a lot of information which existed in the high-level source language. Due to the difficulties in decompilation a decompiler can perform automatic translation of *some* low-level source and semi-automatic translation of other source programs [31].

The problems to be solved by a general decompiler can be divided into different categories [41]:

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.

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[1] at McGill University. They have created Soot [87] - a Java optimisation framework which includes a Java decompiler called Dava [71].

Several commercial decompilers exists (e.g. Class Cracker) thought most have been unmaintained for several years.

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

Chapter 2

Decompiling Java bytecode

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

1. local variable typing
2. merging stack-based instructions into expressions
3. arbitrary control flow
4. exceptions and synchronisation

The remaining 6 problems for general decompilation (see page 2) do not apply for the decompilation of Java bytecode due to the large amount of information retained in a Java class file.

Arbitrary Java bytecode can be generated by a number of different tools, not just Sun's javac Java compiler which also creates a problem for decompilationthree.

1 Java Bytecode Generation and 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 [65] and updated for Java 1.6 in JSR202 [29]. 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 1, page 4) 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.

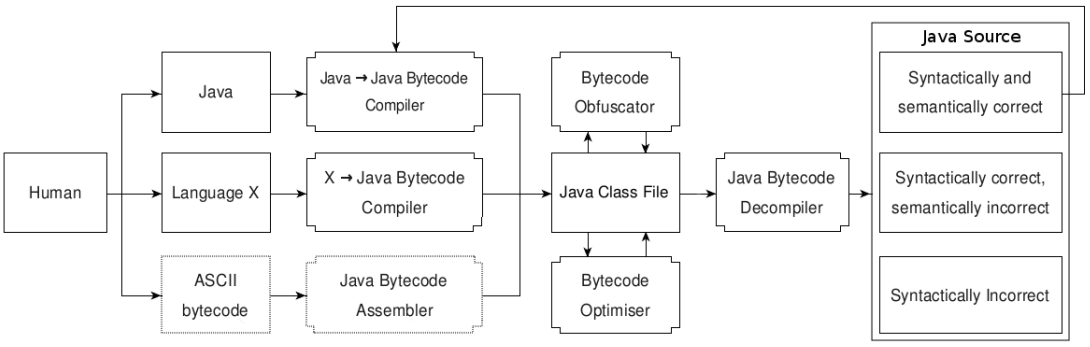

Figure 2.1: Compilation and decompilation of Java Bytecode

1.1 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 [65, 29]. There are several such compilers available besides Sun's javac.

javac Sun's Java compiler, javac [9], is the original Java compiler from the creators of Java. It is now part of the open-source OpenJDK [12].

GNU Java Compiler GCJ [3] 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 [10] 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 [47] 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 [79] 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 [45] form the basis of the Eclipse Java IDE and includes an incremental Java compiler. The Eclipse JDT adheres to the Java Language Specification [47] more closely than javac [38].

Janino Janino [5] 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 [11] 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 [6, 39] is a Java compiler built with the JastAdd compiler compiler system [40]. 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 [47] and Java Virtual Machine Specification [65, 29] 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 [83]. Not all compilers listed support the latest version of Java (currently 1.6).

1.2 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 [1] 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 [61, 28] and is used as a component in many other Java tools [21]. 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 2.1, page 7 shows a Java program using the ASM libraries to generate a Hello World Java program.

BCEL Byte Code Engineering Library [20] is a library for manipulating and generating class files. BCEL was originally created by Markus Dham [34] 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 2.2, page 8 shows the 'BCELified' Hello World Java source.

Jasmin Jasmin [68] is a Java bytecode assembler initially written in 1996 as companion to the book 'Java Virtual Machine' [67] 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 [87]. An extension to the Jasmin language known as JasminXT has been defined as part of the tinapoc project [15] - 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 2.3, page 9.

serp TODO http://serp.sourceforge.net/ still active?

Cojen Cojen [2] is a bytecode generation library with a primary goal of making raw Java class-file generation easy. The library contains classes which equate to bytecode instructions. Cojen is a fork of the no longer maintained Tea Trove bytecode library [14].

[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 2.1: 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,"HelloWorld",null,"java/lang/Object",null);

    // creates a GeneratorAdapter for the (implicit) constructor
    Method m = Method.getMethod("void <init> ()");
    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 'main' method
    m = Method.getMethod("void main (String[])");
    mg = new GeneratorAdapter(ACC_PUBLIC + ACC_STATIC, m, null, null, cw);
    mg.getStatic(Type.getType(System.class),
            "out",
            Type.getType(PrintStream.class));
    mg.push("Hello world");
    mg.invokeVirtual(Type.getType(PrintStream.class),
            Method.getMethod("void println (String)"));
    mg.returnValue();
    mg.endMethod();

    cw.visitEnd();

    code = cw.toByteArray();

    FileOutputStream output = new FileOutputStream("HelloWorld.class");
    output.write(code);
    output.close();
  }
}
```

Listing 2.2: 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("HelloWorld", "java.lang.Object", "
                HelloWorld.java", 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[] {  },
                "<init>", "HelloWorld", il, cp);

    InstructionHandle ih_0 = il.append(factory.createLoad(Type.OBJECT, 0));

    il.append(factory.createInvoke("java.lang.Object", "<init>",
                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[] { "args" }, "main", "HelloWorld", il, cp);

    // getstatic System.out
    InstructionHandle ih_0 = il.append(
        factory.createFieldAccess("java.lang.System",
                "out", new ObjectType("java.io.PrintStream"),
                        Constants.GETSTATIC));

    il.append(new PUSH(cp, "Hello World")); // ldc "Hello World"

    // System.out.println("Hello World")
    il.append(factory.createInvoke("java.io.PrintStream", "println",
                                Type.VOID, new Type[] { Type.STRING },
                                Constants.INVOKEVIRTUAL));

    // return
    InstructionHandle ih_8 = il.append(factory.createReturn(Type.VOID));

    // calculate max stack and locals
    method.setMaxStack();
    method.setMaxLocals();

    // add method to class
    cg.addMethod(method.getMethod());
    il.dispose();
  }

  public static void main(String[] args) throws Exception {
    Example creator = new Example();
    creator.createClassFile(new FileOutputStream("HelloWorld.class"));
  }
}
```

Listing 2.3: 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 'this'
   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 "Hello World" onto stack
        ldc "Hello World"
        ; invoke System.out.println("Hello World")
        invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V

        return
.end method
```

Cojen is another Java library, very similar to ASM and BCEL, but seems simpler - see the Hello World example in listing 2.4, page 10.

Jamaica Jamaica, The JVM Macro Assembler, is an easy to learn assembly language for JVM programming [4]. 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

Listing 2.4: Hello World in Cojen source.

```
public class CojenTest {

  public static void main(String[] args) throws Exception {
    ClassFile classFile = new ClassFile("HelloWorld");
    FileOutputStream outputFile = new FileOutputStream("HelloWorld.class");

    classFile.addDefaultConstructor();

    TypeDesc[] params = new TypeDesc[] {TypeDesc.STRING.toArrayType()};
    MethodInfo mainMethod = classFile.addMethod(Modifiers.PUBLIC_STATIC,
                                                "main", null, params);
    CodeBuilder b = new CodeBuilder(mainMethod);

    TypeDesc printStream = TypeDesc.forClass("java.io.PrintStream");

    b.loadStaticField("java.lang.System", "out", printStream);
    b.loadConstant("Hello World");

    params = new TypeDesc[] {TypeDesc.STRING};
    b.invokeVirtual(printStream, "println", null, params);

    b.returnVoid();

    classFile.writeTo(outputFile);
    outputFile.close();
  }

}
```

automatically generate common bytecode instructions. For example, listing 2.6, page 11 shows how the three lines of bytecode in listing 2.5 can be condensed into one line using the *%println* macro.

Listing 2.5, page 10 shows the Hello World program using Jamaica.

Listing 2.5: Hello World in Jamaica source.

```
public class HelloJamaica {

        public static void main(String[] args) {
                getstatic System.out PrintStream
                ldc "Hello World"
                invokevirtual PrintStream.println(String)void
        }

}
```

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.

Listing 2.6: Hello World in Jamaica source using a macro.

```
public class HelloJamaica {

        public static void main(String[] args) {
                %println "Hello World"
        }

}
```

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' [67] 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 [15] includes a new version 2.0 of Jasmin and an extension of the language known as JasminXT.

1.3 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 [86].

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

1.4 Bytecode Optimisers

A bytecode optimiser takes as input a Java class file, performs optimising transformations and outputs a semantically equivilent 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 equivilent 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 [13] is an open-source Java class file shrinker, optimizer, obfuscator, and preverifier.

JODE JODE [8] 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 [16] is a commerical obfuscator and optimiser sold by Zelix Pty Ltd.

1.5 Bytecode Obfuscators

A bytecode obfuscator takes a Java class file as input, performs some obfuscating transformations and outputs a semantically equivilent 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 2.7), 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 2.7: Randomise Class.

```
public class Randomise {

        public static void main(String[] args) {
                sayHello();
        }

        private static void sayHello() {
                System.out.println("Hello World");
        }

}

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 [13] is an open-source Java class file shrinker, optimizer, obfuscator, and preverifier.

Zelix Classmaster Zelix Classmaster [16] is a commerical obfuscator and optimiser sold by Zelix Pty Ltd.

2 Why is it easy?

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 over machine code decompilers [41]:

- Data is already separated from code as all the data is stored in the class files constant pool (see listing 2.8, page 15)
- 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 (TODO what is this?)

Listing 2.8: Hello World bytecode

```
public class HelloWorld extends java.lang.Object

#DATA BEGINS

const #1 = String         #17;     //  Hello World
const #2 = Method         #21.#3; //  java/lang/Object."<init>":()V
const #3 = NameAndType    #4:#22;//  "<init>":()V
const #4 = Asciz          <init>;
const #5 = Asciz          Ljava/io/PrintStream;;
const #6 = class          #19;     //  java/io/PrintStream
const #7 = Asciz          java/lang/Object;
const #8 = Asciz          ([Ljava/lang/String;)V;
const #9 = Asciz          Code;
const #10 = Asciz         HelloWorld;
const #11 = Asciz         println;
const #12 = NameAndType   #11:#16;//  println:(Ljava/lang/String;)V
const #13 = Method        #6.#12; //  java/io/PrintStream.println:(Ljava/lang/String;)V
const #14 = class         #10;     //  HelloWorld
const #15 = class         #20;     //  java/lang/System
const #16 = Asciz         (Ljava/lang/String;)V;
const #17 = Asciz         Hello World;
const #18 = Asciz         main;
const #19 = Asciz         java/io/PrintStream;
const #20 = Asciz         java/lang/System;
const #21 = class         #7;      //  java/lang/Object
const #22 = Asciz         ()V;
const #23 = Asciz         out;
const #24 = NameAndType   #23:#5;//  out:Ljava/io/PrintStream;
const #25 = Field         #15.#24;       //  java/lang/System.out:Ljava/io/PrintStream;

# DATA ENDS

# CODE BEGINS

{
public HelloWorld();
  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #2; //Method java/lang/Object."<init>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   Stack=2, Locals=1, Args_size=1
   0:   getstatic       #25; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   ldc     #1; //String Hello World
   5:   invokevirtual   #13; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8:   return

}

# CODE ENDS
```

3 Why is it hard?

Decompiling Java bytecode is relatively easy but only in comparison to decompiling machine code. There are still several problems to overcome [71, 41]:

- Type inference for local variables
- Merging stack-based instructions into expressions
- Arbitrary control flow
- Exceptions and synchronisation

3.1 Type Inference

TODO: type inference general, functional programming - ML, statically typed languages (e.g Java). type inference vs type checking

Type inference in decompilation is a general problem in decompilation ([41], etc) 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 CIL bytecode as all types are explicit in the assembly files [41].

One of the first papers written on decompiling Java was a description of a decompiler named Krakatoa [80] 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.

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 [46]. The type analysis performed by the bytecode Verifier estimates the types of local variables at each program point which 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 [64]. The type inference problem for decompilation must give types to each variable that are valid for all uses of those variables.

However, if the type inference algorithm is optimised for the common-case rather than the worst case it is still possible to perform type analysis form most real-world code as worst-case scenerios are unlikely in real-world code [25]. Bellamy *et al.* [25] provide a common-case optimised algorithm to perform type analysis which outperforms Gangon *et al.*'s [46] 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.

Each method frame in the JVM contains an array of local variables referenced by index. The possible types that can be stored in a single local variable slot are boolean, byte, char, short, int, float, reference, or returnAddress and a pair of local variables can hold long or double. Local variables occupy slots in the order they are declared in the Java source starting with the method parameters, followed by other local variables. If it is an instance method slot 0 holds a reference to the class (known in Java source by the keyword *this*).

Figure 2.2: Method sum showing bytecode and local variable table

Figure 2.2, page 17 shows a simple method which accepts an array of integers and returns the sum. The Java source is shown next to the bytecode generated by *javac* 1.6 and below is a conceptual diagram showing the method's local variable table. As can be seen from the diagram local variables are untyped. The opcodes used in the bytecode deal with integers and object references which can be seen from the prefix of the bytecode mnemonic - i for integer and a for object reference. The maximum stack height is 3 which is calculated at compile time by *javac*. The bytecode Verifier will check at each program point that the opcode types match the type of local variables and stack variables; it will also check that the maximum stack height does not exceed that which has been determined by *javac*.
Types in Java bytecode, like Java, are divided into primitive types and reference types where

primitives types are either numeric types, boolean type or returnAddress type. Numeric types are divided into integral types or floating-point types. Integral types are byte, short, int, long, char and floating-point types are float and double. The boolean type encodes either true or false but in actuality the JVM provides limited support for it. A Java compiler will convert Java boolean types into int types where 0 is false and 1 is true.

