Noname manuscript No.
(will be inserted by the editor)

An Empirical Evaluation of Java Bytecode Decompiler Effectiveness

James Hamilton · Sebastian Danicic

Received: date / Accepted: date

Abstract Decompilation of Java bytecode is the act of transforming Java bytecode to Java source code. Although easier than that of decompilation of machine code, problems still arise in Java bytecode decompilation. These include type inference of local variables and exception-handling.

Since the last such evaluation (2003) several new commercial, free and open-source Java decompilers have appeared and some of the older ones have been updated.

In this paper, we evaluate the currently available Java bytecode decompilers using an extension of the criteria that were used in the original study. Although there has been a slight improvement since this study, it was found that none passed all of the tests, each of which were designed to target different problem areas. We give reasons for this lack of success and suggest methods by which future Java bytecode decompilers could be improved.

Keywords decompilation · program transformation · reverse engineering · Java

This paper is a revised, updated and extended version of a previously published paper [Hamilton and Danicic(2009)] which summarised our results. This extended paper provides a more in-depth analysis of the results of our tests and encourages the reader to study the quality of output from the latest versions of various decompilers available.

James Hamilton
Department of Computing, Goldsmiths, University of London, New Cross, London, SE14 6NW, United Kingdom
E-mail: james.hamilton@gold.ac.uk

Sebastian Danicic
Department of Computing, Goldsmiths, University of London, New Cross, London, SE14 6NW, United Kingdom
E-mail: S.Danicic@gold.ac.uk

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 [Haggar(2001)].

Compilation is the act of transforming a high-level language, into a low-level language such as machine code or bytecode. Decompilation is the reverse. It is the act of transforming a low-level language into a high-level language [Caudle(1980)].

Decompilation can be used to assist software theft, also known as software piracy - the act of copying a legitimate application and illegally distributing that software, either free or for profit. There are many methods to protect software including legal methods such as copyright laws, patents and license agreements but these do not always dissuade people from stealing software, especially in emerging markets where the price of software is high and incomes are low. Ethical arguments, such as fair compensation for producers, by software manufacturers, law enforcement agencies and industry lobbyists also do little to counter software piracy. The global revenue loss due to software piracy was estimated to be more than \$50 billion[1] in 2008 [Alliance(2008)]. We therefore require technical measures [Collberg and Thomborson(2002), Collberg and Nagra(2009)] to reduce software piracy, such as software watermarking [Collberg and Thomborson(1998)] and/or obfuscation [Collberg et al(1997)Collberg, Thomborson, and Low].

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

This paper is structured as follows: In section 1.1 we introduce the programming language Java, decompilation and the decompilers. In section 2 we discuss the problems related to Java decompilation. In section 3 we introduce the test programs relating them to decompilation problems, describe our method of evaluation and provide a summary of results. In section 4 we discuss, in detail, the decompiled programs, of which program listings are provided in appendix A. Finally, we draw conclusions and suggest directions for future work in section 5.

[1] This figure is disputed, by some, as the report's conclusions are based on mathematical models not empirical data revealing specific instances of piracy [Pearson(2009), Masnick(2008)]

1.1 Background

1.1.1 Java and the Java Virtual Machine

The Java Virtual Machine is essentially a simple stack based machine which can be separated into five parts: the heap, program counter registers, method area, Java stacks and native method stacks [Venners(1996)] . The Java Virtual Machine Specification [Lindholm and Yellin(1999)] 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 [Lindholm and Yellin(1999)] and updated for Java 1.6 in JSR202 [Buckley et al(2006)Buckley, Rose, Coglio, Corporation, Sun Microsystems, Tmax Soft, Technologies, and AG].

An advantage of the virtual machine architecture is portability - any machine that implements the Java Virtual Machine Specification [Lindholm and Yellin(1999)] is able to execute Java bytecode hence the slogan "Write once, run anywhere" [Kramer et al(1996)Kramer, Joy, and Spenhoff]. Java bytecode is not strictly linked to the Java language and there are many compilers, and other tools, available which produce Java bytecode [Tolksdorf(2010),Gough and Corney(2001)] such as the Jikes compiler, Eclipse Java Development Tools or Jasmin bytecode assembler.

Another advantage of the Java Virtual Machine is the runtime type-safefy of programs. These two main advantages are properties of the Java Virtual Machine not the Java language [Gough and Corney(2001)] which, combined, provides an attractive platform for other languages.

Java bytecode can be generated in three ways:

1. from a Java source program using a Java compiler (such as Sun's *javac*),
2. using a language other than Java to Java bytecode compiler (such as JGNAT [jgn(2009)] - an open-source Ada to Java bytecode compiler) or
3. by writing a class file by hand.

The tedious task of hand-writing a Java class file can be made easier by using a Java assembler, such as Jasmin [Meyer et al(2004)Meyer, Reynaud, Kharon et al], which accepts a human readable form of Java bytecode instructions and generates a Java class file. We use Jasmin source in this paper for examples programs which are written manually, or for examples which require inspection of the bytecode.

Java bytecode can also be manipulated by tools such as obfuscators and optimisers which perform semantics preserving transformations on bytecode contained within a Java class file. Figure 1 shows the Java bytecode cycle from generation to decompilation to Java source.

Java bytecode retains type information about fields, method returns and parameters but it does not, for example, contain type information for local variables. The type information in the Java class file renders the task of decompilation of bytecode easier than decompilation of machine code [Emmerik(2007)]. Decompiling Java bytecode, thus, requires analysis of most local variable types, flattening of stack-based instructions and structuring of loops and conditionals. The task of bytecode decompilation, however, is much harder than compilation. We show that often decompilers cannot fully perform their intended function [Cifuentes(1994)].

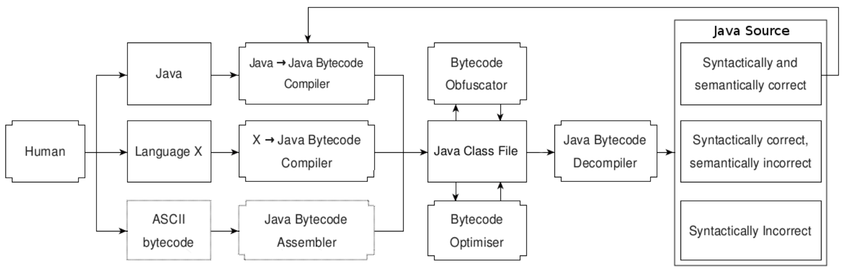


Fig. 1: Compilation and decompilation of Java Bytecode

1.1.2 Decompilation

The problems to be solved by a general decompiler [Emmerik(2007)] can be divided into different categories :

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

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

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

Decompilation has many applications including legitimate uses, such as the recovery of lost source code for a crucial application [Emmerik and Waddington(2004)] and non-legitimate uses such as reverse-engineering a proprietary application. Consider the case in which a company has lost the source code for their application and to continue development on the software they require recovery of source code from Java class files. The company must decompile the Java class files and attempt to recover Java source equivalent to the originally lost source. In this case, in comparison to an illegitimate use, it is likely that the company knows more about how the Java class files were generated. Knowledge of how class files are generated provides information useful in the recovery of the original source as a decompiler can be optimised for the compiler used.

The purpose of a decompilation task indicates the level of decompilation to be achieved. Recompilation of a decompiled program does not require that the source be easy to read or tidy. It does not even require that type inference is performed as long as the program is correctly type-casted. Many of the existing Java decompilers do not perform type inference [Emmerik(2003)] and, while much research has gone into creating efficient type inference algorithms [Gagnon et al(2000)Gagnon, Hendren, and Marceau, Bellamy et al(2008)Bellamy, Avgustinov, de Moor, and Sereni], code with correct type-casts is semantically equivalent.

On the other hand, if the purpose of decompilation is to produce maintainable, extensible code then readable code with full typing is desirable. Such a decompiler should produce code that is similar to a human programmer so that future work on the code can be undertaken easily. Some decompilers produce output which is more readable than others, while optimisations have been been to some decompilers to produce more readable code [Naeem and Hendren(2006), Naeem(2007)].

If the purpose of decompilation is to simply understand a program, the syntactical correctness of a complete decompiled program may not be a high-priority. Correct portions of an incorrect program could help in the understanding of a program, in contrast to the case of source recovery where correct source is needed. Software theives would be able to analyse partial Java programs [Dagenais and Hendren(2008)] allowing them to remove, for example, product key validation code.

There is a fair amount of literature on decompilation but not a lot covering specifically Java decompilation. A lot of interesting research in this subject and generally Java optimisation is performed by the Sable Research Group[2] at McGill University. They have created Soot [Valle-Rai et al(1999)Valle-Rai, Co, Gagnon, Hendren, Lam, and Sundaresan] - a Java optimisation framework which includes a Java decompiler called Dava [Miecznikowski and Hendren(2002)]. Several commercial decompilers exists (e.g. Class Cracker) thought most have been unmaintained for several years.

The design of a decompiler is made easier if it only has to decompile code produced by Sun's *javac* Java compiler as it mostly means inverting a known compilation strategy [Miecznikowski and Hendren(2002)]. *javac* is open-source so developers can see the implementation of the compiler and relatively easily invert it; of course there are still some problems to overcome. Many of the existing decompilers expect classfiles generated by *javac*.

The Java bytecode Verifier and, from version 1.6 onwards, the Java bytecode Checker check the validilty of Java bytecode as a matter of program security. This bytecode validation phase ensures that programs are 'well-behaved'. The verification process does not require that a program is compiled with *javac* and it is possible to create class files which no Java compiler can produce yet they pass the Verifier with flying colours [Ladue(1997)]. This has caused security concerns since the early days of Java.

The decompilation of arbitrary, verifiable bytecode is more difficult than that of decompiling *javac* produced bytecode due the ability to create or change class files in ways that are able to pass verification but do not contain expected bytecode. The decompilation of such arbitary bytecode causes many Java decompilers to fail which prompted the development of Dava [Miecznikowski and Hendren(2002)] - a decompiler designed to deal with arbitrary bytecode.