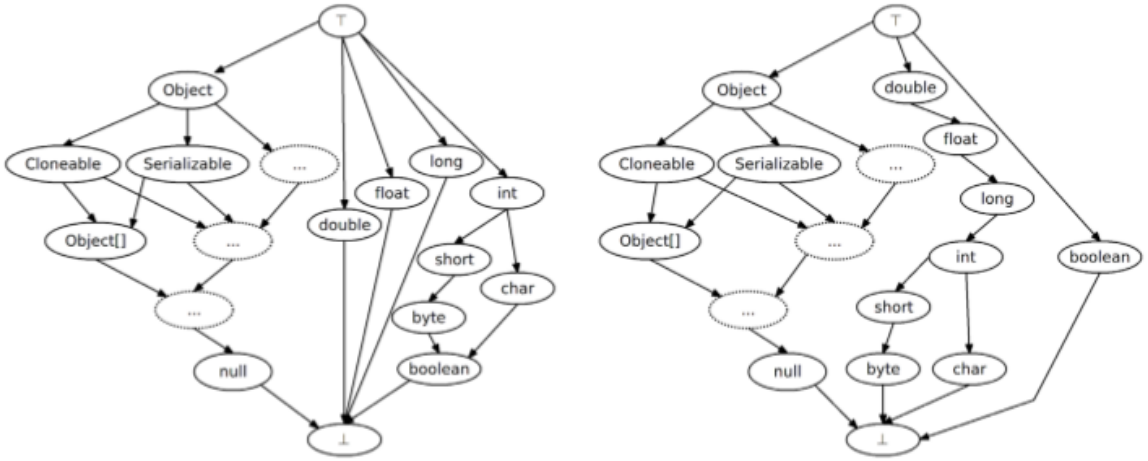

Figure 2.3: The type hierarchy in Java bytecode and Java [25]

Each numeric type has a set of opcodes which act on just that type, for example *iadd*, *ladd*, *fadd*, and *dadd* add two numeric values of their respective subtype and produce a numeric type of the same subtype. Opcodes prefixed with *a*, e.g. *aload*, work with object references. Opcodes prefixed with *b*, e.g. *baload*, work with bytes.

Figure 2.4, page 18 shows that during compilation the Java types boolean, short, int, char and byte are all converted into the int type [57]. In bytecode there is no indication as to which subtype of integer is involved. A decompiler could not retrieve the original code as there are may be no clues as to what integer subtypes are involved as the variables aren't used (in reality this example is a non-problem as the variables aren't used). Some clues which may be available are types in method signatures. The program in listing 2.9, page 19 shows an example in which some decompilers omit the *int* typecast.

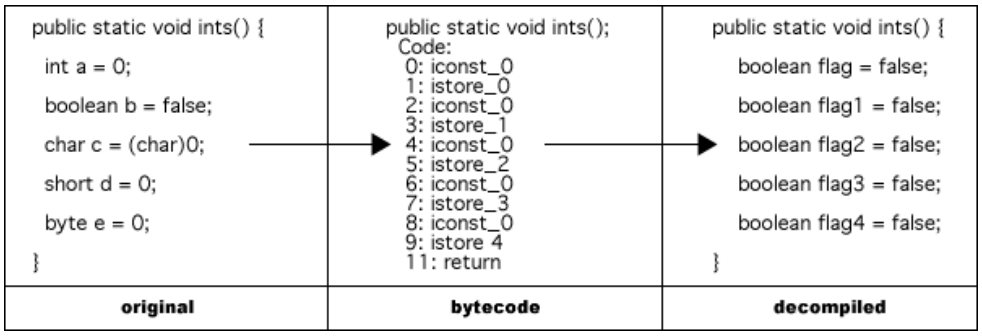

Figure 2.4: 5 different types in Java are convolved in Java bytecode

It is possible for a local variable slot to take values of distinct types at different places in a method [46] which is not possible in Java. For example listing 2.10 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

Listing 2.9: int type cast example

```
public class Type {

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

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

}
```

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 [57] 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 [23, 82, 33] 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. See page 21 for more about SSA form.

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 [46]. This causes problems for some decompilers which expect *javac* generated code. Using an SSA form, as in [57], may change the type hierarchy when types conflict due to interfaces [71].

Listing 2.11, page 20 (from [85]) shows a program which Dava decompiles better than Jad. Jad will type x as Object and insert a typecast in the invocation of method m2(Comparable) whereas Dava correctly types x 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 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.

Java 1.6 records the type java.lang.Comparable in m1's StackMapTable which should make the code easier to decompile (TODO: does it? does Dava use this?).

Listing 2.10: 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
```

Listing 2.11: What is the type of x? [85]

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

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

                m2(x);
        }

        public void m2(Comparable x) {
        }




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



//java 1.6
  StackMapTable: number_of_entries = 2
   frame_type = 9 /* same */
   frame_type = 252 /* append */
     offset_delta = 1
     locals = [ class java/lang/Comparable ]
```

Static Single Assignment Form

Static Single Assignment [23, 82, 33] 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*`) {`
$\quad x_1 = x_0 \times 6$
`}else{`
$\quad x_2 = x_1 + 1$
`}`

$$y = x_? + 3$$

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*`) {`
$\quad x_1 = x_0 \times 6$
`}else{`
$\quad x_2 = x_1 + 1$
`}`

$$\begin{aligned} x_3 &= \phi(x_1, x_2) \\ y &= x_3 + 3 \end{aligned}$$

The bytecode from listing 2.10, page 20 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.

3.2 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 [41]. This problem is solved by introducing variables to represent the stack positions.

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
```

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.

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
```

3.3 Arbitrary 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 creators of the Java language decided to omit the *goto* statement from the language[6] due to frequent misuse of the control in other programming languages [48]. Many computer scientists have criticised the existence of a *goto* control in higher level languages, most notably Edsger Dikjstra in his 1968 letter entitled 'Go To Statement Considered Harmful' [36], as it allows programmers to easily create 'spaghetti' code. Despite many programmers misuse of the *goto* statement other well known computer scientists, such as Knuth, have described ways in which to use *goto*s effectively and efficiently [58].

Java 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* statement. A labelled *break* statement terminates the labelled statement, such as a for-loop, and is useful when using nested loops to break the outer loop.

switch statements in Java 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 multilevel loop exit statements [81]. The Krakatoa decompiler uses an extended version of Ramshaw's *goto* elimination technique [80].

The separate languages of Java and Java bytecode mean that many bytecode features such as arbitrary control flow do not translate easily to Java source. For example, Java bytecode can contain non-reducible control flow which cannot be represented in Java source requiring techniques such as described in [54] to be applied to transform irreducible control flow graphs to reducible control flow graphs.

TODO: more

[6] *goto* is still a reserved word in Java even though it is unused

Structuring Loops and Conditionals

There are several conditional jump instructions in the Java bytecode instruction set which compare values on the stack, which Java *if* statements compile to.

The conditional jump instructions can be divided into three groups. The first group of instructions pops the first integer (a) from the stack and compares it against the constant zero.

`ifeq` X	`if(`$a = 0$`) goto` X
`ifne` X	`if(`$a \neq 0$`) goto` X
`iflt` X	`if(`$a < 0$`) goto` X
`ifgt` X	`if(`$a > 0$`) goto` X
`ifle` X	`if(`$a \leq 0$`) goto` X
`ifge` X	`if(`$a \geq 0$`) goto` X

The second group pops two integer values (a, b) from the top of the stack stack and compares them to each other.

`if_icmpeq` X	`if(`$a = b$`) goto` X
`if_icmpne` X	`if(`$a \neq b$`) goto` X
`if_icmplt` X	`if(`$a < b$`) goto` X
`if_icmpgt` X	`if(`$a > b$`) goto` X
`if_icmple` X	`if(`$a \leq b$`) goto` X
`if_icmpge` X	`if(`$a \geq b$`) goto` X

The third group pops an object reference (a) from the top of the stack and compares it with the special *null* reference.

`ifnull` X	`if(`$a = null$`) goto` X
`ifnonnull` X	`if(`$a \neq null$`) goto` X

In Java bytecode there are no loop statements like there are in Java. Loops are created in bytecode by using conditional and unconditional control flow instructions.

The following while loop

```
int x = 5;

while(x >= 0) {
      x--;
}
```

would be expressed in bytecode as below

```
0:  iconst_5     push 5
1:  istore_0     x = 5
2:  iload_0      push x
3:  iflt  12     if(x < 0) goto 12
6:  iinc  1, -1  x--
9:  goto  2      goto 2
12: return       return
```

3.4 Exceptions and Synchronisation

An exception, as described in the Java Language Specification [47], 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*.

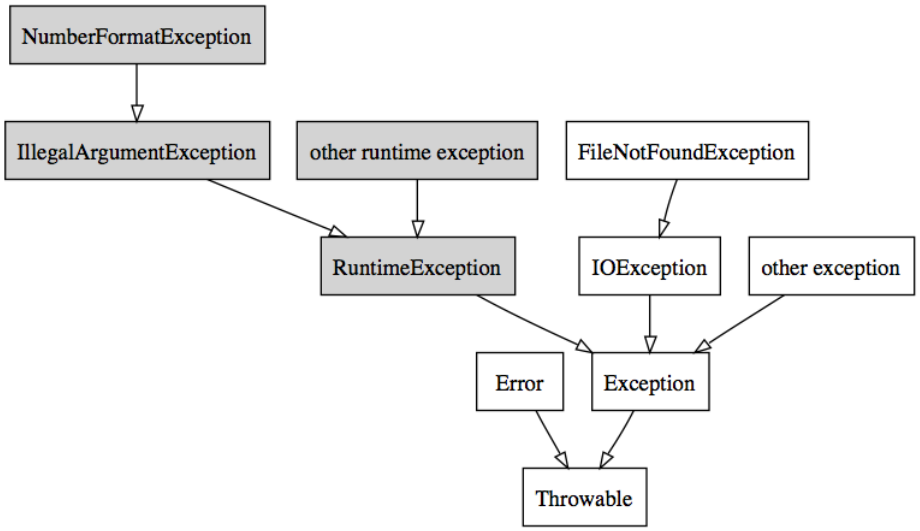

Figure 2.5: Java Exception Hierarchy. Unchecked exception classes are coloured grey.

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**) (Listing 2.12, page 27).

Figure 2.13, page 28 shows a try-catch block with a call to the method Integer.parseInt which accepts a string. If the string is a unicode character which represents an integer the correspoding

Listing 2.12: Structure of try-catch

```
try {
        //try something
}catch(Exception e) {
        //catch any Exception
        //don't worry about Error as recovery is usually impossible
}
```

Listing 2.13: 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");
}
```

integer is returned, otherwise a NumberFormatException is thrown. 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 of it - 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 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 checks 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 2.14, page 29 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 a a checked exception therefore any invocation of the constructor requires an exception handler or that the method doing the invoking 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 and it would need to much exception handling code.

Programmers can throw exceptions themselves using the *throw* keyword. Listing 2.15, page 29 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.

Listing 2.14: Example of checked exceptions

```
public class CheckedException {

        public static void main(String[] args) {

                try {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);
        }
        */
}
```

Listing 2.15: 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); }
}
```

Exceptions in Java bytecode

Java bytecode contains the necessary information to implement exceptions as specified by the Java Language Specification. 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 2.6 shows the bytecode, generated by javac 1.6, for listing 2.13, page 28 with highlighted try-catch sections. It contains two entries in the exception table corresponding to the two types of exception which are to be caught.

```
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 2.6: Java bytecode for listing 2.13, page 28 with highlighted try-catch sections.

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 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 offered to programmers to guarantee the execution even if the try block completes abrupbtly. The subroutine for the finally clause is invoked at each exit point of a try block and its associated catch block. The purpose of a finally clause is to provide a gaurented execution of code no matter if the try block exectued correctly or not. This allows 'clean-up' code to be executed such as closing of database connections or deleting temporary files which will be executed even if the try block completes abruptly.

Listing 2.16 shows an example of the try-finally construct and the output produced shown below the code. The finally clause is executed after the try block and if the try block completes abruptly, e.g. with a return, break or throwing of an exception, the finally block is still executed immediately after the try block exits.

Figure 2.8, page 35 shows the bytecode produced by javac 1.6 along with a control flow graph for lising 2.16, page 33.

Finally sections are coloured green, try areas are coloured blue and the catch exception handler is shown in black. All exit points from a blue area enter into a green area. Byte 5 exits a try section and imediately enters a finally section, where bytes 0 - 5 are the try section from the original program.

The JVM also watches for exceptions being thrown from bytes 0 - 5 which is indicated by being enclosed inside a blue section. If an exception is thrown within this section control is transfered to byte 19 which is the beginning of the exception handler. Byte 19 leads to another copy of the finally section. Both exit routes from bytes 0 - 5 (throwing an exception, or continuing execution from byte 5) execute a finally section.

Byte 19 is the opcode *astore_1* which expects to find an object reference to a Throwable object on the top of the stack. If this is not the case it will itself throw an exception and catch it itself. The next opcode in the sequence begins the second finally section, which is inside the catch section.

The original Java source did not contain a catch clause but one is generated by the compiler to enable the finally clause to be executed if an exception is thrown. If the catch clause was not inserted the method would complete abruptly and the exception would be re-thrown in the invokers method frame without invoking the finally clause.

The exception handler in this method serves the purpose of executing the finally clause and re-throwing the exception that it caught.

The finally clauses is represented in bytecode either by using subroutines or by inlining the subroutines whereever the subroutine would be invoked. Pre-1.4.2 releases of javac used subroutines to implement finally clauses while later versions use inline finally clauses. The Java Virtual Machine still supports Java subroutines but Sun's Java bytecode generation tools have not produced them since java 1.4.2.

Other tools may still produce code with subroutines as it is still supported by the Java Virtual Machine, for example the Jikes compiler produces bytecode with subroutines (see Figure 2.9, page 36).

Java Bytecode Subroutines

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. 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.

Figure 2.9, page 36 shows bytecode produced by Jikes for listing 2.16 along with its control flow graph. There is only one finally section (bytes 17 - 26, enclosed by green lines) in this version of the bytecode - this is the Java subroutine.

Bytes 0 - 5 and 28 represent the try section of the original Java program. The two ways of exiting this block are continuing execution after byte 28 or throwing an exception. Byte 28 invokes the jsr and the exception handler for begins at byte 11.

The exception handler is bytes 11 - 16. The purpose of the exception handler is to store a thrown exception, call the finally jsr and re-throw the stored exception. Byte 12 in the exception handler invokes the finally jsr.

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 (*astore_1* in Figure 2.9).

The *ret* opcode takes one operand that is the index of the local variable slot where the return address is stored. Execution then continues at the address loaded from the local variable slot (slot 1 in Figure 2.9).

Figure 2.9 has an overlapping try/catch section which causes difficulties for decompilers.

Subroutines are Polymorphic

Java subroutines are polymorphic over local variables which they do not use. Consider the Java program in listing 2.17 and (possible) bytecode for the program in listing 2.18, page 34.

The finally jsr begins at byte 21, ends at byte 26 and is called from three places in the method - bytes 7, 16 and 28. When the jsr is called at byte 7 local variable 1 should contain an integer value but when the jsr is called at byte 16 local variable 1 should contain a object reference value. The jsr is called with different types of variables in local variable slots at different points in the program.

Subroutines are **not** polymorphic over the types of variables that they use, for example local variable 0 must contain an integer type as bytes 22 and 25 load and store an int from local variable 0.

The bytecode was generated by jikes and slightly modified to demonstrate our point as the original output produced code which used an extra local variable than shown so the problem didn't occur. The Java Virtual Machine does not enforce any measures to ensure that local variables only take on one type and even if all compilers created code which only used local variable slots for one type the Java Virtual Machine would still accept other programs.

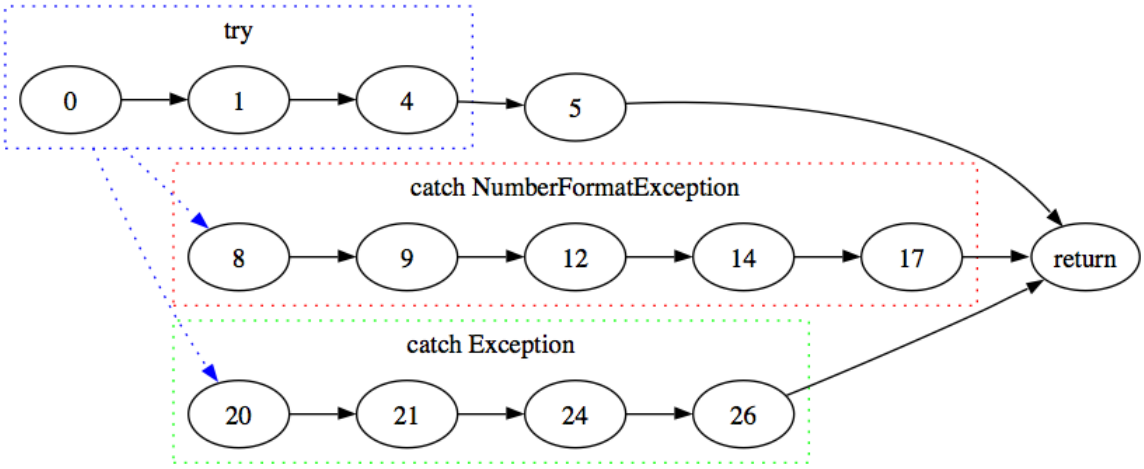

Figure 2.7: Java bytecode control flow graph for listing 2.13, page 28 with highlighted try-catch sections.

Listing 2.16: Example of Try-Finally

```
try {
        System.out.println("try");
        //return or break or throw exception
}finally{
        System.out.println("finally");

        //CLEAN UP HERE e.g. close database connections
}

output:

try
finally
```

Listing 2.17: Try Finally

```
public static int test(int i) {

        try {
                if(i == 5) return 5;
        }finally{
                i = i + 1;
        }

        return i;

}
```

Listing 2.18: possible bytecode for TryFinally listing 2.17

```
    0:  iload_0          //push i onto stack
    1:  iconst_5         //push 5 onto stack
    2:  if_icmpne 12     //if i != 5 goto 12
    5:  iconst_5         //push 5 onto stack
    6:  istore_1         //pop int from stack, store in local 1
    7:  jsr       21     //push 10 (returnAddress) onto stack, jump to 21
   10:  iload_1          //push int in local 1 onto stack
   11:  ireturn          //pop int from stack and return it

   12:  goto     28

//begin catch
   15:  astore_1         //pop object reference from stack, store in local 1
   16:  jsr       21     //push 19 (returnAddress) onto stack, jump to 21
   19:  aload_1          //push object reference in local 1 onto stack
   20:  athrow           //pop exception from stack and throw it
//end catch

//begin jsr
   21:  astore_2         //pop returnAddress from stack, store in local 2
   22:  iload_0          //push int from local 0 onto stack
   23:  iconst_1         //push 1 onto stack
   24:  iadd             //add i and 1
   25:  istore_0         //pop int from stack and store in local 0
   26:  ret       2      //jump to returnAddress in local 2
//end jsr

   28:  jsr       21     //push 31 (returnAddress) onto stack, jump to 21
   31:  iload_0          //push local 0 onto stack
   32:  ireturn          //pop int from stack and return it
```

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 2.19, page 37 shows Java source which, when compiled with Jikes to Java 1.4 compatible bytecode, results in being rejected by the JVM [85]. 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 2.19, page 37 (Figure 2.20, 41).

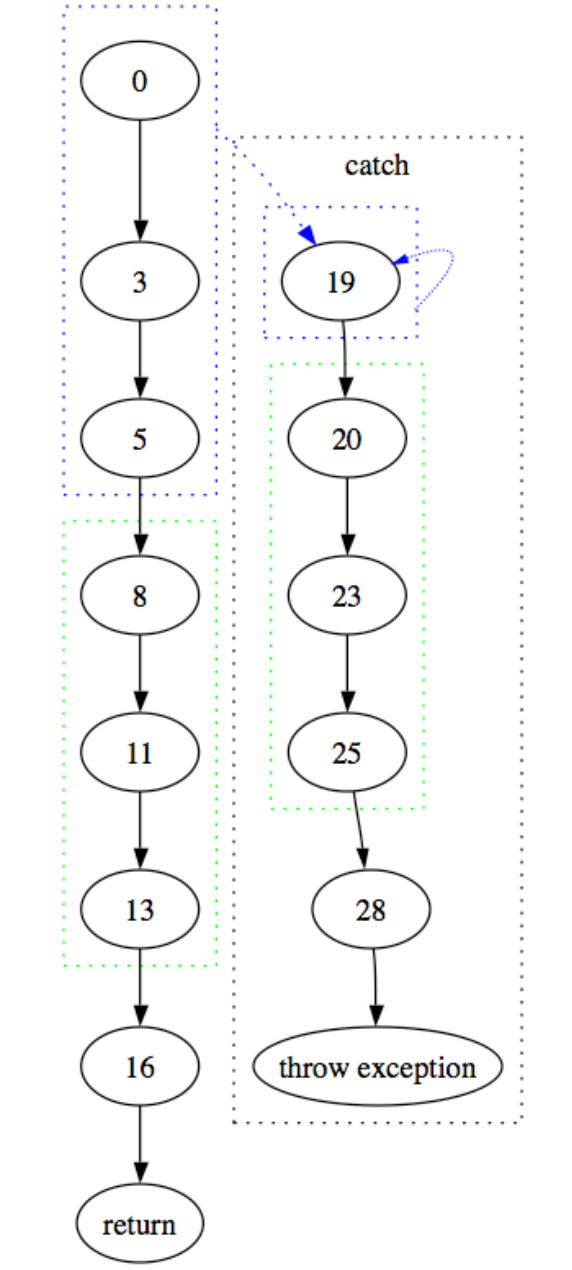

Figure 2.8: Java 1.6 (produced with javac 1.6) bytecode control flow graph for listing 2.16, page 33 with highlighted try-catch-finally sections.

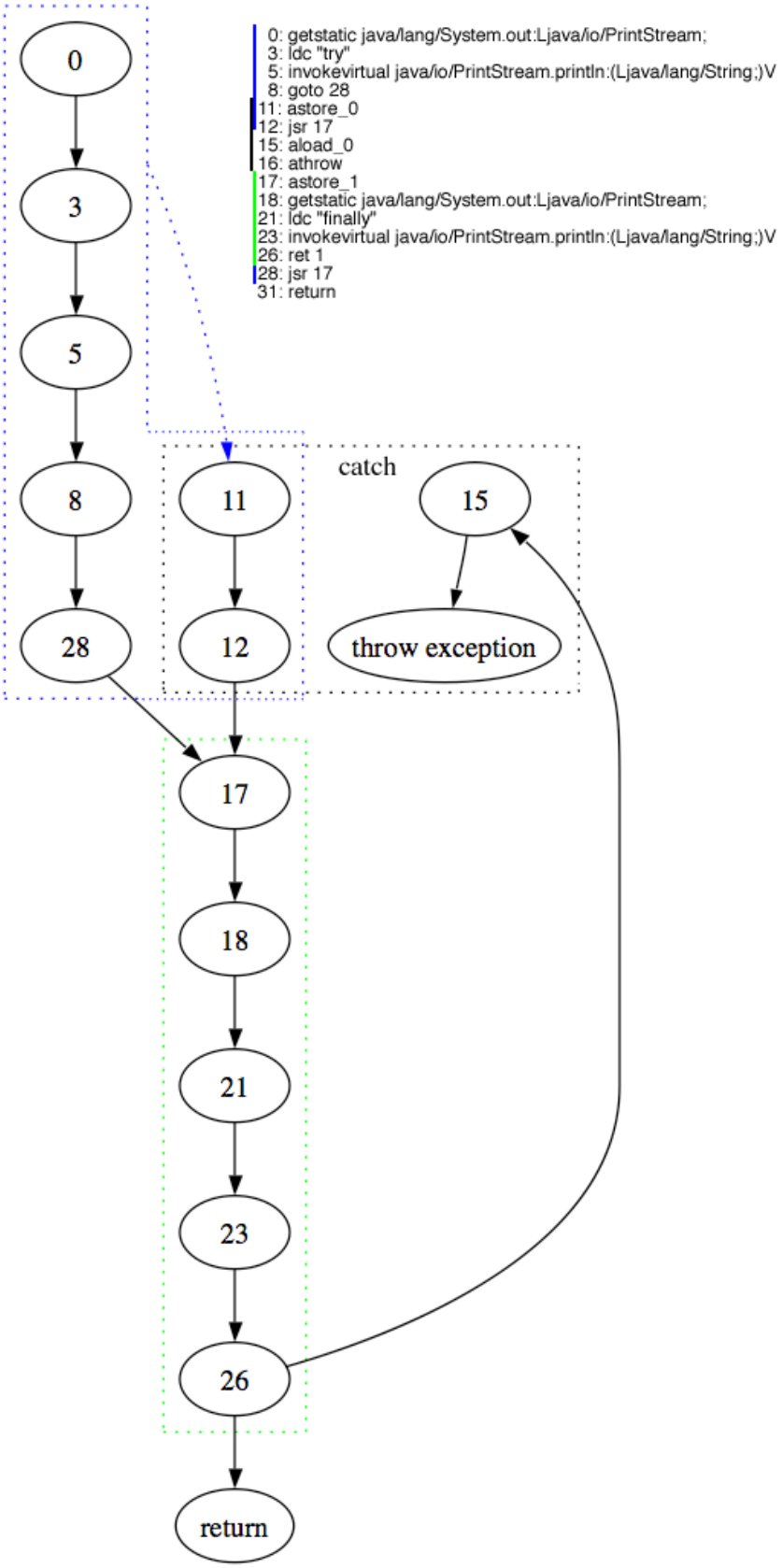

Figure 2.9: Java 1.4 (produced with Jikes) bytecode control flow graph for listing 2.16, page 33 with highlighted try-catch-finally sections.

Listing 2.19: Legal Java source which produces illegal bytecode [85]