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

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 [Emmerik(2007)]. Some of the early decompilers are fooled by the insertion of a *pop* opcode at the end of a method as this is unexpected even though it passes the Java Verifier.

One source of the problem of arbitrary bytecode is that the class file format is entirely independent of the Java language and bytecode is more powerful than the Java language [Ladue(1997)]. For example in the Java Virtual Machine specification [Lindholm and Yellin(1999)] in relation to exception handling it lists the conditions with which *javac* produces exception handling bytecode but it notes that there are no restrictions enforced by the Verifier to check these conditions and suggests this does not pose a threat to the integrity of the Java Virtual Machine. Therefore unexpected exception handling code can be written and still pass the verification process.

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

The decompilation of Java bytecode involves transforming the low-level, stack based bytecode instructions into high-level Java source. There are several main problems [Miecznikowski and Hendren(2002), Emmerik(2007)] to solve in the decompilation of Java bytecode:

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

Decompiling Java bytecode is easy when compared to decompiling machine code as there are far fewer problems to overcome due to the amount of information contained within a class file. Java bytecode decompilers have the following advantages [Emmerik(2007)] over machine code decompilers:

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

1.2 Java Decompilation Example

In this section, we present a step by step decompilation of the Jasmin code in listing 1 using a stack evaluation method and taking advantage of Java to Java bytecode patterns.

Listing 1: Jasmin code for sum method

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

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

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

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

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

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

```
int total = 0;
```

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

```
int i = 0;
```

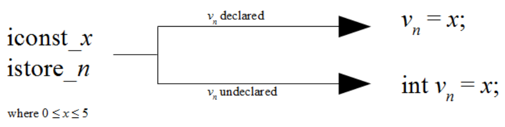


Fig. 2: Decompilation Pattern - Integer variable assignment

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

numbers
total

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

numbers.length
total

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

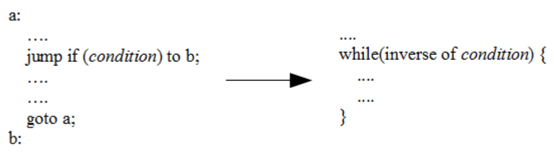


Fig. 3: Decompilation Pattern - While Loop

the bytecode condition transfers control flow to the end of the loop if the condition is true; however, we want the loop to start at the beginning. Our condition is therefore `i < numbers.length`.

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

i
numbers
total

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

numbers[i]
total

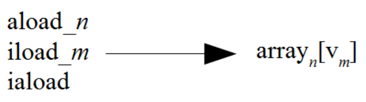


Fig. 4: Decompilation Pattern - Integer Array

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

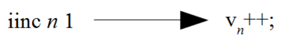
The following instruction, `istore_1`, pops an integer from the stack and stores it in local variable slot 1. We pop an expression from our expression stack and assign it to our variable `total`.

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

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

So the entire `while` loop looks like this:

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



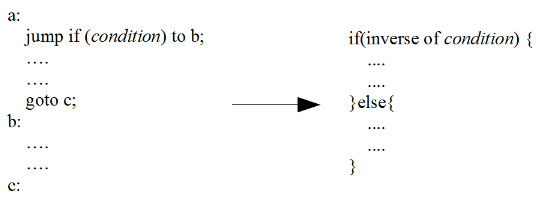
```
iinc n 1  ──────►  v_n++;
```

Fig. 5: Decompilation Pattern - Integer Increment

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

$$\boxed{\text{total}}$$

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



```
a:
  jump if (condition) to b;                if(inverse of condition) {
  ....                                       ....
  ....                                       ....
  goto c;                   ──────►        }else{
b:                                           ....
  ....                                       ....
  ....                                     }
c:
```

Fig. 6: Decompilation Pattern - If-then-else

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

"total is less than zero :("
java.lang.System.out

The next instruction `invokevirtual` invokes the specified instance method, by first popping the correct number of arguments from the stack and lastly popping the object reference, on which the instance method acts, from the stack. The method specified is `java/io/PrintStream/println(Ljava/lang/String;)V` - that is, in Java, the `void println(String s)` in the `java.io.PrintStream` class. The object reference `java.lang.System.out`, on the bottom of the stack, is an instance of `java.io.PrintStream`. We therefore pop the string from the stack, followed by the object reference and build our method call for Java:

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

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

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

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

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

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

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

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

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

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

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

Putting all this together gives us the Java code in listing 2. We can tidy the code up slightly by removing `java.lang.`, converting the *while* loop to a *for* loop and turning the `StringBuilder` into standard string concatenation to obtain listing 3.

Listing 2: Java code for sum method

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

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

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

Listing 3: Java code for sum method - nicer version

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

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

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

1.3 The Decompilers

The currently available decompilers are summarised in figure 7 and explained further in this section.

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

Fig. 7: The Decompilers

1.4 Mocha

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

1.5 SourceTec (Jasmine)

SourceTec [sou(1997)], 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).

1.6 SourceAgain

SourceAgain [Software(2004)] was a commercial decompiler from Ahpah Software, Inc. It is no longer sold or supported though they keep a web based version of their decompiler available on their website which can only decompile single class files. The original survey [Emmerik(2003)] used the Pro version which is no longer available.

1.7 ClassCracker3

ClassCracker3 [Research(2005)] 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.

1.8 Jad

Jad [Kouznetsov(2001)] is a popular decompiler that is free for non-commercial use but is no longer maintained[3]. It is a closed source program written in C. The last update for Linux and Windows version for Jad was in 2001, while a small update added an OS X version in 2006. Jad is used as the back-end by many decompiler GUIs including an Eclipse IDE plug-in named JadClipse [Grishchenko and Gyger(2009)].

1.9 JODE

JODE [Hoenicke(2004)] is an open-source decompiler that also includes a bytecode optimiser. The latest version 1.1.2-pre1 was released February 24, 2004. JODE performed best in the original survey [Emmerik(2003)] by decompiling six out of ten programs correctly.

1.10 jReversePro

jReversePro [Kumar(2005)] is an open-source disassembler and decompiler project which is currently at version 1.4.2 though hasn't been updated for several years.

1.11 Dava

Dava [Miecznikowski(2003), Naeem and Hendren(2006), Naeem(2007), Miecznikowski and Hendren(2001), Miecznikowski and Hendren(2002)] is a decompiler which is part of the Soot Java Optimisation Framework [Valle-Rai et al(1999)Valle-Rai, Co, Gagnon, Hendren, Lam, and Sundaresan] from the Sable Research Group[4] at McGill University in Montreal, Quebec, Canada. Soot is under constant development at the Sable Research Group and the latest release was version 2.4.0 on March 29th, 2010. The first version of our evaluation used version 2.3.0.

1.12 jdec

jdec [Belur and Bettadapura(2008)] 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.

1.13 Java Decompiler

Java Decompiler [Dupuy(2009)] is a free Java decompiler aimed at decompiling Java 5 and above class files. It is in its early stages, at only version 0.3.2, and has been in development for about a year. The first version of our evaluation used version 0.2.7.

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

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

1.14 NMI Code Viewer

NMI Code Viewer was included in the original survey with results ‘startlingly similar’ to Jad [Emmerik(2003)]. We do not include this (unmaintained) decompiler as, in actual fact, it is a front-end for Jad [Kouznetsov(2001)].

1.15 jAscii

jAscii was included in the original survey, with poor results [Emmerik(2003)], but it is now obsolete and unavailable for our evaluation.

2 Problems with Java Decompilation

2.1 Casting

The program in listing 4 shows an example in which some decompilers omit the int typecast [Emmerik(2003)].

Listing 4: Java Casting example

```
public class Type {

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

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

}
```

2.2 Inner Classes

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

2.3 Type Inference

Type inference is a general problem in decompilation and is needed to recover type information lost during the compilation process. Type analysis in Java bytecode decompilation is easier than that of machine code decompilation due to the explicit typing of class fields and parameters in a Java class file. It is easier to decompile Microsoft's CIL bytecode as all types are explicit in the assembly files [Emmerik(2007)].

One of the first papers written on decompiling Java was a description of a decompiler named Krakatoa [Proebsting and Watterson(1997)] which focused on two problems: merging stack-based instructions into expressions and turning arbitrary control flow into high-level Java constructs. The type-inference problem was dismissed as trivial and solvable by well known techniques such as those used by the Java bytecode Verifier. The Verifier subjects Java programs to a verification process in order to guarantee that the code will behave normally [Rose(1998)].

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 [Gagnon et al(2000)Gagnon, Hendren, and Marceau]. The type analysis performed by the bytecode Verifier estimates the types of local variables at each program point. This is used to ensure that each bytecode instruction is operating on the correct type. For example if the Verifier encounters the *iadd* opcode it will ensure that there are two integers at the top of the stack at that program point [Leroy(2003)]. The type inference problem for decompilation must give types to each variable that are valid for all uses of those variables.

If the type inference algorithm is optimised for the common-case rather than the worst case it is still possible to perform type analysis for most real-world code as worst-case scenerios are unlikely [Bellamy et al(2008)Bellamy, Avgustinov, de Moor, and Sereni]. Bellamy *et al.* [Bellamy et al(2008)Bellamy, Avgustinov, de Moor, and Sereni] provide a common-case optimised algorithm to perform type analysis which outperforms Gangon *et al.*'s [Gagnon et al(2000)Gagnon, Hendren, and Marceau] worst-case optimised algorithm. The algorithm relies on the assumption that multiple inheritance, via interfaces, is rarely used in real-world Java programs which could prove a problem if their assumption is false, or becomes false in the future.

Java's restricted form of multiple inheritance, interfaces, leads to problems in finding a static type in some cases where the code is valid but not generated directly from Java source [Gagnon et al(2000)Gagnon, Hendren, and Marceau]. This causes problems for some decompilers which expect *javac* generated code. Using an static single assignment form, as in [Knoblock and Rehof(2001)], may change the type hierarchy when types conflict due to interfaces [Miecznikowski and Hendren(2002)].

Listing 5, page 18 (from [Strk et al(2001)Strk, Schmid, and Brger]) shows a program which some decompilers may decompile better than others. If a decompiler does not perform type inference it will type x as Object and insert a typecast in the invocation of method m2(Comparable) whereas others may correctly type x as Comparable. The Java Virtual Machine Specification states that "the merged state contains a reference to an instance of the first common superclass of the two types" in regards to the bytecode verifier - the first common superclass of Integer and String is Object. Therefore the verifier has to insert a run-time check to ensure that both Integer and String are of type Comparable.

2.4 Control Flow

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

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

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

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

Ramshaw's *goto* elmination technique can be used to replace *goto*s in a program, if and only if the control flow graph is reducible, by replacing them with multi-level loop exit statements [Ramshaw(1988)]. The Krakatoa decompiler uses an extended version of Ramshaw's *goto* elimination technique [Proebsting and Watterson(1997)].

Java bytecode can contain non-reducible control flow which cannot be represented in Java source requiring techniques such as described in [Janssen and Corporaal(1997)]

Listing 5: What is the type of x? [Strk et al(2001)Strk, Schmid, and Brger]

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

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

    m2(x);
  }

  public void m2(Comparable x) {
  }

//bytecode:

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

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

to be applied to transform irreducible control flow graphs to reducible control flow graphs.

2.5 Exceptions

An exception, as described in the Java Language Specification [Gosling et al(2005)Gosling, Joy, Steele, and Bracha], 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*.

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

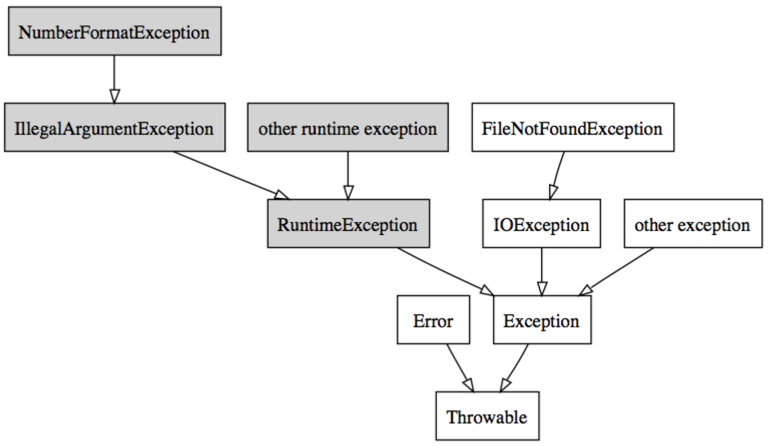


Fig. 8: Java Exception Hierarchy. Unchecked exception classes are coloured grey.

Listing 6: Example of try-catch

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

allow programmers to catch errors from which recovery is usually possible (Exception) and not require the inclusion of extra code to catch errors from which recovery is usually impossible (Error). Figure 8, page 19 shows the Java exception hierarchy.

Figure 6, page 19 shows a try-catch block with a call to the method *Integer.parseInt* that takes a Unicode string and returns it's *int* value or throws a NumberFormatException if the string does not represent an integer. If an exception is thrown control passes to the catch block corresponding to the class, or a super-class, of the exception thrown. The first catch clause, in order of appearance in the source file with a matching type is executed - in this case if *s* is not an integer the message "Enter an integer" will be displayed. The order of catch clauses is important because more than one catch clause could handle the same exception. For example the clause *catch(Exception e)* {...} can also handle the NumberFormatException as Exception is a superclass; subclass catch clauses must precede superclass catch clauses.

Programmers can define their own exceptions by extending Throwable or any of its subclasses. The Java Language Specification [Gosling et al(2000)Gosling, Joy, Steele, and Bracha] recommends that all user defined exceptions extend Exception rather than Throwable or RuntimeException because Exception and its subclasses are **checked** exceptions whereas RuntimeException and its subclasses are **unchecked**.

A Java compiler ensures that a program contains handlers for checked exceptions during compilation so for each method which could possibly throw a checked exception there must be a handler or the method must declare that it itself throws the exception.

Listing 7: Example of checked exceptions

```
public class CheckedException {

        public static void main(String[] args) {

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

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

        }

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

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

If a method declares that it throws an exception it must be caught or thrown in the invoking method, and so on.

Listing 7, page 20 shows three methods which return a FileReader object for the specified filename. FileReader's constructor throws a FileNotFoundException if the file specified was not found. This is a checked exception therefore any invocation of the constructor requires an exception handler or that the method containing the invocation declares that itself throws FileNotFoundException (or a superclass of FileNotFoundException).

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

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

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

Programmers can throw exceptions themselves using the *throw* keyword. Listing 8, page 21 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.

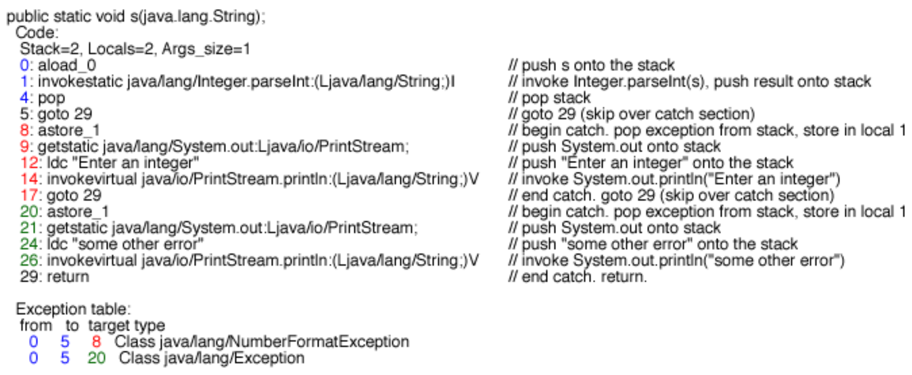
WrongInputException is a subclass of Exception which means that it is a checked exception therefore when the yesOrNo method is invoked an exception handler must be used or the calling method must declare that it throws a WrongInputException.

Listing 8: Throwing an exception

```
public class YesOrNo {

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


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

}


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

Java bytecode contains the necessary information to implement exceptions as specified by the Java Language Specification [Gosling et al(2000)Gosling, Joy, Steele, and Bracha]. 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 9 shows the bytecode, generated by *javac* 1.6, for listing 6, page 19. It contains two entries in the exception table corresponding to the two types of exception which are to be caught.

The first exception table entry lists an exception handler for the type NumberFormatException between bytes 0 (inclusive) to 5 (exclusive) and the second entry lists an exception handler for the type Exception between bytes 0 (inclusive) to 5 (exclusive). The Java Virtual Machine searches the exception table starting at the top for the first matching entry for the type of exception that was thrown. So if a NumberFormatEx-



```
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
```

Fig. 9: Java bytecode for listing 6, page 19.

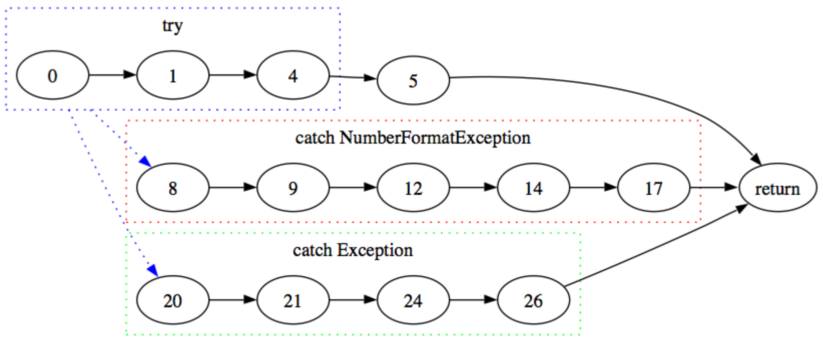


Fig. 10: Java bytecode control flow graph for listing 6, page 19 with highlighted try-catch sections.

ception 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.

2.5.1 Try-Finally

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

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

Java bytecode subroutines, like subroutines in any other programming language, were designed to allow code re-use for the implementation of finally clauses. For each exit point in a try clause the same finally block must be executed which seems to suggest subroutines would be a good idea to implement this. Java 1.4.2 and above repeat the finally clause bytecode at every try exit point rather than using subroutines. This has the disadvantage that the code size is much bigger than when using subroutines but data flow analysis is much simplified [Freund(1998)]. It also means that bytecode verification is simplified as the verifier performs a form of data flow analysis on the bytecode. However, bytecode produced by javac <1.4.2 and other compilers could still contain subroutines.

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

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

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

2.6 Variable Re-Use

It is possible for a local variable slot to take values of distinct types at different places in a method [Gagnon et al(2000)Gagnon, Hendren, and Marceau] which is not possible in Java. For example listing 9 shows a simple valid method in which local variable 0 is used to store an integer then a String. At byte 1 local variable 0 contains a variable of type integer but at byte 3 it contains an object reference type. This is perfectly valid bytecode and is akin to the declaration of an integer followed by the declaration of a String in Java, but in the bytecode these two distinct variables are using the same local variable slot.

Multiple types for local variables is valid bytecode as long as the lifetimes of the two uses of the local variable does not overlap [Knoblock and Rehof(2001)] 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 [Alpern et al(1988)Alpern, Wegman, and Zadeck, Rosen et al(1988)Rosen, Wegman, and Zadeck, Cytron et al(1989)Cytron, Ferrante, Rosen, Wegman, and Zadeck] where

Listing 9: Multiple local variable types

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

all static assignments to a variable will have unique names. The possibility of multiple types for a single local variable slot causes a problem for decompilation as each variable needs a single type which is valid for all uses of that variable.

2.7 Arbitrary bytecode

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

3 Empirical Evaluation

Our evaluation is based on a previous survey conducted in 2003 [Emmerik(2003)] which tested many of the decompilers that are still available today. We attempt to decompile a set of test programs using each decompiler and manually inspect the results to classify them as one of our ten categories of decompiler effectiveness (section 3.2 describes our effectiveness measure). We use the test programs from the original survey, where possible, and also extend the evaluation to include more problem areas such as the correct decompilation of try-finally blocks and local variable slot re-use.