```
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;
}
```

4 javac bytecode vs arbitrary bytecode

The design of a compiler is made easier if a decompiler only has to decompile code produced by Sun's *javac* Java compiler as it mostly means inverting a known compilation strategy [71]. *javac* is open-source so developers can see the implementation of the compiler and relatively easily invert it. Of course there are still the problems listed in section 3 to overcome. Many of the available Java decompilers use this strategy (TODO: citation?).

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 [63]. 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 promoted the development of Dava [71] - 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 [41]. 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 entierly independent of the Java language and bytecode is more powerful than the Java language [63]. For example in the Java Virtual Machine specification [65] in relation to exception handling it list 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 **JVM!** (**JVM!**). 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

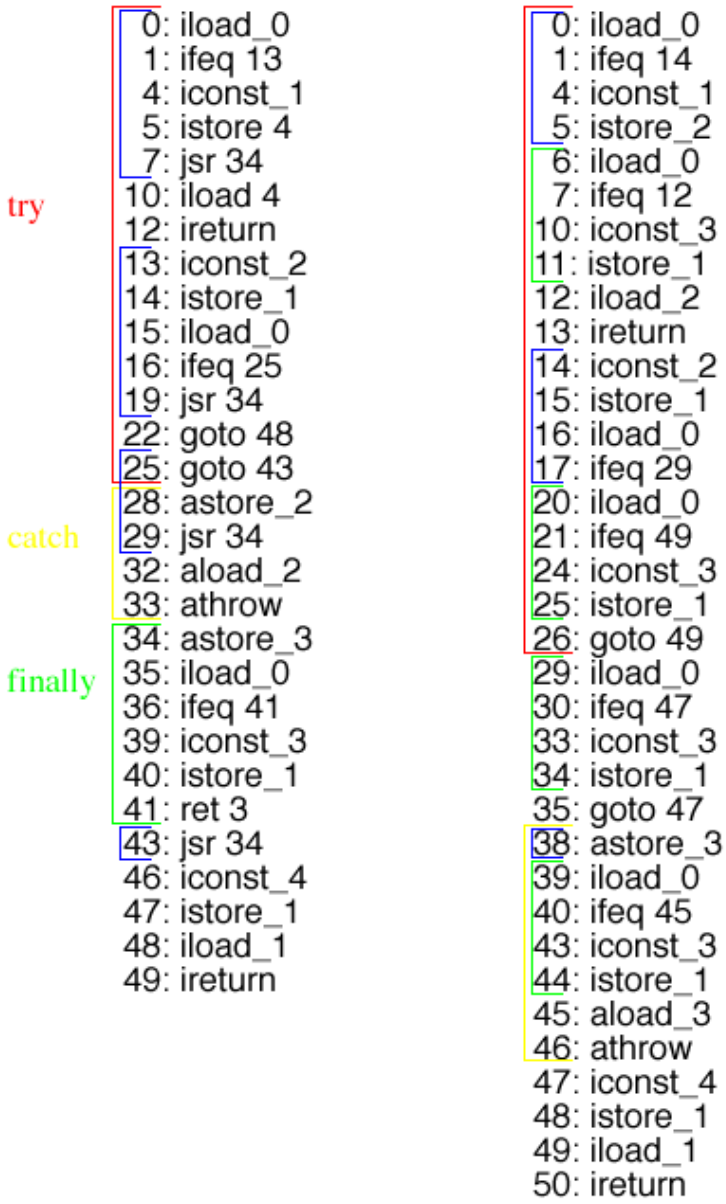

Figure 2.10: Bytecode resulting from compilation of listing 2.19, page 37. Left-hand-side compiled with jikes (Java 1.4), right-hand-side compiled with javac 1.6. Blue sections indicate exception table entries.

in verificable bytecode which doesn't easily translate to syntatically correct Java source code.

5 Conclusion

Listing 2.20: Jad decompiler output for listing 2.19, page 37 (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;
    }
```

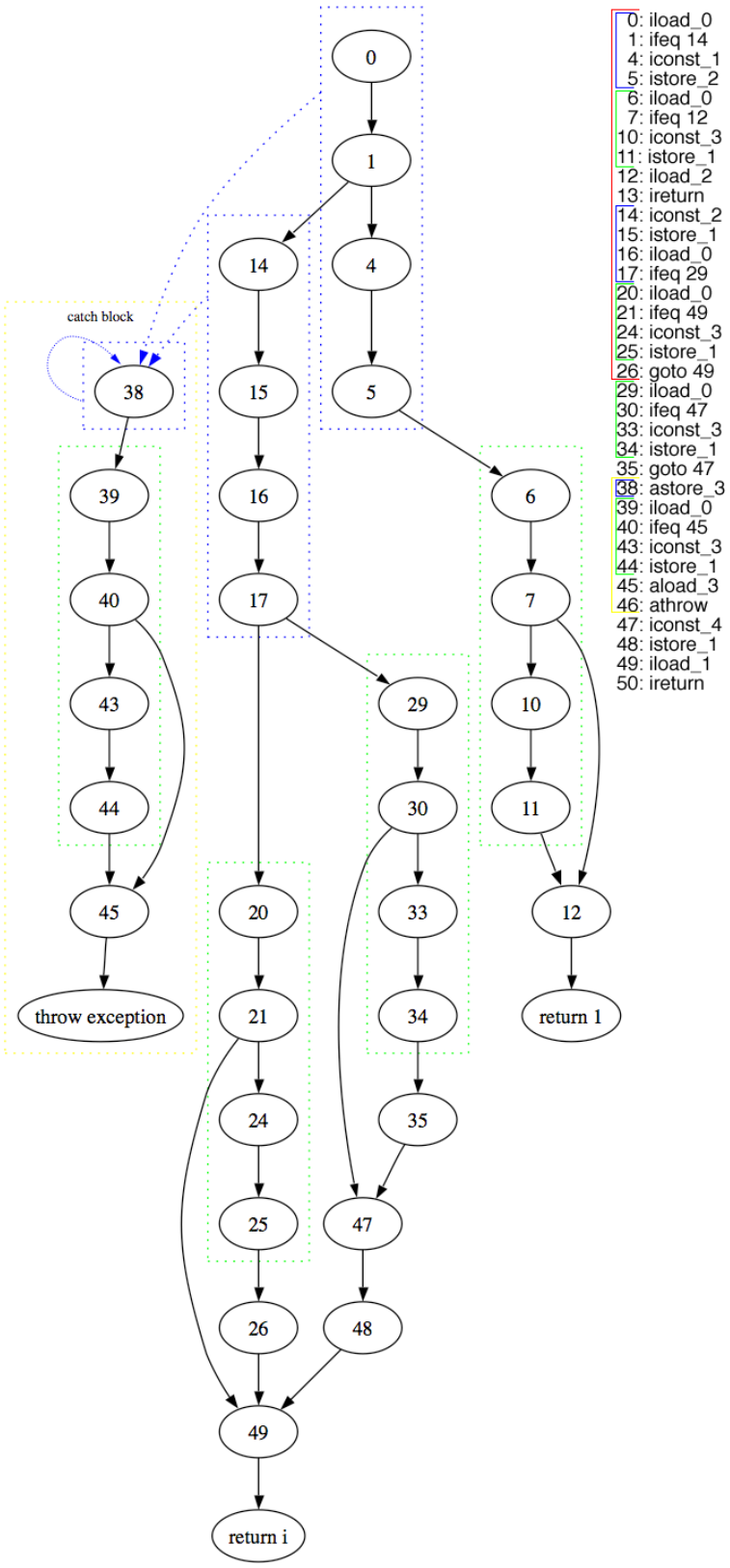

Figure 2.11: Control flow graph for javac 1.6 generated bytecode of listing 2.19, page 37.

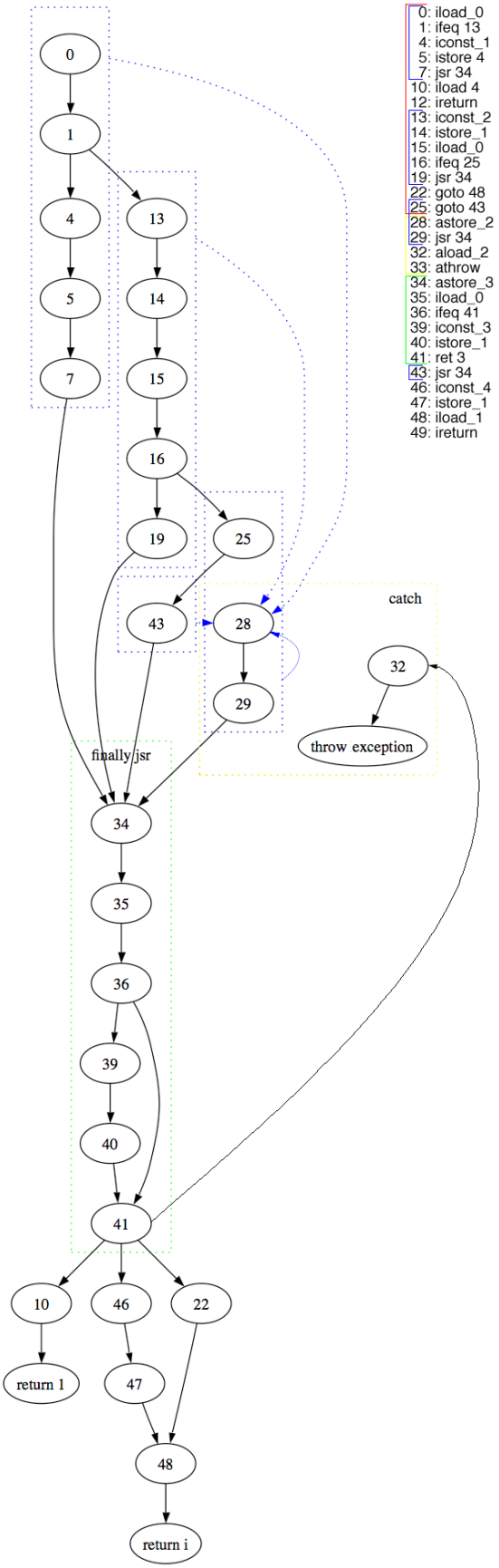

Figure 2.12: Control flow graph for jikes (Java 1.4) generated bytecode of listing 2.19, page 37.

Chapter 3

Current Decompilers

1 Introduction

There are several decompilers for Java bytecode available - commercial, free and open-source. There are many that are unmainted such as Mocha, one of the first Java decompilers. Most Java decompilers are aimed at decompiling bytecode generated by a Java compiler (especially *javac*) and only one decompiler, Dava, aims to decompile arbitrary bytecode. This distinction between two sets of decompilers is important as Java bytecode can be generated or manipulated by software other than a compiler. Even simple changes to a Java class file can cause many decompilers to fail even though the bytecode is syntactically, semantically and verifiably correct.

The portability and security of the Java Virtual Machine makes the platform attractive to developers of other languages as an execution platform. This had led to the development of Java bytecode compilers for languages other than Java. Java bytecode generated in this way is unlikely to generate bytecode in the same way as a Java compiler and causes decompilers to fail. Note that the combination of a language X to bytecode compiler and a bytecode to Java compiler provides a translation system for language X to Java.

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 [42] which tested 9 decompilers with a range of test programs. 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.

2 Measuring the Effectiveness of a Decompiler

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. 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 [24].

Naeem *et al.* [75] suggest using software metrics [52, 56] 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.

For each program, decompiler pair, we give a score between zero and nine (see Figure 3.1). A score of 0 is a perfect, or good, decompilation whereas 9 means that the decompiler failed and no output was produced.

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 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 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.

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 3.1: Decompilation correctness classification

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.

3 The Decompilers

The following decompilers were tested:

Mocha [88], 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. It is not useful nowadays except for simple programs. Mocha is obsolete but is still available on several websites as the original license permitted its free distribution.
SourceTec (Jasmine) [84], 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).
SourceAgain [22] 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.
ClassCracker3 [66] 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.
Jad [59] is a popular decompiler that is free for non-commercial use but is no longer maintained. 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 [59] including an Eclipse IDE plug-in named JadClipse [50].
JODE [53] is an open-source decompiler that also includes a bytecode optimiser. The latest version 1.1.2-pre1 was released February 24, 2004.
jReversePro [62] is an open-source disassembler and decompiler project which is currently at version 1.4.2 though hasn't been updated for several years.
Dava [69, 76, 74, 70, 71] is a decompiler which is part of the Soot Java Optimisation Framework [87] from the Sable Research Group[1] at McGill University in Montreal, Quebec, Canada. Soot is under constant development at the Sable Research Group and the latest release was version 2.3.0 on June 3, 2008.
jdec [26] 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.
Java Decompiler [37] is a free Java decompiler aimed at decompiling Java 5 and above class files. It is in its early stages, at only version 0.2.7, and has been in development for about a year.
NMI Code Viewer was included in the original survey with results 'startingly similar' to Jad [42]. We do not include this (unmaintained) decompilar as, in actual fact, it is a front-end for Jad [59].
jAscii was a commercial decompiler though the company has now gone out of business. In the tests jAscii performed similarily, though slightly better, than SourceTec (Jasmine). jAscii supports Java 2 classfiles and inner classes unlike SourceTec (Jasmine) and is able to decompile the Usa inner class test program correctly. This decompiler is obsolete and we could not obtain jAscii for re-testing.

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

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
Dava	open-source	current	2.0.1	2.3.0	2008
jdec	open-source	current	N/A	2.0	2008
Java Decompiler	free	current	N/A	0.2.7	2008

Figure 3.2: Decompilers: Some of the currently available decompilers have, since the original survey [42] in 2003, been upgraded, become unmaintained or obsolete. There are two new decompilers, jdec and Java Decompiler which have become available since 2003. Many commercial Java bytecode decompilers have become obsolete and the currently maintained decompilers are open-source and/or free. In the original survey JODE was determined to be the best by decompiling 6 out of 9 programs correctly, closely followed by Jad and Dava.

4 Tests

Different Java bytecode decompilation problems are tested using programs taken from different sources which each provide a specific area to test.

The original tests showed that different decompilers are sometimes better in different areas but no decompiler passed all the tests [42]. Of the decompilers tested in the original survey, JODE performed the best by correcctly decompiling 6 out of the 9 test programs, with Dava and Jad close behind.

In our tests we include two types of Java class file: those generated by *javac* and those generated by other tools. Java source files are compiled with *javac* version 1.6.0_10. Arbitrary Java class files include three hand-written using the Jasmin assembler version 2.1, one optimised using the Soot Framework and one compiled by JGNAT. 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. The Java class files will be decompiled using each decompiler and the result will be classified into one of our ten categories of decompiler effectiveness.
The class files considered were:

Fibonacci is a trivial test for a decompiler. It is a fairly simple program to output the Fibonacci number of a given input number.
Casting [77] is a simple program to test if a decompiler can correctly detect the need to cast a *char* to an *int*.
InnerClass [77] is a simple program containing inner classes to test if a decompiler can handle inner classes.
TypeInference [71] is a program which tests a decompiler's ability to perform type inference for local variables. A specific variable in the program is difficult for a decompiler to type because it depends on the value of another variable. The original paper [71], written by the developers of Dava, tested three different decompilers against Dava. They showed that their decompiler could correctly type the variables whereas the other decompilers failed to do so.
TryFinally is a simple test to determine whether decompilers can decompile the implementation of try-finally blocks using inline code instead of Java bytecode subroutines. A lot of the

old decompilers will be expecting the use of subroutines for try-finally blocks but *javac* 1.4.2+ generates inline code for finally blocks rather than using subroutines.
ControlFlow [71] is a program which tests a decompilers handling of control flow.
Exceptions [71] 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 tests is incorrect and Dava, which should be able to decompile the program, exits with a null pointer exception. A re-written version is used in our tests based on the call graph in the original paper [71].
Optimised was generated by using the Soot optimiser on the TypeInference test program. An example of an optimisation that has been performed is the removal of the *dup* opcode (which duplicates the value on the top of the stack) and replacing it with *load* and *store* instructions.
VariableReUse 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*. A compiler would not generate such code that we created using Jasmin. Multiple types for local variables is valid bytecode as long as the lifetimes of the two uses of the local variable does not overlap [57] i.e. a local variable only has the type (and value) of the last variable stored in it.
Ada [44] is an implementation of the game Connect Four originally written in Ada and compiled with JGNAT 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.

Listing 3.1: Fibonaci test program [18]

```
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);
    }

}
```

Listing 3.2: Casting test program

```
from Decompiling Java chapter 1
 public class Casting {
    public static void main(String args[]){
        for(char c=0; c < 128; c++) {
            System.out.println("ascii " + (int)c + " character "+ c);
        }
    }
}
```

Listing 3.3: Inner class test program

```
//from the book Decompiling Java, this time chapter 3
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 3.4: Type 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...
    }
}

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

public class Main {

    public static void f(short i) {
        Circle c; Rectangle r; Drawable d;
        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[])
        { f((short) 11); }
}
```

Listing 3.5: Optimised test program

```
.method public static f(S)V
    .limit stack 3
    .limit locals 2
    iload_0
    bipush 10
    if_icmple label0
    new Rectangle
    astore_1
    aload_1
    iload_0
    iload_0
    invokespecial Rectangle/<init>(SS)V
    aload_1
    invokevirtual Rectangle/isFat()Z
    istore_0
    aload_1
    astore_1
    goto label1
label0:
    new Circle
    astore_1
    aload_1
    iload_0
    invokespecial Circle/<init>(I)V
    aload_1
    invokevirtual Circle/isFat()Z
    istore_0
    aload_1
    astore_1
label1:
    iload_0
    ifne label2
    aload_1
    invokeinterface Drawable/draw()V 1
label2:
    return
.end method

.method public static main([Ljava/lang/String;)V
    .limit stack 1
    .limit locals 1
    bipush 11
    invokestatic Main/f(S)V
    return
.end method
```

Listing 3.6: While-continue test program

```
//from the paper "Decompiling Java Bytecode: Problems, Traps and Pitfalls", Figure 5
public int foo(int i, int j) {
        while (true) {
                try
                {   while (i < j)
                        i = j++/i;
                }
                catch (RuntimeException re)
                {   i = 10;
                continue;
                }
                break;
        }
        return j;
}
```

5 Results

No decompiler was able to decompile all test programs with JODE decompiling the most programs correctly. JODE only managed to decompile 6 programs while four (unmaintained) decompilers could not decompile any of the test programs correctly. The top three decompilers were JODE, jad and Java Decompiler whereas the worst decompilers were the unmaintained commercial decompilers - SourceTec, ClassCracker3 and Mocha. Some decompilers failed simple because they could not parse the latest class files or arbitrary class files.

Dava, unsurprisingly, was better at decompiling the arbitrary bytecode test programs than the *javac* test programs. Dava is the third best decompiler based on our effectiveness measures, but is the best arbitrary bytecode decompiler. Dava performs similarly to jdec with *javac* generated bytecode, though jdec decompiles 2 of these programs correctly compared to 1 for Dava. 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 performing slightly better at decompiling the arbitrary bytecode programs. JODE was able to decompile one more program correctly than Java Decompiler though. JODE was the best decompiler in the original survey and is still one of the best in our evalution but Java Decompiler beats JODE with our effectiveness measures and only decompiles one less program correctyly than JODE.

Mocha

Mocha [88], released as a beta version in 1996, was one of the first decompilers available for Java along with a companion obfuscator named Crema. Unfortunately the author, Hanpeter van Vliet, died of cancer and the program remained in beta. Mocha can only decompile earlier versions of Java as it is an old program. It is not useful nowadays except for simple programs and has been excluded from testing. Mocha is obsolete but is still available on several websites as the original license permitted its free distribution.

SourceTec (Jasmine)

SourceTec, 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

Listing 3.7: Exception test program [17]

```
.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
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
```

Listing 3.8: Try-finally test program

```
public class TryFinally {

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

}
```

Listing 3.9: Args 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
```

		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	0	5	5	0	5	9	5	9
	Usa	8	8	0	0	8	0	8	9	0	9
	Sable	8	2	1	1	7	0	9	9	1	9
	TryFinally	8	7	5	0	0	7	7	9	7	9
arbitrary	ControlFlow	8	1	4	1	7	1	9	9	8	9
	Exceptions	8	0	4	7	7	9	8	9	7	8
	Optimised	8	2	4	4	5	2	6	9	3	9
	Args	8	1	2	2	3	0	5	4	0	4
	connectfour	8	3	8	5	8	9	8	4	3	4

Figure 3.3: Decompiler Test Results: Each decompiler was tested in different problem areas, with 5 test programs representing *javac* generated bytecode and 5 representing *arbitrary* bytecode. The results are given using our effectiveness measurement scale with 0 being a perfect decompilation and 9 being the case in which a decompiler fails. No decompiler was able to correctly decompile all our test programs with JODE correctly decompiling the most correctly.

by SourceTec (Jasmine). Emmerik included this decompiler in his tests and concluded that it is only useful for very simple Java 1.1 programs as it failed all the tests. SourceTec (Jasmine) fails to parse Java 1.6 files therefore fails many of our tests for this reason. This confirms the original test results which indicate that the decompiler is obsolete and not useful for most decompilation tasks.

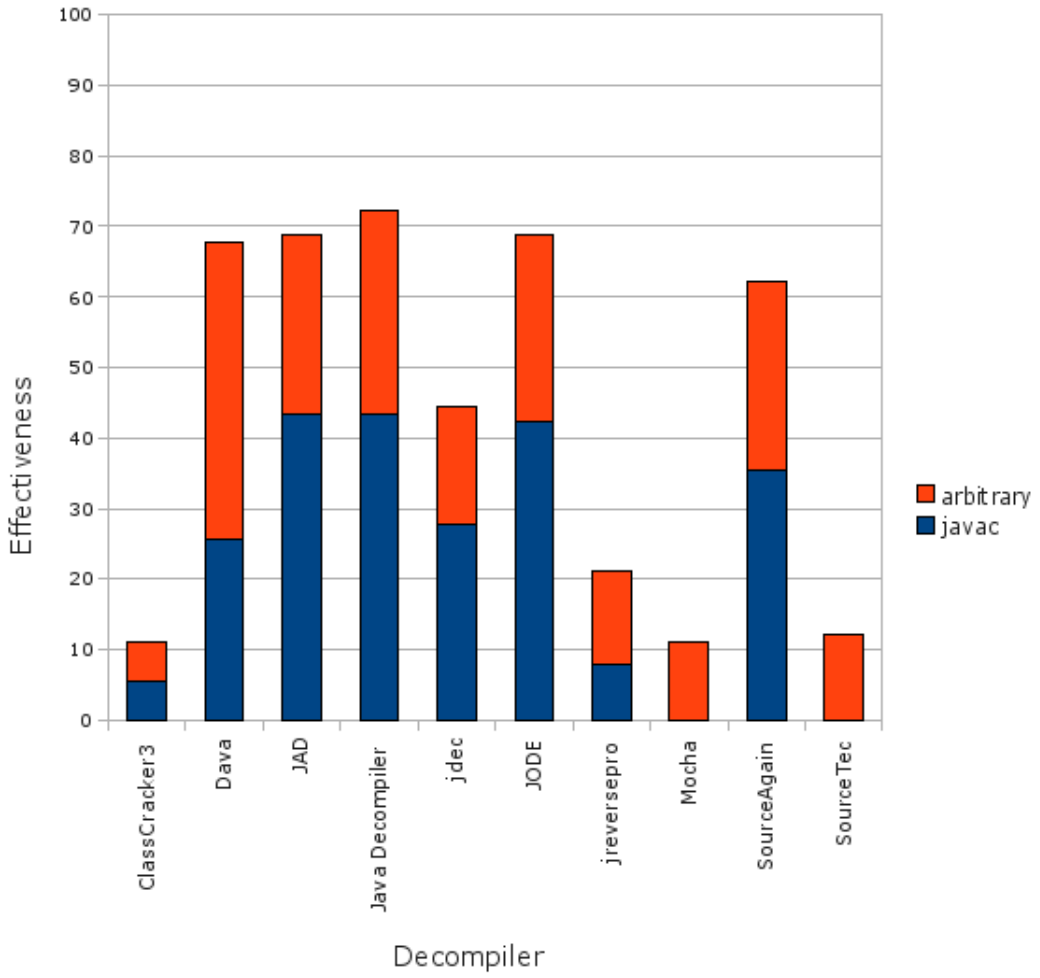

Figure 3.4: Decompiler Effectiveness

SourceAgain

Emmerik used the professional version of the decompiler for his tests and found it could decompile the Fibonacci program as well as the Usa inner classes program but failed the other tests.

The SourceAgain web version correctly decompiles the Args test program but incorrectly decompiles the TryFinally program (Listing 3.10, page 59).

ClassCracker3

Tests were performed on the evaluation version of the program which is available at the Mayon Software Research's website[2]. We attempted to decompile the Args and TryFinally programs with the result that the class files did not fully decompile. We were unable to assertain whether this was due to the software being a trial version. The ClassCracker 3 website suggests that the first five methods of a class should decompile though only the method signature appears in the decompiler output.

[2] `http://mayon.actewagl.net.au/`

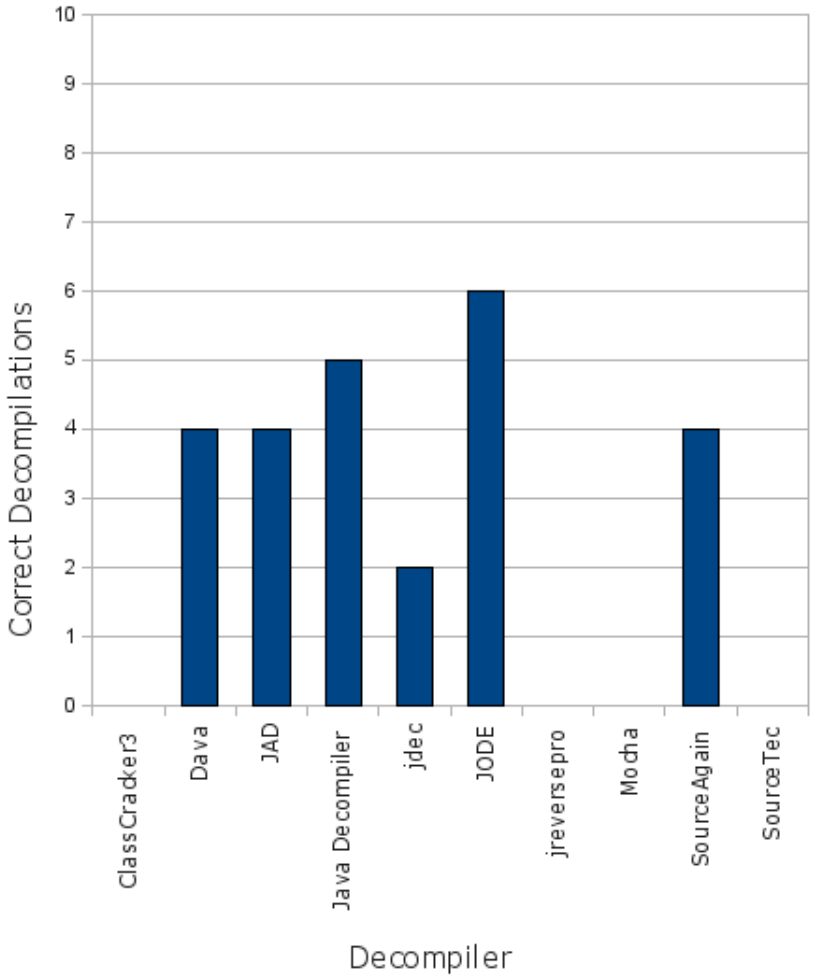

Figure 3.5: Correct Decompilations

JAD

It passes the first few tests but has trouble with exceptions, control flow, optimised code and performs no type inference.

Jad incorrectly decompiles the TryFinally test program as it places *System.out.println("test");* outside of the try section (Listing 3.11, page 59).

The Args test program also decompiles incorrectly, using Jad, with an attempted assignment of the integer 0 to the *args* variable which is of type *String[]* (Listing 3.12, page 59).

Listing 3.10: SourceAgain TryFinally decompiled output

```
public class TryFinally {

    public static void main(String[] as)
    {
        try
        {
            System.out.println( "try" );
        }
        finally
        {
            try
            {
            }
            finally
            {
                System.out.println( "finally" );
                throw obj;
            }
        }
        System.out.println( "finally" );
    }
}
```

Listing 3.11: Jad TryFinally decompiled output

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

Listing 3.12: Jad Args decompiled output