3.1 Test Programs

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

The original survey showed that different decompilers are sometimes better in different areas but no decompiler passed all the tests [Emmerik(2003)]. Of the decompilers tested in the original survey, JODE [Hoenicke(2004)] performed the best by correctly decompiling 6 out of the 9 test programs, with Dava [Miecznikowski(2003)] and Jad [Kouznetsov(2001)] close behind.

In our tests we include two types of Java class file: those generated by *javac* and those generated by other tools, which will be refered to as *arbitrary* bytecode class files. Java source files are compiled with *javac* version 1.6.0_10. Arbitrary Java class files include two hand-written using the Jasmin assembler version 2.1, one optimised using the Soot Framework [Valle-Rai et al(1999)Valle-Rai, Co, Gagnon, Hendren, Lam, and Sundaresan] and one compiled by JGNAT [jgn(2009)].

3.1.1 Fibo

Fibo (Listing 10, page 26) is a fairly simple program to output the Fibonacci number of a given input number. It should be a trivial test for a decompiler but contains many of the basic Java language constructs. The program tests decompilation of basic Java language constructs such as *if* statements, methods declarations & invocations and exception handling.

3.1.2 Casting

Casting [Nolan(2004)] is a simple program to test if a decompiler can correctly detect the need to cast a *char* to an *int*. The program (Listing 11, page 26) iterates through the first 128 ASCII characters and prints out the ASCII code and the character. A cast from *char* to *int* is used to achieve this. The original survey [Emmerik(2003)] showed that some decompilers were unable to detect the need for the cast and instead printed the ASCII character instead of the ASCII code.

Listing 12, page 27 shows the Jasmin source for the casting test program. The Java compiler (*javac*) converts string concatenations into string buffer appends for efficiency. The line marked #1 is the point at which the integer ASCII code is appended to the string buffer, and the line marked #2 is the point at which the ASCII character is appended to the string buffer - notice that line #1 invokes the method *append(integer)*, while line #2 invokes the method *append(character)*.

Listing 10: Fibo Test Program

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

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

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

}
```

Listing 11: Casting Test Program

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

3.1.3 Usa

Usa [Nolan(2004)] is a simple program (Listing 13, page 28) containing inner classes. The implementation of inner classes in Java is handled by the compiler - the Java Virtual Machine has no concept of inner classes. The Java compiler converts the inner classes in a Java source file into separate class files. For example, the Usa test program's main class is called *Usa* and it has an inner class called *England* - the Java compiler generates a class named *Usa$England* for the inner class and the inner class contains a private variable *this$0* which is a reference to the outer class. Some decompilers in the original survey [Emmerik(2003)] could not reconstruct the original Java source from the constituent class files.

3.1.4 Sable

Sable [Miecznikowski and Hendren(2002)] is a program which tests a decompiler's ability to perform type inference for local variables. Variable d in the program (line 3, Listing 14, page 29) is difficult for a decompiler to type because it depends on the value of the method parameter, i. If $i > 10$ then d becomes a *Rectangle* object, whereas if $i <= 10$ d becomes a *Circle* object. Variable d should be typed *Drawable* as this is the

Listing 12: Casting Test Program in Jasmin

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

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

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

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

    return

.end method

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

    Label2:
    iconst_0
    istore_1

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

    Label0:
    return

.end method
```

Listing 13: Inner class Test Program

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

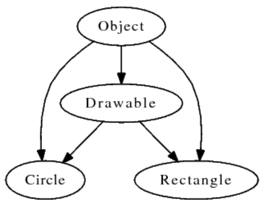


Fig. 11: Type hierarchy for the type inference test program. *Drawable* is the lowest common ancestor of *Circle* and *Rectangle*.

lowest common ancestor of *Circle* and *Rectangle* (see Figure 11, page 28). The original paper [Miecznikowski and Hendren(2002)], written by the developers of Dava, tested three different decompilers with the Sable test program, against Dava. They showed that their decompiler could correctly type variable *d* whereas the other decompilers failed to do so. Some decompilers do not perform type inference but insert a typecast where necessary in order to produce a semantically equivalent program [Miecznikowski and Hendren(2002), Emmerik(2003)].

3.1.5 TryFinally

TryFinally is a simple test program (Listing 18, page 30) to determine whether decompilers can decompile the implementation of try-finally blocks using in-line code instead of Java bytecode subroutines.

Listing 19, page 30 shows the Jasmin source for the TryFinally test program. Sections *b* and *d* contain duplicate code which is the finally clause of the TryFinally block. Compare this with listing 20, page 31 which shows the Jasmin source for the TryFinally test program compiled with *Jikes*. This version uses a Java subroutine (section *d*) to implement the finally clause of the test program, which is invoked from sections *b* and *e*.

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

Listing 14: Type Inference Test Program

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

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

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

Listing 15: Drawable interface for Type Inference Test Program

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

Listing 16: Circle class for Type Inference Test Program

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

Listing 17: Rectangle class for Type Inference Test Program

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

Listing 18: Try Finally Test Program

```
public class TryFinally {

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

}
```

Listing 19: Try Finally Test Program *javac* - Jasmin Source

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

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

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

.end method

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

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

Listing 20: Try Finally Test Program *Jikes* - Jasmin Source

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

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

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

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

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

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

.end method

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

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

    return

.end method
```

3.1.6 ControlFlow

ControlFlow [Miecznikowski and Hendren(2002)] is a program which tests a decompiler's handling of control flow. The program contains two while loops: one outer, infinite loop; and one inner loop which is contained within a try-catch block. If the

Listing 21: Control Flow Test Program

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

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

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

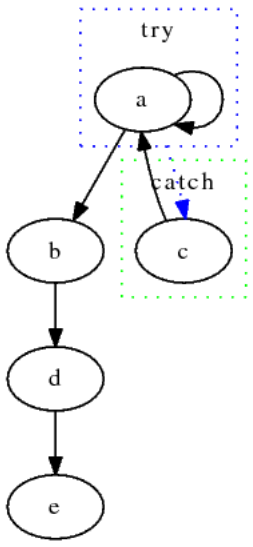


Fig. 12: Control flow graph for the control flow program.

inner loop throws a *RuntimeException* then the outer loop continues otherwise the outer loop is broken.

3.1.7 Exceptions

The Exceptions test program [Miecznikowski and Hendren(2002)] contains two intersecting try-catch blocks. The intersecting try-catch block is allowed in Java bytecode

Listing 22: Control Flow Test Program

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

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


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

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

    return

.end method


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

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

.end method
```

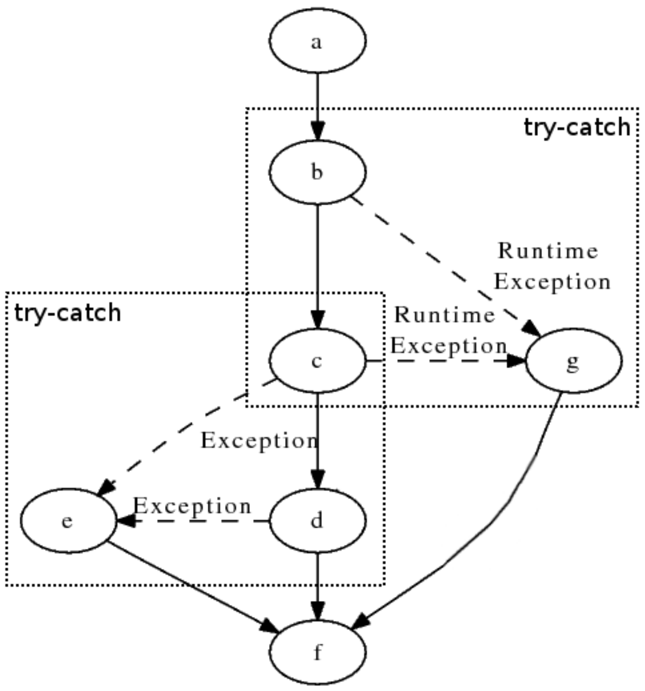


Fig. 13: Control flow graph for the exceptions test program [Miecznikowski and Hendren(2002)]. Solid edges indicate normal control flow while dashed edges represent exception control flow.

but would not be generated by a Java compiler - the program here is created using Jasmin. The program used in the original survey [Emmerik(2003)] is incorrect and Dava, which should be able to decompile the program, exits with a null pointer exception. A re-written version (Listing 23, page 34) is used in our tests based on the call graph in the original paper [Miecznikowski and Hendren(2002)]. Figure 13, page 34 shows the program's control flow graph and outlines the try-catch blocks.

3.1.8 Optimised

Optimised was generated by using the Soot optimiser [Valle-Rai et al(1999)Valle-Rai, Co, Gagnon, Hendren, Lam, and Sundaresan] on the *TypeInference* test program (listing 14, page 29). An example of an optimisation that has been performed is the replacement of the *dup* opcode (which duplicates the value on the top of the stack) with *load* and *store* instructions. Listing 24, page 36 shows the Jasmin source for the TypeInference test program and listing 25, page 37 shows the Jasmin souce for the Optimised test program. The optimised program's f method is 3 bytes smaller than in the original program.

Listing 23: Exception Test Program (Jasmin)

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

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

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

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

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

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

  return

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

.end method
```

Listing 24: Jasmin source for the TypeInference test program

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

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

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

.end method

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

.end method

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

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

.end method
```

Listing 25: Jasmin source for the Optimised (TypeInference) test program

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

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

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

.end method

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

.end method

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

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

.end method
```

Listing 26: Variable re-use test program

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

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


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

     iconst_0
     istore_0

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

     return

.end method
```

3.1.9 Args

Args (listing 26, page 38) 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*.

3.1.10 connectfour

The connectfour test program [Fagin and Carlisle(2005)] is an implementation of the game Connect Four originally written in Ada and compiled with JGNAT [jgn(2009)] 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.