```
class Args {
    public static void main(String args[]) {
        args = 0;
        System.out.println(args);
    }
}
```

JODE

Re-running the tests on version 1.1.2-pre1 produced that same results as Emmerik's tests. Emmerik found that JODE completed most tests with few errors including the type inference in the Sable test program. JODE had programs with intersecting exception handlers and optimised code.

JODE incorrectly decompiles the TryFinally test program as a try-catch block (Listing 3.13, page 60). JODE correctly decompiles the Args test program.

Listing 3.13: JODE TryFinally decompiled output

```
public class TryFinally {
    public static void main(String[] strings) {
        try {
            System.out.println("try");
        }catch(Object object) {
            System.out.println("finally");
            throw object;
        }
        System.out.println("finally");
    }
}
```

jReversePro

Emmerik test version 1.4.1 and found that it failed all his tests including the trivial Fibo program which most other decompilers correctly decompiled. Surprisingly jReversePro performs type inference and correctly typed the variable *d* in the Sable test program (Listing 3.4, page 51), unlike many other decompilers which typed *d* as *Object* and insert typecasts as necessary. jReversePro does not support Java 1.6 class files and fails to parse the test programs generated by *javac*. A small change in the source-code allowed Java 1.6 class files to be accepted though it does not parse some of the test programs correctly. It does parse the class files generated by Jasmin as the default class file version is 45.3 but fails to decompile the programs correctly.

Dava

Emmerik used version 2.1.0 in his tests and found the output generated by Dava, though mostly correct, hard to read program which is a problem which should have been solved by the back-end which restructures code to be more 'programmer-friendly' [76, 74].

Dava failed the inner class test program and casting test programs with both resulting in semantically incorrect code. In Emmerik's test Dava failed to parse the Exceptions program though this is due to a incorrectly implemented program. The Exceptions program was based on a control flow graph and description of a test in a paper [71] in which Dava was the only decompiler to correctly decompile the Exceptions program.

The Dava decompiler could not decompile the TryFinally program (Listing 3.8, page 55). The resulting program (Listing 3.14, page 61) does not compile due to the variable *$r4* being declared twice. Dava did not decompile the program as a try-finally block and has instead placed and *while* loop inside the catch section. The problem occurs as Dava expects the try-finally block to be implemented with a Java subroutine, whereas *javac* 1.4.2+ implements finally clauses by producing the finally block code inline. As Dava aims at decompiling arbitrary class files it does not look for specific patterns such as try-finally blocks, and therefore does not detect that this program contains a try-finally block.

The latest version of Dava still fails the Usa inner classes test program (Listing 3.3, page 50) and the casting program (Listing 3.2, page 50). It correctly decompiles the new exceptions program (Listing 3.7, page 54).

Dava correctly decompiles the Args test program though it types the local variable as *byte* instead of *int*.

Listing 3.14: Dava TryFinally decompiled output

```
public class TryFinally {

    public static void main(String[] r0) throws java.lang.Throwable {

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

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

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

jdec

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

The following results were obtained after running tests on the jdec decompiler:

Fibo The program (Listing 3.15, page 63) is semantically correct and produces the desired output though it is not as tidy as the original source.

Casting The result (Listing 3.16, page 64) looks very different to the original source and the *int* cast has not been included. The decompiled code uses a while loop instead of a for-loop and a StringBuffer instead of a String. This is due to *javac* converting string concatenation to use a StringBuffer as it is more efficient.

The result is semantically incorrect as the cast is omitted.

Usa The decompiler could not decompile the inner classes at all with only the outer class in the output Java source (Listing 3.17, page 64).

Sable The decompiler could not decompile the Sable program correctly. The result (Listing 3.18, page 65) shows incorrectly type variables and but also variables used in the wrong places - for example $aCircle2$ is used but not declared that should be the variable d from the original program. The resulting program does not even compile due to the variable $aCircle2$ not being declared. jdec also types the boolean in the program as an int.

Optimised The optimisations caused jdec some trouble in decompilation. The resulting program (Listing 3.19, page 66) contains assignment operations within the constructors. For example the assignment highlighted in the code below should appear on the line after but it has been placed within the constructor.

Circle JdecGenerated28 = new Circle(**aCircle1 = JdecGenerated28;** short2);

This is not surprising as jdec is aimed at decompiling *javac* generated code.

ControlFlow The results for the ControlFlow program (Listing 3.20, page 67) do not compile as the decompiler has placed the catch declaration intersecting the if-else block.

Exceptions jdec could not decompile the Exceptions program with the resulting program (Listing 3.21, 67) missing out certain try blocks. The blocks d, e and f are missing and the program won't compile because on try block does not have an associated catch block.

TryFinally jdec correctly decompiles the TryFinally program.

Args jdec incorrectly decompiles the Args test program. The resulting program (Listing 3.22, page 68) not only tries to assign 0 to variable of type *String[]* but also re-declares the method argument variable.

jdec only passed one test and failed the other tests producing code which is semantically and/or syntactically incorrect. jdec is aimed at decompiling code produced by *javac* so it is not suprising that it could not decompile the optimised program or the exceptions program as these used patterns which *javac* would not have produced. However it is disspointing that the other programs could not be decompiled either, which were generated using *javac*. jdec development

seems to be in its early stages, even though it is version 2.0, but there has been no release since May 2008. jdec could decompile the *javac* 1.6 generated TryFinally program something which other decompilers, such as Dava, could not.

Listing 3.15: jdec Fibo decompiled output

```
class Fibo {

//  CLASS: Fibo:
 Fibo() {
    super();
    return;
 }

//  CLASS: Fibo:
 private static int fib(  int int3) {
    if(int3 > 1) {
      return (fib((int3- 1)) + fib((int3- 2)));
    }
    return int3;
 }

//  CLASS: Fibo:
 public static void main(java.lang.String[] aString1) throws Exception {
    int int4= 0;
    int int5= 0;
    int4=0;
    try {
      int4=Integer.parseInt(aString1[0]);
    }catch(Exception  aException1) {
      System.out.println("Input error");
      System.exit(1);
    }
    int5=fib(int4);
    StringBuffer JdecGenerated35 = new StringBuffer();
    System.out.println(JdecGenerated35.append("fibonacci(").append(int4).append(") = ").
    return;
 }
}
```

Listing 3.16: jdec Casting decompiled output

```
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("ascii ").append(char1)
                           .append(" character").append(char1).toString());
        char1=(char)((char1 + 1));
        continue;
      }
    }
    return;
 }
}
```

Listing 3.17: jdec Usa inner classes program decompiled output

```
public class Usa {

 /***
 ** Class Fields
 ***/
 public String name ;

// CLASS: Usa:
 public Usa() {
    super();
    this.name ="Detroit";
    return;
 }
}
```

Listing 3.18: jdec Sable program decompiled output

```
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;
 }
}
```

Java Decompiler - Yet Another Fast Java Decompiler

Java Decompiler is a newer decompiler which outperforms Jad and Dava overall. The reason which Java Decompiler out performs Dava is that it is correctly able to decompile the TryFinally program which is not something that Dava is aimed at doing. Java Decompiler also has trouble decompiling arbitrary bytecode which is where Dava does better.
The following results were obtained after running tests on Java Decompiler:

Fibo The Fibonacci program correctly decompiled with tidy presented source code (Listing 3.23, page 69).

Casting The decompiled Casting program (Listing 3.24, page 69) is contains a syntax error and does not include the *int* cast. This compiler does correctly type the for-loop variable as a *char* but incorrectly attempts to inititalise the variable with a single quote.

Listing 3.19: jdec Optimised Sable program decompiled output

```
public class Optimised {

//  CLASS: Optimised:
 public static void f(short short2) {
    Rectangle aRectangle1= null;

    short short2= 0;
    if(short2 > 10) {
      Rectangle JdecGenerated8 =
        new Rectangle(aRectangle1 = JdecGenerated8; short2,short2);
      short2=aRectangle1.isFat();
      aRectangle1 =aRectangle1;
    }else{
      Circle JdecGenerated28 =
                new Circle(aCircle1 = JdecGenerated28; short2);
      short2=aCircle1.isFat();
      aCircle1 =aCircle1;
    }

    if(short2==0) {
      aCircle1.draw();
      return;
    }
 }

//  CLASS: Optimised:
 public static void main(String[] aString1) {
    f((short)11);
    return;
 }
}
```

Usa The decopmiled Usa program (Listing 3.25, page 70) is correct, with the inner classes reproduced correctly.

Sable jdec decompiled the Sable test program correctly (Listing 3.26, page 70) but did not correctly type *d* as *Drawable*. No type inference was performed and a type cast was inserted at the *draw*() method call.

Optimised jdec incorrectly decompiled (Listing 3.27, page 71) the optimised Sable test program.

Firstly the boolean variable has not been declared and the method argument is used in the if statement instead of a separate boolean variable. The Rectangle and Circle object initiliasation is spread over two lines and the result includes a called to the method *¡init¿()*. The only local variable is assigned to itself in the last line in each of the if and else sections. There is only one local variable and declared where there were four in the original source - this is due to the re-use of local variable slots in the bytecode.

ControlFlow TODO: this maybe correct? (Listing 3.28, page 71).

Exceptions The decompiled Exceptions program (Listing 3.29, page 72) is incorrect. The

Listing 3.20: jdec ControlFlow program decompiled output

```
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 3.21: jdec Exceptions program decompiled output

```
 public void Exceptions2() {
    System.out.println("a");
    try {
      System.out.println("b");
      try {
        System.out.println("c");
        return;
      }catch(RuntimeException  this) {
        System.out.println("g");
      }
    }
 }
```

Listing 3.22: jdec Args program decompiled output

```
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;
 }


}
```

resulting program does not represent the correct control flow of the original control flow graph. The program also uses a variable named *this* which is a reserved word in Java.

TryFinally The TryFinally program decompiles correctly (Listing 3.30, page 72).

Args The Args program decompiles incorrectly (Listing 3.31, page 72) as it tries to assign the value 0 to method argument which is of type *String[]*.

Java Decompiler is able to decompile only simple programs such as the Fibo program but could not decompile the other programs well. Many contained semantic and syntactic errors. Java Decompiler, like jdec, could decompile the *javac* 1.6 generated TryFinally program something which other decompilers, such as Dava, could not. Java Decompiler seems similar to jdec in the quality of decompilation though their version numbers seem to suggest that jdec is more mature - with jdec at version 2.0 and Java Decompiler at version 0.2.7.

Java Decompiler, like jdec, seems aimed at decompiling *javac* generated bytecode rather than arbtrary bytecode like Dava. Java Decompiler passed the inner classes test unlike some of the other decompilers like jdec, and Dava.

Being at an early stage in development it is hard to tell how good the decompiler could be in the future but it is already comparable jdec and some of the decompilers in Emmerik's tests. Unfortunately Java Decompiler is not open-source so we must rely on the current developer to improve the software.

Listing 3.23: jd Fibo program decompiled output

```
class Fibojd {
  private static int fib(int paramInt) {
    if (paramInt > 1)
      return (fib(paramInt - 1) + fib(paramInt - 2));
    return paramInt;
  }

  public static void main(String[] paramArrayOfString) throws Exception {
    int i = 0;
    try {
      i = Integer.parseInt(paramArrayOfString[0]);
    }catch(Exception localException) {
      System.out.println("Input error");
      System.exit(1);
    }
    int j = fib(i);
    System.out.println("fibonacci(" + i + ") = " + j);
  }
}
```

Listing 3.24: jd Casting program decompiled output

```
public class Casting {
  public static void main(String[] paramArrayOfString) {
    for (char c = '; c < 128; c = (char)(c + '\1'))
      System.out.println("ascii " + c + " character " + c);
  }
}
```

6 Conclusion

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 [43], 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 [69, 76, 74, 70, 71] decompiler, part of the Soot Optimisation Framework [87], 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.

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

Listing 3.25: jd Usa program decompiled output

```
public class Usa {
  public String name;

  public Usa() { this.name = "Detroit"; }

  public class England {
    public String name;

    public England() { this.name = "London"; }

    public class Ireland {
        public String name;

        public Ireland() { this.name = "Dublin"; }

        public void print_names() {
           System.out.println(this.name);
        }
    }
  }
}
```

Listing 3.26: jd Sable program decompiled output

```
public class Sable {
  public static void f(short paramShort) {
    Object localObject;
    boolean bool;
    if(paramShort > 10) {
      Rectangle localRectangle = new Rectangle(paramShort, paramShort);
      bool = localRectangle.isFat();
      localObject = localRectangle;
    }else{
      Circle localCircle = new Circle(paramShort);
      bool = localCircle.isFat();
      localObject = localCircle;
    }
    if (!(bool)) ((Drawable)localObject).draw();
  }

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

Listing 3.27: jd Optimised Sable program decompiled output

```
public class Optimised {

  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)
      ((Drawable)localObject).draw();
  }

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

Listing 3.28: jd ControlFlow program decompiled output

```
public class ControlFlow {
  public static int foo(int paramInt1, int paramInt2) {
    try {
      while (paramInt1 < paramInt2)
        paramInt1 = paramInt2++ / paramInt1;
    }catch(RuntimeException localRuntimeException) {
      while (true) {
        paramInt1 = 10;
      }
    }
    return paramInt2;
  }
}
```

Listing 3.29: jd Exceptions program decompiled output

```
  public void Exceptions2() {
    System.out.println("a");
    try {
      System.out.println("b");
    }catch (java.lang.RuntimeException this) {
      try {
        System.out.println("c");
        System.out.println("d");
        System.out.println("f");
        return;
      }catch (java.lang.Exception this) {
        while (true)
          System.out.println("e");
        this = this;
        System.out.println("g");
      }
    }
  }
```

Listing 3.30: jd TryFinally program decompiled output

```
public class TryFinally {
  public static void main(String[] paramArrayOfString) {
    try {
      System.out.println("try");
    }finally{
      System.out.println("finally");
    }
  }
}
```

Listing 3.31: jd Args program decompiled output