3.2 Measuring the Effectiveness of Java Decompilers

The effectiveness of a Java decompiler, depends heavily on how the bytecode was produced. Arbitrary bytecode can contain instruction sequences for which there is no valid Java source due to the more powerful and less-restrictive nature of Java bytecode. For example there are no arbitrary control flow instructions in Java but there are in Java bytecode. In fact, many Java bytecode obfuscators rely on the fact that most decompilers fail when encountering unexpected, but valid, bytecode sequences [Batchelder and Hendren(2007)].

Naeem *et al.* [Naeem et al(2007)Naeem, Batchelder, and Hendren] suggest using software metrics [Halstead(1977), Kearney et al(1986)Kearney, Sedlmeyer, Thompson, Gray, and Adler] for measuring the effectiveness of decompilers. They compare the output of several decompilers against the input programs using software metrics. Software metrics are a good measure of the complexity of the output of a decompiler however they cannot quantify the effectiveness of the general output of a decompiler.

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

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

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

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

For each program, decompiler pair, we give a score between 0 and 9 (see Figure 14). A score of 0 is a perfect, or good, decompilation whereas 9 means that the decompiler failed and no output was produced. This is loosely based on Dyer's 1997 Java decompiler survey categories [Dyer(1997)].

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

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

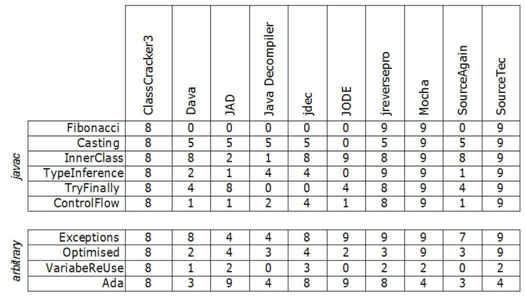
Category 9 indicates that a decompiler failed to produce any output at all, and most likely failed to parse the input file. Problems could occur due to arbitrary bytecode or the latest Java class files which decompilers are not expecting.

As the categories increase the difficulty to read and understand the code also increases, for example both category 0 and 1 are semantically correct but category 1 is

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

Fig. 14: Decompilation correctness classification

hard to read. Category 2 and higer programs are harder to understand as they are syntactically and/or semantically incorrect.



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

Fig. 15: Decompiler Test Results: Each decompiler was tested in different problem areas, with 6 test programs representing javac generated bytecode and 4 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.

3.3 Summary of Results

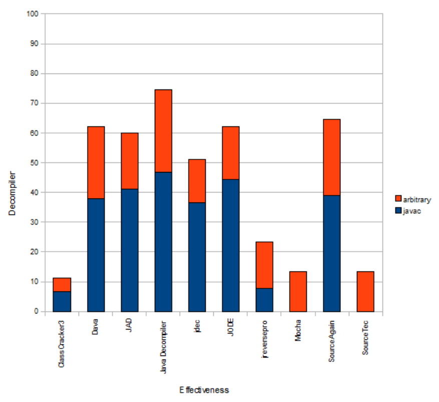
No decompiler was able to decompile all test programs with JODE decompiling the most programs correctly. JODE only managed to decompile 5 programs while four (unmaintained) decompilers could not decompile any of the test programs correctly. The top four decompilers (excluding obsolete SourceAgain) were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

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

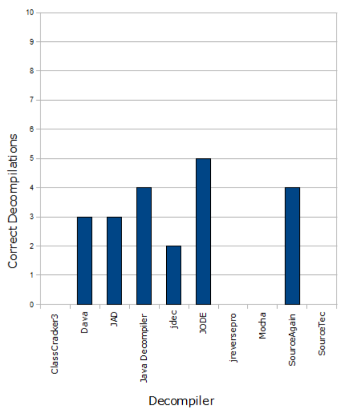
Java Decompiler scored the highest using our effectiveness measures, beating Jad and JODE by performing slightly better at decompiling the arbitrary bytecode programs. JODE was able to decompile one more program correctly than Java Decompiler. JODE was the best decompiler in the original survey and is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

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

Figure 15, page 41 shows a table of raw results detailing our effectivess score given to each decompiler for each program. Figures 16(a) and 16(b), page 42 show the the decompiler effectiveness scores. The charts show percentages of effectiveness with higher percentage meaning a higher effectiveness score, calculated from table 15, page 41.



(a) Decompiler Effectiveness



(b) Correct Decompilations

4 Discussion of Results

In this section we discuss the results of the tests, explaining the output of the decompilers when they failed to decompile a program. Full program listings of the decompiled programs are in appendix A, beginning on page 59.

4.1 ClassCracker3

ClassCracker3 did not decompile any of our test programs completely with just the method signatures in the resulting Java source file. The original survey found a similar result and the author concluded that the decompiler 'needs some work to cater for the tricky cases covered in these tests' [Emmerik(2003)].

4.2 Dava

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

4.2.1 Casting

The decompiled casting test program (Listing 28, page 59) indicates that Dava was unable to detect the need to insert a cast, and therefore Dava produced a semantically incorrect program. The decompiled program iterates through the characters as in the original program (except using Unicode instead of ASCII) but in the *System.out.println* invocation there is no cast inserted. The output of the program, when executed, is therefore two lists of characters rather than the original ASCII code followed by ASCII character.

4.2.2 Args

Dava correctly decompiled the Args test program (Listing 29, page 59) but typed the int local variable as byte, and inserted two unnecessary typecasts. We therefore classify this program as semantically equivalent but with 'untidy' source code.

4.2.3 ControlFlow

Dava produces a semantically equivalent ControlFlow program (Listing 30, page 60) which is slightly different from the original program. Dava's program contains a *labelled while loop*, and uses *labelled continue* statements to continue iterating from two points: inside the try section and inside the catch section. This version of the program has only one loop and uses a *labelled continue* to mimic the inner loop in the original program (Listing 21, page 32). Both Java class files contain the same bytecode sequences though they are generated from different Java source (Listing 22, page 33).

4.2.4 Sable

Dava correctly performs type inference on the Sable test program (Listing 14, page 29) but unfortunately fails to insert a typecast in the method invocation for method f, which results in a syntactically incorrect Java program (Listing 31, page 61). The main problem, type inference for variable d, was solved correctly and inserting a type cast results in a semantically and syntactically correct program.

4.2.5 Usa

Dava failed to correctly decompile the Usa test program resulting in a class file with out the inner classes (Listing 32, page 62). The resulting program contains none of the inner classes present in the original program (Listing 13, page 28).

4.2.6 Exceptions

Dava fails to correctly decompile the Exceptions test program, producing an incomplete program (Listing 27, page 59). This is surprising because the program was designed by the creators of Dava to demonstrate their decompiler [Miecznikowski and Hendren(2002)]. The previous version of Dava, 2.3.0, could decompile the program correctly and the result decreased the effectiveness score in this version.

4.2.7 TryFinally

The decompiled TryFinally program (Listing 33, page 62) is interesting as it decompiles a nearly syntactically, and somewhat semantically correct program. The decompiled program contains a small syntactic error (variable $\$r5$ was originally $\$r4$, which had already been declared) which when corrected produces a compiled Java program which produces an equivalent output to the original program. Even though both programs produce the same output they are not identical. The decompiled program does not make use of a try-finally block, and instead uses a mix of try-catch blocks and a labelled while-loop and a labelled continue statement. If re-compiled the program does not produce the same bytecode as the original program.

4.2.8 Optimised

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

4.2.9 Ada

Dava failed to correctly decompile the Ada test program with several syntactic and semantic errors. Listing 35, page 60 shows an extract from the decompiled program where a *boolean* is compared with an *int*.

4.3 Jad

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

4.3.1 Sable

Jad produced a semantically equivalent Java program (Listing 36, page 64) but did not perform type inference. Variable d (Obj in the decompiled program) is typed as $Object$, and an explicit typecast is inserted at the point where the $draw()$ method is invoked. We therefore classify this program as category 1 - semantically and syntactically correct, but no type inference was performed.

4.3.2 ControlFlow

Jad decompiles the ControlFlow program correctly (Listing 37, page 65) though produces slightly different Java source than the original. The program uses a for-loop instead of a while loop for the inner loop and a do-while loop instead of a standard while loop for the outer loop.

4.3.3 Casting

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

4.3.4 Usa

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

4.3.5 Exceptions

Jad fails to correctly decompile the Exceptions test program, producing a syntactically incorrect program (Listing 40, page 67). The program contains some illegal Java code, including *goto* statements and the the word ‘*this*’. The program also contains only one try-catch block which includes ‘c’ and ‘d’ in the try clause and ‘e’ in the catch clause. The program is syntactically incorrect and not equivalent to the original.

4.3.6 Optimised

The Optimised test program is incorrectly decompiled by Jad. The program (Listing 41, page 68) contains only one local variable in the f method compared with 4 in the original. The program also contains some bytecode instructions where Jad could not determine how the objects are constructed using the optimisations. Jad was expecting *javac* generated code so could not deal with the unexpected optimised sequence for object construction.

4.3.7 TryFinally

Jad produces a syntactically incorrect TryFinally program (Listing 42, page 69). The decompiled program contains illegal Java code as it had trouble analysing the control flow of the program.

4.3.8 Args

Jad produces a syntactically incorrect Java program (Listing 43, page 69). Jad attempts to assign an integer to an array of String objects, which is syntactically incorrect.

4.4 Java Decompiler

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

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

4.4.1 Casting

Java Decompiler produces a semantically incorrect program (Listing 45, page 70) in the same way that Dava does (Listing 28, page 59). The decompiled program loop iterates through the characters as in the original program (except using unicode instead of ASCII) but in the *System.out.println* invocation there is no cast inserted. The output of the program, when executed, is therefore two lists of characters rather than the original ASCII code followed by ASCII character

4.4.2 InnerClass

The InnerClass program is semantically correct but contains some redundant code (Listing 46.

4.4.3 Sable

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

4.4.4 ControlFlow

Java Decompiler produces a semantically incorrect control flow program (Listing 48, page 71) containing an extra break statement and some labels. If the syntactic errors are removed the program is almost semantically correct.