```
class Args
{
  public static void main(String[] paramArrayOfString)
  {
    paramArrayOfString = 0;
    System.out.println(paramArrayOfString);
  }
}
```

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. [25, 70, 57]) and machine code (e.g. [73]). 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 [46], 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 [25].

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. Of these, JODE is the best decompiler as it correctly decompiles 6 out of the 10 test programs correctly. Unfortunately Jad is unmaintained and so is not guaranteed to work for future versions of Java class files. The commercial decompiler SourceAgain falls slightly behind in the effectiveness measures, but was able to decompile 4 programs correctly - the same number as Dava and Jad. SourceAgain performed similarly to Jad and Dava but is now obsolete and only available as a web application which can decompile single class files. Java Decompiler is a newer decompiler in active development, which performs highest in our effectiveness measures and correctly decompiles 5 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.

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.

In terms of our evaluation, Dava and JODE are two of the best decompilers, with Dava being the best arbitrary bytecode decompiler and JODE being the best *javac* oriented bytecode decompiler. Dava faces some challenges in decompilation of Java specific code while JODE has problems with arbitrary bytecode. Future work will investigate the possibility of combining the desirable features of these to produce a single decompiler capable of decompilation of both Java specific features (such as try-finally blocks) and arbitrary bytecode.

There are two popular books on decompiling Java: 'Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering' [55] and 'Decompiling Java' [77]. The former is very focused on using tools to perform decompilation, with Jad as an example, and less focused on the theory and implementation of a decompiler. They demonstrate Jad as an excellent decompiler and dismiss decompilation problems suggesting they only occuring due to obfuscation.

‘Decompiling Java’ demonstrates the ideas of Java bytecode, decompilation, obfuscation and shows the begining of an implementation of a simple decompiler.

Chapter 4

Protection for Java Bytecode

If applications are to be distributed as Java class files we should be able to protect the intellectual property contained within the bytecode (such as proprietary algorithms) from reverse-engineering. An adversery (such as a software pirate or competing company) having access to the relatively easy to decompile Java class files presents a greater risk of them reverse-engineering the application than if it were distributed as machine code binaries.

There are two different ways in which we could protect intellectual property stored within class files. Either we hide the information or we prevent the user from decompiling the class file (or both).

1 Native Code

Java code can call native methods which means that the important sections in a class file could be written using a language such as C. The benefit is that the native code is difficult to decompile but the downside is that the code is no longer portable - losing one of attributes that makes Java popular.

2 Encryption

One technique used by several commercial products to encrypt Java class files, thereby hiding intellectual property from adversaries. ChainKey's Java Code Protector [7] uses this technique and suggests, in their marketing material, that Java class files are impenetrable after using their product.

However, it is trivial to access the unencrypted Java class files. A Java virtual machine, which implements the Java Virtual Machine Specification, only accepts class files adhering to the specification. A Java virtual machine therefore does not understand encrypted class files and so class files must be decrypted before they are sent to the virtual machine. It is a trivial task to re-write method *java.lang.ClassLoader.defineClass()* in the Java runtime that is the entry point for all class files. Class files pass to this method must adhere to the Java Virtual Machine Specification and therefore must be *un-encrypted*. Code can be added to the class loader method to output all class files that it recieves.

Encryption is therefore a fundamentally flawed approach to protecting intellectual property and as such it is advised not to use such techniques to protect Java class files from adversaries [35].

3 Watermarking

The watermarking process is designed to limit distribution of illegal software by allowing the software owner to embed a hidden watermark into their program. Watermarking will be discussed in detail in chapter 5.

4 Code Obfuscation

To obfuscate a program is to perform a semantics preserving transformation with three conditions: functionality must be maintained, the obfuscated code must be efficient and the code must be sufficiently difficult to understand if reverse-engineered [27]. Code obfuscation will be discussed in detail in chapter 6.

Chapter 5

Watermarking

1 Introduction

The watermarking process is designed to limit distribution of illegal software by allowing the software owner to embed a hidden watermark into their program. This hidden watermark can be used, at a later date, to prove that they are the owner of software that has been stolen. 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 neccesary that the watermark is resillent to semantics preserving transformations (such as optimizations or obfuscations).

2 Survey of Techniques

2.1 Add Expression

The Add Expression algorithm embeds a integer watermark into a random method in a Java class file. The algorithm splits the integer watermark into two (a and b) and, in a randomly chosen method, creates a new local variable which the sum of a and b is assigned to.

The algorithm relies on the use of a named variable within the class file to locate the watermark (local variable name will begin with $sm\$$). Local variable names can be stored in a method's local variable table for debugging purposes but they are not neccessary, as local variables do not require names in Java bytecode. Therefore by removing local variable tables the recognition algorithm no longer works.

Assuming we no longer rely on a name stored within the class file itself, but use an external key file instead, the local variable used to store the watermark is not used for anything else. Therefore simple static analysis removes the watermark from the class file (something that a bytecode optimiser will do as part of dead code elimination).

2.2 Monden

Monden *et al.* [72] suggest a technique whereby a dummy method within a class file is transformed to encode the watermark.

3 Embedding a Unique ID

4 Fingerprinting

Chapter 6

Code Obfuscation

The relative ease of decompilation of Java class files increases the risk of exposing intellectual property such as proprietary algorithms. A company must protect is intellectual property but by distributing software as Java class files the risk of exposing implementation details is far greater than if the software is distributed as machine code executables. Java brings many advantages, such as portability and verifiable class files, but also brings with it the possibility of exposing the original source code of the application via decompilation.

To obfuscate a program is to perform a semantics preserving transformation with three conditions: functionality must be maintained, the obfuscated code must be efficient and the code must be sufficiently difficult to understand if reverse-engineered [27]. Obfuscating transformations can be applied to Java source or Java bytecode.

Techniques to secure the contain of Java class files via obfuscation have been researched since Java was first developed. One of the first Java decompilers, Mocha, had a companion obfuscator named Crema. There are now several commercial (e.g. Zelix Class Master) and open-source obfuscators for Java byte code.

There are two ways in which obfuscating Java bytecode can reduce the possibility of intellectual property falling into the hands of an attacker. Firstly obfuscating Java bytecode in such ways that it produces unexpected bytecode sequences will increase the dificulty of decompilation. Dava is aimed at decompiling such arbitrary code whereas most other decompilers fail. Secondly, if a program can be decompiled correctly, is to increase the difficulty of understand the decompiled code. The harder a program is to understand the longer it will take to produce a competing product based on the proprietary algorithms. This will either deter an attacker or will simply to so long that by the time an attacker has understood the code it is out-of-date.

Much research has gone into obfuscating programs to decrease understanding and there are many free and commercial obfuscators which implement such techniques. A side effect of some semantics preserving transformations applied to bytecode is that some decompilers, which are aimed at decompiling *javac* generated bytecode, fail when encountering unexpected bytecode sequences.

1 Techniques

1.1 Identifier Renaming

Well written source code is self-documenting [78] as the meaning of methods, variables and classes can be given a name which describes their purpose. Replacing meaningful identifier names with random, meaningless ones results in a less readable program. Identifier renaming is one of the most common obfuscations and is simple to implement. Identifier renaming can

be applied to Java source code or Java bytecode, the latter being easy to implement as it can be as simple as renaming entries in the constant pool.

Identifier renaming is a one-way transformation as the original names of the identifiers cannot be recovered but some techniques have been suggested to intelligently generate meaningful names based on types and uses of the identifiers [32].

Chan *et al* suggest using illegal identifiers, such as Java keywords, by modifying Java class files. The motivation behind the use of illegal identifiers is to make the class files un-decompilable and they show that several decompilers fail or produce illegal Java source. Contratry to Chan *et al*'s suggestions it is trivial to deobfuscate these kind of obfuscations as it merely involves renaming the illegal identifiers using legal names (allowing decompilation to legal Java source).

2 How Not To Obfuscate

Obfuscating techniques can be useful in increasing the difficulty of understanding a decompiled program but we must be careful to obfuscate every part of the program. This section is based on a trial version of an application distributed as a jar file which requires a code to activate the full version.

Suppose we have an application which displays a dialog box where the user enters a username and a code which they recieve from the vendor by an algorithm based on the username. The username is checked to see if it matches the code in the application in a highly obfuscated class which is hard to decompile and even harder to understand. Other classes of the application are also obfuscated so it is not easy to tell which class deals with registration checking.

How can we crack this program?

Step 1 We use *grep* to search for instances of the word 'Registered'. Surprisingly we find that several class files including Main.class match the search.

Step 2 We decompile Main.class, using Jad, though it doesn't matter if the class decompiles correctly as we only want to find the where the word 'Registered' is used.

Step 3 We find in Main.jad uses of a variable named *$isRegistered*. We look for an assignment to *isRegistered* which calls a method abc in an obfuscated class XYZ. XYZ is the registration checking class. Method abc accepts to Strings - username and code - and returns a boolean.

Step 4 We replace the method call with the value *true* but this doesn't work as the registration checking code is probably used in other places in the code. We decide it will be easier if method abc always returned true.

Step 5 Unfortunately returning true will not always work as the method call is of the form

> isRegistered = XYZ.abc(username, "9" + code) ? false : XYZ.abc(username, code);

We therefore, assuming all calls to XYZ.abc in other classes are the same, need to replace the method code of XYZ.abc with bytecode instructions which return false if code begins with 9 and true otherwise.

Step 6 We use a bytecode editor to edit XYZ.class and replace the code attribute with new bytecode as shown in listing 6.1, page 81.

Step 7 Re-create the jar file and execute the program to use the full version of the software.

Leaving a variable named *$isRegistered* is like holding up a big sign saying 'crack me here'. It doesn't matter how obfuscated XYZ.class is if there is a plaintext variable name which shows us exactly which method we're interested in and how it should work. In order to fully protect a program all variables must be obfuscated not just variables in the seemingly important classes.

It may also be a good idea to encode string constants and replace all string constants in a program with an encoded string and method call to a decoder - this is something that Zelix Class Master does (see TODO).

Listing 6.1: bytecode representing which returns 0 if string begins with a, otherwise returns 1

```
    aload_0
    invokevirtual java/lang/String/length()I
    ifle a
    aload_0
    iconst_0
    invokevirtual java/lang/String/charAt(I)C
    bipush 57
    if_icmpne a
    iconst_0
    goto b
a:
    iconst_1
b:
    ireturn
```

Appendix A

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 A.2, page 85 shows a hex dump of the Java class file for the hello world program (Listing A.1, page 84).

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 [19].

Version Numbers

The next four bytes are the major and minor version numbers of the compiler used. The minor version of listing A.2, page 85 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.

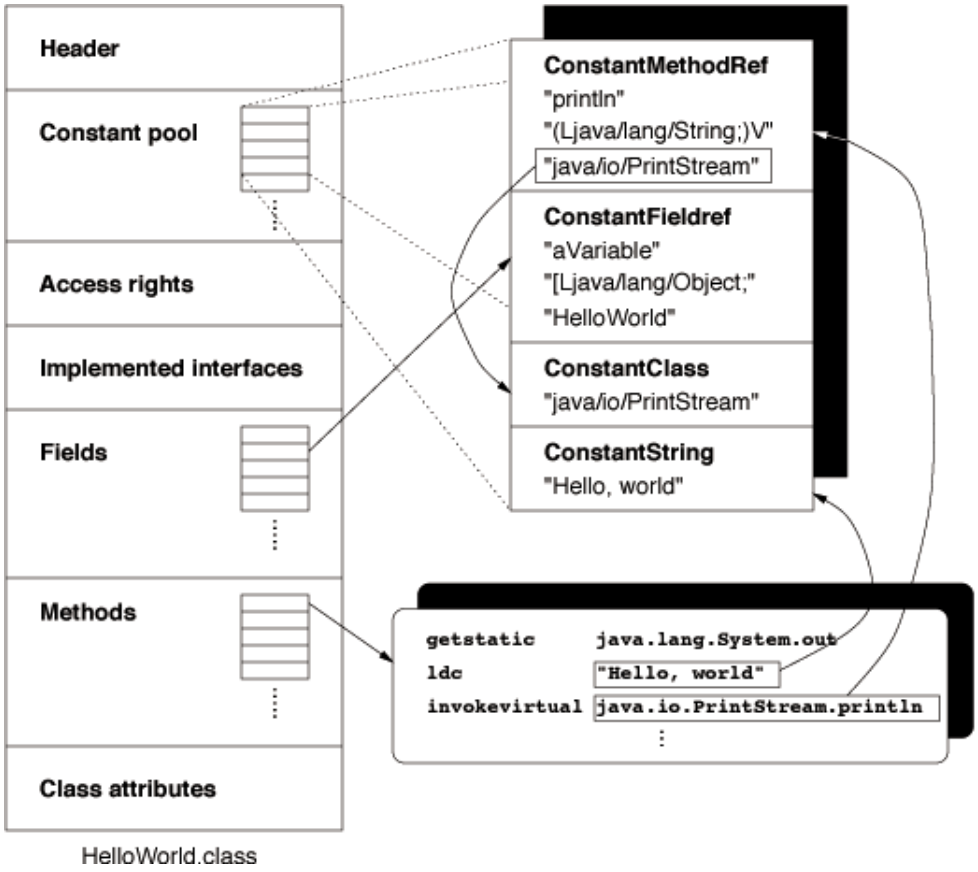

Figure A.1: Conceptual diagram of a Java class file. Source: [20]

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 A.2, page 85 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 A.1: Hello World Java Program

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

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 A.2: Hex Dump of Hello World Java Program