4.4.5 Exceptions

Java Decompiler fails to correctly decompile the Exceptions test program resulting in a syntactically incorrect program (Listing 49, page 72). The program uses the word 'this' as a variable name which is a reserved word in Java. Correcting these syntactic errors leads to a semantically incorrect program (see Figure 13, page 72).

4.4.6 Optimised

Java Decompiler produces a syntactically incorrect program with some interesting syntactic errors (Listing 50, page 72) in the same way as the TypeInference test program.

4.4.7 Args

Java Decompiler produces a semantically correct Java program (Listing 51, page 73).

4.4.8 connectfour

Java Decompiler produces a syntactically incorrect connectfour program with several syntactic errors. One such error is similar to the optimised test program which suggests the Ada-to-Java compiler uses an optimised form of object instantiation (Listing 52, page 70). Many of the methods which throw exceptions are missing the class name of the exception (Listing 53, page 70).

4.5 jdec

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

4.5.1 Casting

jdec decompiles the Casting test program with the same semantic error (Listing 54, page 74) as other decompilers such as Dava (Listing 28, page 59) - the crucial cast from *char* to *int* is missing. The decompiled program loop iterates through the characters as in the original program but in the *System.out.println* invocation there is no cast inserted. The output of the program, when executed, is therefore two lists of characters rather than the original ASCII code followed by ASCII character

4.5.2 Usa

jdec fails to decompile the Usa test program, decompiling only the main class (Listing 55, page 75). Dava also has this problem.

4.5.3 Sable

jdec produces a syntactically and semantically incorrect type inference test program, with corrected syntactic errors leading to a semantically incorrect program (Listing 56, page 76). This program contains two *Rectangle* objects rather than the one *Rectangle* original and one *Drawable* object in the original.

4.5.4 ControlFlow

jdec produces a syntactically incorrect control flow test program with a *catch* clause intersecting a *else* clause which, if corrected produces a semantically incorrect program (Listing 57, page 77).

4.5.5 Exceptions

jdec only partially decompiles the exceptions test program, with blocks ‘d’, ‘e’ and ‘f’ missing. The program is syntactically and semantically incorrect (Listing 58, page 78).

4.5.6 Optimised

jdec produces a syntactically and semantically incorrect optimised program, with several syntactic errors (Listing 59, page 79). The program contains variable assignment statements within constructor invocation parameters and uses the variable *this* in many places. The method is static so should contain no use of the *this* keyword. The method also contains no arguments which is strange because jdec introduced a variable called $arg0$.

4.5.7 Args

jdec produces a syntactically incorrect program (Listing 60, page 80), in a similar way to Jad by attempting to assign an integer to a string array. jdec also attempts to re-declare the method argument.

4.5.8 connectfour

jdec produces a seemingly incomplete, and syntactically incorrect, program with many empty blocks (for example, listing 63, page 81) and syntax errors. The program has several examples of missing commas within method parameter lists (for example, listing 62, page 73) and assignment statements within array declarations (for example, listing 61, page 81).

4.6 JODE

JODE has a similar overall result to Java Decompiler and Jad but is beaten using our effectiveness measures by Java Decompiler. In comparison to Java Decompiler and Jad, JODE is able to decompile 5 test programs perfectly whereas Java Decompiler and Jad only decompile 3 test programs correctly. Surprisingly JODE is unable to correctly decompile the TryFinally test program. JODE was the best decompiler in the original survey and is still one of the best in our evaluation.

4.6.1 TryFinally

JODE's decompilation of the TryFinally test program (Listing 64, page 81) fails to include a try-finally block but instead includes a try-catch block and an incorrectly typed exception variable. The catch clause includes an exception variable typed as *Object* but such a variable must be typed *Exception* or a sub-class of *Exception*. This makes the decompiled program both syntactically and semantically incorrect. The finally clause code is duplicated after the try-catch block, and if the syntactic error is corrected the program produces the same output as the original. However, this is not the same as the original program which uses a try-finally block rather than a try-catch block.

4.6.2 Usa

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

4.6.3 ControlFlow

JODE produces a semantically correct decompilation (Listing 65, page 81) of the control flow test program (Listing 21, page 32). However, the source is more obtuse than the original. JODE's decompilation uses for-loops instead of while loops which, while semantically equivalent, do not look as user-friendly as the original. In this version of the program there is no *continue* statement in the catch clause but instead a break is used in the try clause. Control flows from the catch clause to the beginning of the loop as there is no break clause.

4.6.4 Exceptions

JODE could not decompile the exceptions test program, exiting with an exception regarding the intersecting try-catch blocks (Listing 67, page 82).

4.6.5 Optimised

JODE has a slight problem with object construction in the optimised test program, similar to other decompilers such as Java Decompiler. If the constructor problem is corrected the program is semantically correct, including the correctly typed variable (Listing 67, page 82).

4.6.6 Ada

JODE could not parse the Ada test program and exits with an exception (Listing 68, page 83), due to the arbitrary nature of the bytecode instructions.

4.7 jReversePro

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

4.7.1 Fibo

jReversePro is unable to decompile the trivial Fibo test program as it could not parse the latest class file format (Listing 69, page 83).

4.7.2 Casting

jReversePro was able to parse the simpler casting test program but produced a semantically incorrect program (Listing 70, page 84). The program fails to include the crucial cast in the same way as other decompilers such as Jad and Java Decompiler but also contains an infinite loop due to the inclusion of an extra variable.

4.7.3 Usa

The inner class program, as decompiled by jReversePro, contains none of the inner classes (Listing 71, page 85).

4.7.4 Sable

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

4.7.5 TryFinally

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

4.7.6 Exceptions

jReversePro exits with an exception due to intersecting try-catch blocks (Listing 73, page 86), like JODE.

4.7.7 Optimised

jReversePro was able to parse the optimised program, most likely because the optimiser output an earlier version class file format, but produced a syntactically incorrect program (Listing 74, page 87). The program contains several syntactic errors, included attempting to assign a boolean to an int and constructor problems.

4.7.8 Args

jReversePro incorrectly decompiled the variable re-use test program producing a syntactically incorrect program (Listing 75, page 88), similar to Jad and Java Decompiler.

4.7.9 connectfour

jReversePro produced a syntactically incorrect and incomplete connectfour test program. jReversePro has many blocks of missing code (Listing 76, page 84), like jdec (Listing 63, page 81) and contains many syntactic errors such as incomplete if-then-else blocks appearing within loops (Listing 77, page 85).

4.8 Mocha

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

4.8.1 Args

Mocha fails to produce a syntactically correct program, producing a similar program to other decompilers such as jReversePro and Jad, which attempts to assign an int to an array of strings (Listing 78, page 88).

4.8.2 Exceptions

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

4.8.3 connectfour

Mocha's decompilation of the connectfour test program contains many syntactic errors (for example, listing 79, page 85), and some bytecode instructions are left in place (for example, listing 81, page 90 and listing 80, page 89).

4.9 SourceAgain

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

4.9.1 Casting

SourceAgain, like other decompilers such as Jad, fails to insert the crucial cast resulting in a semantically incorrect program (Listing 82, page 90).

4.9.2 Usa

SourceAgain fails to correctly decompile (Listing 71, page 85) the inner class test program but unlike other decompilers, such as jReversePro, this is because the decompiler is web-based and only accepts single class files - the inner classes could not be uploaded along with the main class. The original survey [Emmerik(2003)] used the professional version of the decompiler which was able to correctly decompile inner classes. This is no longer available.

4.9.3 Sable

SourceAgain produces a semantically equivalent type inference program but could not correctly type the variable *d* (Listing 84, page 91). The online decompiler warns that it could not determine the inheritance relationship between the objects as a result of the restriction of decompiling single class files. In the original survey [Emmerik(2003)] the professional version of SourceAgain did not correctly type the variable.

4.9.4 TryFinally

SourceAgain produces a syntactically incorrect try finally test program containing two try-finally blocks, duplicated finally clause code and an undeclared variable (Listing 85, page 92).

4.9.5 ControlFlow

SourceAgain correctly decompiles the control flow program producing a semanticaly equivalent, though some-what obtuse, program (Listing 86, page 92). SourceAgain produces a very similar output to Dava (Listing 30, page 60) which is slightly different from the original program. Both Java class files contain the same bytecode sequences though they are generated from different Java source (Listing 22, page 33).

4.9.6 Exceptions

SourceAgain produces a semantically incorrect exceptions test program (Listing 87, page 93) which contains all the necessary blocks, unlike other decompilers, but is not semantically equivalent to the original. For example, block 'd' should be contained within a catch clause, leading to block 'e' if an exception is thrown.

4.9.7 Optimised

SourceAgain produces a syntactically incorrect program (Listing 88, page 94) containing, like other decompilers, object initialisation and constructor problems. SourceAgain also includes self assignments which aren't necessary.

4.9.8 connectfour

SourceAgain produces a syntactically incorrect connectfour test program but with fewer errors than other decompilers. If problems such as misnamed *this* variables (Listing 89, page 90) and multiple variable declarations (Listing 90, page 91) are removed the program becomes compilable and semantically correct. SourceAgain also inserts a large number of warning notices into the source which must be removed as they begin with a #.

4.10 SourceTec (Jasmine)

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

5 Conclusion and Future Work

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

Decompilation has many uses in the real world, such as the recovery of lost source code for a crucial application [Emmerik and Waddington(2004)], 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 [Miecznikowski(2003), Naeem and Hendren(2006),Naeem(2007),Miecznikowski and Hendren(2001),Miecznikowski and Hendren(2002)] decompiler, part of the Soot Optimisation Framework [Valle-Rai et al(1999)Valle-Rai, Co, Gagnon, Hendren, Lam, and Sundaresan], which is a research project carried out by the Sable Research Group at McGill University. Dava differs from other decompilers in that it aims to decompile arbitrary bytecode whereas other decompilers rely on known patterns produced by Java compilers (and this is usually *javac*). Dava is better at decompiling arbitrary bytecode whereas other decompilers are better at decompiling *javac* generated bytecode. However, Java Decompiler was just as good at decompiling arbitrary bytecode according to our effectivess score. Java Decompiler is aimed at decompiling *javac* generated bytecode.