```
CAFEBABE00000031001D0A0006000F09001000110800120A
0013001407001507001601000 63C696E69743E0100032829
56010004436F646501000F4C696E654E756D626572546162
6C650100046D61696E010016285B4C6A6176612F6C616E67
2F537472696E673B295601000A536F7572636546696C6501
000F48656C6C6F576F726C642E6A6176610C000700080700
170C0018001901000B48656C6C6F20576F726C6407001A0C
001B001C01000A48656C6C6F576F726C640100106A617661
2F6C616E672F4F626A6563740100106A6176612F6C616E67
2F53797374656D0100036F75740100154C6A6176612F696F
2F5072696E7453747265616D3B0100136A6176612F696F2F
5072696E7453747265616D0100077072696E746C6E010015
284C6A6176612F6C616E672F537472696E673B2956002100
0500060000000000020001000700080001000900000001D00
010001000000052AB70001B100000001000A000000060001
000000010009000B000C000100090000002500020001000 0
0009B200021203B60004B100000001000A0000000A000200
0000030008000400010000D00000002000E
```

Listing A.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 "Hello_World">
5       invokevirtual #4 <Method java/io/PrintStream.println (Ljava/lang/String;)V>
8       return
```

Listing A.4: Simple use of Java if statement

```
public class if_simple {

        public static void main(String[] args) {
                if(args[0] == "test") {
                        System.out.println("hello");
                }else{
                        System.out.println("goodbye");
                }
        }

}
```

Listing A.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>":()V
   4:   return

public static void main(java.lang.String[]);
  Code:
   0:   aload_0
   1:   iconst_0
   2:   aaload
   3:   ldc     #2; //String test
   5:   if_acmpne       19
   8:   getstatic       #3; //Field java/lang/System.out:Ljava/io/PrintStream;
   11:  ldc     #4; //String hello
   13:  invokevirtual   #5; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   16:  goto    27
   19:  getstatic       #3; //Field java/lang/System.out:Ljava/io/PrintStream;
   22:  ldc     #6; //String goodbye
   24:  invokevirtual   #5; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   27:  return

}
```

Listing A.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 A.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>":()V
    4:   return

public static void main(java.lang.String[]);
  Code:
    0:   iconst_0
    1:   istore_1
    2:   iload_1
    3:   bipush  100
    5:   if_icmpge       21
    8:   getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
    11:  iload_1
    12:  invokevirtual   #3; //Method java/io/PrintStream.println:(I)V
    15:  iinc    1, 1
    18:  goto    2
    21:  iconst_0

    22:  istore_1
    23:  iload_1
    24:  bipush  100
    26:  if_icmpge       42
    29:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
    32:  iload_1
    33:  invokevirtual   #3; //Method java/io/PrintStream.println:(I)V
    36:  iinc    1, 1
    39:  goto    23
    42:  return

}
```

Listing A.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>":()V
    4:   return

public static void main(java.lang.String[])   throws java.lang.Exception;
  Code:
    0:   iconst_0
    1:   istore_1
    2:   iconst_0
    3:   istore_2
    4:   iconst_0
    5:   istore_3
    6:   iconst_1
    7:   istore  4
    9:   iconst_1
    10:  istore  5
    12:  new     #2; //class java/util/Scanner
    15:  dup
    16:  getstatic       #3; //Field java/lang/System.in:Ljava/io/InputStream;
    19:  invokespecial   #4; //Method java/util/Scanner."<init>":(Ljava/io/InputStream;)V
    22:  astore  6
    24:  aload   6
    26:  invokevirtual   #5; //Method java/util/Scanner.nextInt:()I
    29:  istore_1
    30:  goto    38
    33:  astore  7
    35:  iconst_0
    36:  istore  4
    38:  aload   6
    40:  invokevirtual   #5; //Method java/util/Scanner.nextInt:()I
    43:  istore_2
    44:  goto    52
    47:  astore  7
    49:  iconst_0
    50:  istore  5
    52:  iconst_0
    53:  istore  7
    55:  iconst_0
    56:  istore  7
    58:  iload   7
    60:  tableswitch{ //0 to 6
                0: 104;
                1: 107;
                2: 127;
                3: 164;
                4: 147;
                5: 177;
                6: 190;
                default: 204 }
    104: iconst_m1
    105: istore  7
    107: iconst_m1
    108: istore  7
    110: iload   4
    112: ifeq    121
    115: iconst_2
    116: istore  7
    118: goto    204
    121: iconst_3
    122: istore  7
    124: goto    204
    127: iconst_m1
    128: istore  7
    130: iload   5
    132: ifeq    141
    135: iconst_4
    136: istore  7
    138: goto    204
    141: iconst_5
    142: istore  7
    144: goto    204
    147: iconst_m1
    148: istore  7
    150: getstatic       #7; //Field java/lang/System.out:Ljava/io/PrintStream;
    153: ldc     #8; //String loop
    155: invokevirtual   #9; //Method java/io/PrintStream.print:(Ljava/lang/String;)V
    158: iconst_2
    159: istore  7
    161: goto    204
    164: iconst_m1
    165: istore  7
    167: iinc    1, 1
    170: bipush  6
    172: istore  7
    174: goto    204
    177: iconst_m1
    178: istore  7
    180: iinc    2, 1
    183: bipush  6
    185: istore  7
    187: goto    204
    190: iconst_m1
    191: istore  7
    193: iload_1
    194: iload_2
    195: iadd
    196: istore_3
    197: getstatic       #7; //Field java/lang/System.out:Ljava/io/PrintStream;
    200: iload_3
    201: invokevirtual   #10; //Method java/io/PrintStream.print:(I)V
    204: iload   7
    206: iconst_m1
```

Appendix B

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

Bibliography

[1] Asm.

[2] Cojen.

[3] Gcj: The gnu compiler for java.

[4] Jamaica, the JVM macro assembler.

[5] Janino – an embedded java[tm] compiler.

[6] The jastadd extensible java compiler.

[7] Java code protector.

[8] Java optimize and decompile environment (JODE).

[9] The java programming-language compiler (javac).

[10] Jikes' home.

[11] Kopi java compiler.

[12] Openjdk.

[13] ProGuard.

[14] Tea trove.

[15] tinapoc - the java reverse engineering toolkit.

[16] Zelix KlassMaster.

[17] Decompiler exception test source. http://www.program-transformation.org/Transform/DecompilerExceptionTestSource, 2003. from Decompiling Java Bytecode: Problems, Traps and Pitfalls.

[18] Decompiler fibo test source. http://www.program-transformation.org/Transform/DecompilerFiboTestSource, 2003.

[19] Origin of 0xcafebabe. http://radio.weblogs.com/0100490/2003/01/28.html, 2003.

[20] Bcel - byte code engineering library (bcel). http://jakarta.apache.org/bcel/manual.html, June 2006.

[21] Asm - users. http://asm.objectweb.org/users.html, October 2008.

[22] Ahpah Software. SourceAgain. `http://www.ahpah.com/cgi-bin/suid/~pah/demo_license.cgi`, 2004.

[23] 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.

[24] Michael Batchelder and Laurie J. Hendren. Obfuscating java: the most pain for the least gain. In *CC '07: Proceedings of Compiler Construction, 16th International Conference*, pages 96–110, 2007.

[25] Ben Bellamy, Pavel Avgustinov, Oege de Moor, and Damien Sereni. Efficient local type inference. *SIGPLAN Not.*, 43(10):475–492, 2008.

[26] Swaroop Belur and Kartik Bettadapura. Jdec: Java decompiler. `http://jdec.sourceforge.net/`, 2008.

[27] Russel Impagliazzo Steven Rudich Amit Sahai Salil Vadhan Ke Yang Boaz Barak, Oded Goldreich. On the (im)possiblity of obfuscating programs.

[28] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. Technical report, 2002.

[29] 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.

[30] W.L. Caudle. On inverse of compiling. *Sperry-UNIVAC*, April 1980.

[31] C. Cifuentes. *Reverse Compilation Techniques.* Phd thesis, Queensland University of Technology, 1994.

[32] S. Cimato, A. De Santis, and U. Ferraro Petrillo. Overcoming the obfuscation of java programs by identifier renaming. *J. Syst. Softw.*, 78(1):60–72, 2005.

[33] 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*, pages 25–35, New York, NY, USA, 1989. ACM.

[34] Markus Dahm. Byte code engineering with the bcel api. Technical report, Freie University, Berlin, Institut fur Informatik, 2001.

[35] Stephen de Vries. ChainKey java code protection bypass issue, June 2006.

[36] Edsger W. Dijkstra. Letters to the editor: go to statement considered harmful. *Commun. ACM*, 11(3):147148, 1968.

[37] Emmanuel Dupuy. Java decompiler. `http://java.decompiler.free.fr/`, 2008.

[38] Bruce Eckel and Kirk Pepperdine. *JDT more correct than Javac.* 2006. Published: \urlhttp://www.theserverside.com/news/thread.tss?thread_id=38644.

[39] Torbjrn Ekman and Grel Hedin. The jastadd extensible java compiler. In *Object-Oriented Programming, Systems and Languages (OOPSLA)*, page 1?18, 2007.

[40] Torbjrn Ekman and Grel Hedin. The jastadd system ?- modular extensible compiler construction. *Sci. Comput. Program.*, 69(1-3):14?26, 2007.

[41] Michael Van Emmerik. *Static Single Assignment for Decompilation.* Phd thesis, The University of Queensland, 2007.

[42] Mike Van Emmerik. Java decompiler tests. `http://www.program-transformation.org/Transform/JavaDecompilerTests`, 2003.

[43] 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.

[44] Barry Fagin and Martin Carlisle. Connect Four(TM) game, written in ada, 2005.

[45] The Eclipse Foundation. Jdt core component.

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

[47] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)).* Addison-Wesley Professional, 2005.

[48] James Gosling and Henry Mcgilton. The java language environment: A white paper. Technical report, October 1996.

[49] K John Gough and Diane Corney. Implementing languages other than java on the java virtual machine. 2001.

[50] Vladimir Grishchenko and Johann Gyger. JadClipse. `http://jadclipse.sourceforge.net/wiki/index.php/Main_Page`, 2009.

[51] Peter Haggar. Understanding bytecode makes you a better programmer. `http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/`, July 2001.

[52] Maurice H Halstead. *Elements of software science (Operating and programming systems series).* Elsevier, 1977. Published: Hardcover.

[53] Jochen Hoenicke. JODE. `http://jode.sourceforge.net/`, 2004.

[54] Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):10311052, 1997.

[55] Alex Kalinovsky. *Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering.* Pearson Higher Education, 2004.

[56] Kearney, Sedlmeyer, Thompson, Gray, and Adler. Software complexity measurement. *Commun. ACM*, 29(11):10441050, 1986.

[57] Todd B. Knoblock and Jakob Rehof. Type elaboration and subtype completion for java bytecode. *ACM Trans. Program. Lang. Syst.*, 23(2):243–272, 2001.

[58] Donald E. Knuth. Structured programming with go to statements. *ACM Comput. Surv.*, 6(4):261301, 1974.

[59] Pavel Kouznetsov. Jad - the fast java decompiler. http://www.kpdus.com/jad.html, March 2006.

[60] Douglas Kramer, Bill Joy, and David Spenhoff. The java[tm] platform. Technical report, Sun Microsystems, May 1996.

[61] Eugene Kuleshov. Using the asm framework to implement common java bytecode transformation patterns. Vancouver, Canada, March 2007.

[62] Karthik Kumar. JReversePro - java decompiler / disassembler. `http://jreversepro.blogspot.com`, 2005. JReversePro is a java decompiler / disassembler written in Java.

[63] Mark D. Ladue. When java was one: Threats from hostile byte code. In *Proceedings of the 20th National Information Systems Security Conference*, 1997.

[64] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. *J. Autom. Reason.*, 30(3-4):235–269, 2003.

[65] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*. Prentice Hall PTR, April 1999. Published: Paperback.

[66] Mayon Software Research. ClassCracker 3. `http://mayon.actewagl.net.au/`, 2005.

[67] Jon Meyer and Troy Downing. *The Java Virtual Machine*. pub-ORA, pub-ORA, 1997.

[68] Jonathan Meyer, Daniel Reynaud, Iouri Kharon, et al. Jasmin, 2004.

[69] Jerome Miecznikowski. *New algorithms for a Java decompiler and their implementation in Soot*. Masters thesis, McGill University, 2003.

[70] 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.

[71] 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 111?127, London, UK, 2002. Springer-Verlag.

[72] 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.

[73] 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.

[74] Nomair A. Naeem. *Programmer-Friendly Decompiled Java*. Masters thesis, 2007.

[75] 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.

[76] 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 327—336. IEEE Computer Society, 2006.

[77] Godfrey Nolan. *Decompiling Java*. APress, 2004.

[78] Tim Ottinger. Meaningful names. http://tottinge.blogsome.com/meaningfulnames/, 1997.

[79] David J. Pearce. The java compiler kit (jkit).

[80] Todd A Proebsting and Scott A Watterson. Krakatoa: Decompilation in java (does bytecode reveal source?). pages 185–197, 1997.

[81] Lyle Ramshaw. Eliminating go to's while preserving program structure. *J. ACM*, 35(4):893–920, 1988.

[82] 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.

[83] Marco Schmidt. List of java bytecode compilers. http://schmidt.devlib.org/java/bytecode-compilers.html.

[84] SourceTec Software Inc. SourceTec (Jasmine). `http://www.sothink.com/product/javadecompiler/index.htm`, 1997.

[85] R. F. Strk, J. Schmid, and E. Brger. *Java and the Java Virtual Machine: Definition, Verification and Validation*. Springer-Verlag, 2001.

[86] Robert Tolksdorf. Programming languages for the java virtual machine jvm. http://www.is-research.de/info/vmlanguages/index.html.

[87] 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.

[88] Hanpeter van Vliet. Mocha, the java decompiler, 1996.

[89] Bill Venners. *Inside the Java Virtual Machine*. McGraw-Hill, Inc., New York, NY, USA, 1996.