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

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

Though there is a lack of commercial Java decompilers Java bytecode decompilation and decompilation in general are fruitful research areas. One of the main areas for research is type inference both in bytecode (e.g. [Bellamy et al(2008)Bellamy, Avgustinov, de Moor, and Sereni, Miecznikowski and Hendren(2001), Knoblock and Rehof(2001)]) and machine code (e.g. [Mycroft(1999)]). 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 [Gagnon et al(2000)Gagnon, Hendren, and Marceau], 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 [Bellamy et al(2008)Bellamy, Avgustinov, de Moor, and Sereni].

All the decompilers tested had some problems decompiling some of the tests. In terms of our effectiveness measures, Dava, Jad, Java Decompiler and JODE were the four best decompilers (excluding SourceAgain). Of these, JODE and Java Decompiler are the best decompilers: JODE correctly decompiles 5 out of the 10 test programs correctly and Java Decompiler performs best using our effectiveness measures but decompiles one less program correctly. JODE is open-source and is therefore open to further improvements but is not currently maintained, while Java Decompiler is in development and available free (but is not open-source). Unfortunately Jad is unmaintained and so is not guaranteed to work for future versions of Java class files. The commercial decompiler SourceAgain performs well in the effectiveness measures, but was only able to decompile 4 programs correctly. SourceAgain performed similarly to Dava but is now obsolete and only available as a web application which can decompile single class files. Java Decompiler is a newer decompiler in active development, which performs highest in our effectiveness measures and correctly decompiles 4 out of 10 programs. This decompiler performs best at *javac* generated bytecode and may improve in the future even more as it is in development.

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

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

We proposed a system to quantify the effectiveness of a decompiler, however the analysis is some-what subjective and could be further formalised.

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

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

References

[moc(1996)] (1996) Mocha, the java decompiler. URL `http://www.brouhaha.com/~eric/computers/mocha.html`

[sou(1997)] (1997) SourceTec (Jasmine). URL `http://www.sothink.com/product/javadecompiler/index.htm`

[jik(2005)] (2005) Jikes. URL `http://jikes.sourceforge.net/`

[jgn(2009)] (2009) JGNAT. URL `http://code.google.com/p/jgnat/`

[Alliance(2008)] Alliance BS (2008) Sixth annual BSA and IDC global software piracy study. Tech. Rep. 6, Business Software Alliance

[Alpern et al(1988)Alpern, Wegman, and Zadeck] Alpern B, Wegman MN, Zadeck FK (1988) Detecting equality of variables in programs. In: POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, p 111, DOI http://doi.acm.org/10.1145/73560.73561

[Batchelder and Hendren(2007)] Batchelder M, Hendren L (2007) Obfuscating java: the most pain for the least gain . Braga, Portugal

[Bellamy et al(2008)Bellamy, Avgustinov, de Moor, and Sereni] Bellamy B, Avgustinov P, de Moor O, Sereni D (2008) Efficient local type inference. SIGPLAN Not 43(10):475492, DOI http://doi.acm.org/10.1145/1449955.1449802

[Belur and Bettadapura(2008)] Belur S, Bettadapura K (2008) Jdec: Java decompiler. URL `http://jdec.sourceforge.net/`, 2008

[Buckley et al(2006)Buckley, Rose, Coglio, Corporation, Sun Microsystems, Tmax Soft, Technologies, and AG] Buckley A, Rose E, Coglio A, Corporation BS, Sun Microsystems I, Tmax Soft I, Technologies S, AG E (2006) JSR 202: JavaTM class file specification update. URL `http://jcp.org/en/jsr/detail?id=202`

[Caudle(1980)] Caudle W (1980) On inverse of compiling. Sperry-UNIVAC URL `http://www.program-transformation.org/view/Transform/OnInverseOfCompiling?skin=print.pattern`

[Cifuentes(1994)] Cifuentes C (1994) Reverse compilation techniques. PhD thesis, Queensland University of Technology, URL `http://citeseer.ist.psu.edu/cifuentes94reverse.html`

[Collberg and Nagra(2009)] Collberg C, Nagra J (2009) Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley Professional

[Collberg and Thomborson(1998)] Collberg C, Thomborson C (1998) On the limits of software watermarking. Tech. Rep. 164, URL `http://www.cs.auckland.ac.nz/collberg/Research/Publications/CollbergThomborson98e/index.html`, http://www.cs.auckland.ac.nz/~collberg/Research/Publications/CollbergThomborson98e/index.html

[Collberg et al(1997)Collberg, Thomborson, and Low] Collberg C, Thomborson C, Low D (1997) A taxonomy of obfuscating transformations. Tech. Rep. 148, URL `<ahref="http://www.cs.auckland.ac.nz/collberg/Research/Publications/CollbergThomborsonLow97a/index.html">Collberg97a</a>`, http://www.cs.auckland.ac.nz/$\sim$collberg/Research/Publications/CollbergThomborsonLow97a/index.html

[Collberg and Thomborson(2002)] Collberg CS, Thomborson C (2002) Watermarking, Tamper-Proofing, and obfuscation - tools for software protection. In: IEEE Transactions on Software Engineering, vol 28, p 735746, URL `http://citeseer.nj.nec.com/collberg02watermarking.html`

[Cytron et al(1989)Cytron, Ferrante, Rosen, Wegman, and Zadeck] Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK (1989) 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, ACM, New York, NY, USA, p 2535, DOI http://doi.acm.org/10.1145/75277.75280

[Dagenais and Hendren(2008)] Dagenais B, Hendren L (2008) Enabling static analysis for partial java programs. SIGPLAN Not 43(10):313328, DOI http://doi.acm.org/10.1145/1449955.1449790

[Dupuy(2009)] Dupuy E (2009) Java decompiler. URL `http://java.decompiler.free.fr/`

[Dyer(1997)] Dyer D (1997) Java decompilers compared. URL `http://www.javaworld.com/javaworld/jw-07-1997/jw-07-decompilers.html`

[Emmerik(2003)] Emmerik MV (2003) Java decompiler tests. http://www.program-transformation.org/Transform/JavaDecompilerTests, URL `http://www.program-transformation.org/Transform/JavaDecompilerTests`

[Emmerik(2007)] Emmerik MV (2007) Static single assignment for decompilation. PhD thesis, The University of Queensland

[Emmerik and Waddington(2004)] Emmerik MV, Waddington T (2004) Using a decompiler for Real-World source recovery. In: WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, p 2736

[Fagin and Carlisle(2005)] Fagin B, Carlisle M (2005) Connect Four(TM) game, written in ada. URL `http://webdiis.unizar.es/asignaturas/EDA/gnat/jgnat/connect_four/test.html`, 2005

[Freund(1998)] Freund SN (1998) The costs and benefits of java bytecode subroutines. In: In Formal Underpinnings of Java Workshop at OOPSLA

[Gagnon et al(2000)Gagnon, Hendren, and Marceau] Gagnon E, Hendren LJ, Marceau G (2000) Efficient inference of static types for java bytecode. In: Static Analysis Symposium, p 199219, URL `www.sable.mcgill.ca/publications`

[Gosling et al(2000)Gosling, Joy, Steele, and Bracha] Gosling J, Joy B, Steele G, Bracha G (2000) The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass., URL `citeseer.ist.psu.edu/gosling00java.html`

[Gosling et al(2005)Gosling, Joy, Steele, and Bracha] Gosling J, Joy B, Steele G, Bracha G (2005) Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)). Addison-Wesley Professional

[Gough and Corney(2001)] Gough KJ, Corney D (2001) Implementing languages other than java on the java virtual machine

[Grishchenko and Gyger(2009)] Grishchenko V, Gyger J (2009) JadClipse. URL `http://jadclipse.sourceforge.net/wiki/index.php/Main_Page`

[Haggar(2001)] Haggar P (2001) Understanding bytecode makes you a better programmer. http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/, URL `http://www.ibm.com/developerworks/ibm/library/it-haggar_bytecode/`

[Halstead(1977)] Halstead MH (1977) Elements of software science (Operating and programming systems series). Elsevier, published: Hardcover

[Hamilton and Danicic(2009)] Hamilton J, Danicic S (2009) An evaluation of current java bytecode decompilers. In: Ninth IEEE International Workshop on Source Code Analysis and Manipulation, IEEE Computer Society, Edmonton, Alberta, Canada, vol 0, pp 129–136, DOI http://doi.ieeecomputersociety.org/10.1109/SCAM.2009.24

[Hoenicke(2004)] Hoenicke J (2004) JODE. URL `http://jode.sourceforge.net/`, 2004

[Janssen and Corporaal(1997)] Janssen J, Corporaal H (1997) Making graphs reducible with controlled node splitting. ACM Trans Program Lang Syst 19(6):10311052, DOI http://doi.acm.org/10.1145/267959.269971

[Kearney et al(1986)Kearney, Sedlmeyer, Thompson, Gray, and Adler] Kearney, Sedlmeyer, Thompson, Gray, Adler (1986) Software complexity measurement. Commun ACM 29(11):10441050, DOI http://doi.acm.org/10.1145/7538.7540

[Knoblock and Rehof(2001)] Knoblock TB, Rehof J (2001) Type elaboration and subtype completion for java bytecode. ACM Trans Program Lang Syst 23(2):243272, DOI http://doi.acm.org/10.1145/383043.383045

[Kouznetsov(2001)] Kouznetsov P (2001) Jad - the fast java decompiler. URL `http://www.varaneckas.com/jad`

[Kramer et al(1996)Kramer, Joy, and Spenhoff] Kramer D, Joy B, Spenhoff D (1996) The java[tm] platform. Tech. rep., Sun Microsystems, URL `http://java.sun.com/docs/white/platform/javaplatformTOC.doc.html`

[Kumar(2005)] Kumar K (2005) JReversePro - java decompiler / disassembler. URL `http://jreversepro.blogspot.com`

[Ladue(1997)] Ladue MD (1997) When java was one: Threats from hostile byte code. In: Proceedings of the 20th National Information Systems Security Conference

[Leroy(2003)] Leroy X (2003) Java bytecode verification: Algorithms and formalizations. J Autom Reason 30(3-4):235269

[Lindholm and Yellin(1999)] Lindholm T, Yellin F (1999) The Java(TM) Virtual Machine Specification (2nd Edition). Prentice Hall PTR, published: Paperback

[Masnick(2008)] Masnick M (2008) A detailed explanation of how the BSA misleads with piracy stats | techdirt. http://www.techdirt.com/articles/20080718/1226541724.shtml, URL `http://www.techdirt.com/articles/20080718/1226541724.shtml`

[Meyer et al(2004)Meyer, Reynaud, Kharon et al] Meyer J, Reynaud D, Kharon I, et al (2004) Jasmin. URL `http://jasmin.sourceforge.net/`

[Miecznikowski(2003)] Miecznikowski J (2003) New algorithms for a java decompiler and their implementation in soot. Masters thesis, McGill University

[Miecznikowski and Hendren(2001)] Miecznikowski J, Hendren L (2001) Decompiling java using staged encapsulation. In: WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01), IEEE Computer Society, Washington, DC, USA, p 368

[Miecznikowski and Hendren(2002)] Miecznikowski J, Hendren LJ (2002) Decompiling java bytecode: Problems, traps and pitfalls. In: CC '02: Proceedings of the 11th International Conference on Compiler Construction, Springer-Verlag, London, UK, p 111127

[Mycroft(1999)] Mycroft A (1999) Type-Based decompilation (or program reconstruction via type reconstruction). In: ESOP '99: Proceedings of the 8th European Symposium on Programming Languages and Systems, Springer-Verlag, London, UK, p 208223

[Naeem(2007)] Naeem NA (2007) Programmer-Friendly decompiled java. Masters thesis, URL `http://www.sable.mcgill.ca/publications/thesis/#nomairMastersThesis`

[Naeem and Hendren(2006)] Naeem NA, Hendren L (2006) Programmer-Friendly decompiled java. In: ICPC '06: Proceedings of the 14th IEEE International Conference onProgram Comprehension (ICPC'06), IEEE Computer Society, p 327336, DOI http://dx.doi.org/10.1109/ICPC.2006.20

[Naeem et al(2007)Naeem, Batchelder, and Hendren] Naeem NA, Batchelder M, Hendren L (2007) Metrics for measuring the effectiveness of decompilers and obfuscators. In: ICPC '07: Proceedings of the 15th IEEE International Conference on Program Comprehension, IEEE Computer Society, Washington, DC, USA, p 253258, DOI http://dx.doi.org/10.1109/ICPC.2007.27

[Nolan(2004)] Nolan G (2004) Decompiling Java. APress

[Pearson(2009)] Pearson M (2009) Piracy: BSA's $53b global piracy tab grossly inflated, argue skeptics. http://www.ecommercetimes.com/story/67049.html?wlc=1252256745, URL `http://www.ecommercetimes.com/story/67049.html?wlc=1252256745`

[Proebsting and Watterson(1997)] Proebsting TA, Watterson SA (1997) Krakatoa: Decompilation in java (Does bytecode reveal source?). p 185197, URL `http://citeseer.ist.psu.edu/10614.html`

[Ramshaw(1988)] Ramshaw L (1988) Eliminating go to's while preserving program structure. J ACM 35(4):893920, DOI http://doi.acm.org/10.1145/48014.48021

[Research(2005)] Research MS (2005) ClassCracker 3. URL `http://mayon.actewagl.net.au/`

[Rose(1998)] Rose E (1998) Lightweight bytecode verification

[Rosen et al(1988)Rosen, Wegman, and Zadeck] Rosen BK, Wegman MN, Zadeck FK (1988) Global value numbers and redundant computations. In: POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, New York, NY, USA, p 1227, DOI http://doi.acm.org/10.1145/73560.73562

[Software(2004)] Software A (2004) SourceAgain. URL `http://www.ahpah.com/cgi-bin/suid/~pah/demo_license.cgi`, year = 2004

[Strk et al(2001)Strk, Schmid, and Brger] Strk RF, Schmid J, Brger E (2001) Java and the Java Virtual Machine: Definition, Verification and Validation. Springer-Verlag

[Tolksdorf(2010)] Tolksdorf R (2010) Programming languages for the java virtual machine JVM. http://www.is-research.de/info/vmlanguages/index.html, URL `http://www.is-research.de/info/vmlanguages/index.html`

[Valle-Rai et al(1999)Valle-Rai, Co, Gagnon, Hendren, Lam, and Sundaresan] Valle-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V (1999) Soot - a java bytecode optimization framework. In: CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research, IBM Press, p 13

[Venners(1996)] Venners B (1996) Inside the Java Virtual Machine. McGraw-Hill, Inc., New York, NY, USA

A Decompiled Program Listings

Listing 27: Test Result: Dava - Exceptions test program

```
import java.io.PrintStream;

class Exceptions2
{

    public static void main(String[]  r0)
    {

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

    public void Exceptions2()
    {


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

Listing 28: Test Result: Dava - Casting test program

```
import java.io.PrintStream;

public class Casting
{

    public static void main(String[]  r0)
    {

        char c0;
        for (c0 = '\u0000'; c0 < '\u0080'; c0 = (char) (c0 + 1))
        {
            System.out.println((new StringBuilder()).append("ascii ").
                 append(c0).append(" character ").append(c0).toString());
        }
    }
}
```

Listing 29: Test Result: Dava - Args test program

```
import java.io.PrintStream;

class Args
{

    public static void main(String[]  r0)
    {

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

Listing 30: Test Result: Dava - ControlFlow test program

```
import java.io.PrintStream;

public class ControlFlow
{

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

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

            return i1;
        }
    }

    public static void main(String[]  r0)
    {


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

Listing 35: Test Result: Dava - Extract 1 from connectfour test program

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

Listing 31: Test Result: Dava - Sable test program

```
public class Sable
{

    public static void f(short  s0)
    {

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

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


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

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

    }

    public static void main(String[]  r0)
    {


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

Listing 32: Test Result: Dava - Usa test program

```
public class Usa
{
    public String name;

    private final void this()
    {


        name = "Detroit";
    }

    public Usa()
    {

        this.this();
    }
}
```

Listing 33: Test Result: Dava - TryFinally test program. Variable $\$r5$ was originally $\$r4$.

```
import java.io.PrintStream;

public class TryFinally
{

    public static void main(String[]  r0)
    {

        Throwable r2;
        try
        {
            System.out.println("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");
    }
}
```

Listing 34: Test Result: Dava -Optimised test program

```
public class Optimised
{

    public static void f(short  s0)
    {

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

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


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

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

    }

    public static void main(String[]  r0)
    {


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

Listing 36: Test Result: Jad - Sable test program

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


public class Sable
{

    public Sable()
    {
    }

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

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

Listing 44: Test Result: Jad - Extract 1 from connectfour test program

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

Listing 37: Test Result: Jad - ControlFlow test program

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

import java.io.PrintStream;

public class ControlFlow
{

    public ControlFlow()
    {
    }

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

Listing 38: Test Result: Jad - Casting test program

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

import java.io.PrintStream;

public class Casting
{

    public Casting()
    {
    }

    public static void main(String args[])
    {
        for(char c = '\0'; c < 128; c++)
            System.out.println((new StringBuilder()).append("ascii ").
                 append(c).append(" character ").append(c).toString());

    }
}
```

Listing 39: Test Result: jad - Usa test program

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

import java.io.PrintStream;

public class Usa
{
    public class England
    {
        public class Ireland
        {

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

            public String name;
            final England this$1;

            public Ireland()
            {
                this$1 = England.this;
                super();
                name = "Dublin";
            }
        }


        public String name;
        final Usa this$0;

        public England()
        {
            this$0 = Usa.this;
            super();
            name = "London";
        }
    }


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

    public String name;
}
```

Listing 40: Test Result: Jad - Exceptions test program

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)
// Source File Name:   Exceptions2.jasmin

import java.io.PrintStream;

class Exceptions
{

    Exceptions()
    {
    }

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

    public void Exceptions()
    {
        System.out.println("a");
        System.out.println("b");
        try
        {
            System.out.println("c");
            System.out.println("d");
        }
        // Misplaced declaration of an exception variable
        catch(Exceptions this)
        {
            System.out.println("e");
        }
_L2:
        System.out.println("f");
        return;
        this;
        System.out.println("g");
        if(true) goto _L2; else goto _L1
_L1:
    }
}
```

Listing 41: Test Result: Jad - Optimised test program

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


public class Optimised
{

    public Optimised()
    {
    }

    public static void f(short arg0)
    {
        Object obj;
        if(arg0 > 10)
        {
            obj = JVM INSTR new #43  <Class Rectangle>;
            ((Rectangle) (obj)).Rectangle(arg0, arg0);
            arg0 = ((Rectangle) (obj)).isFat();
            obj = obj;
        } else
        {
            obj = JVM INSTR new #19  <Class Circle>;
            ((Circle) (obj)).Circle(arg0);
            arg0 = ((Circle) (obj)).isFat();
            obj = obj;
        }
        if(arg0 == 0)
            ((Drawable) (obj)).draw();
    }

    public static void main(String arg0[])
    {
        f((short)11);
    }
}
```

Listing 42: Test Result: Jad - TryFinally test program

```
// Decompiled by Jad v1.5.8e. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://www.geocities.com/kpdus/jad.html
// Decompiler options: packimports(3)
// Source File Name:   TryFinally.java
//Overlapped try statements detected. Not all exception handlers will be
    resolved in the method main
//Couldn't fully decompile method main
//Couldn't resolve all exception handlers in method main

import java.io.PrintStream;

public class TryFinally
{

    public TryFinally()
    {
    }

    public static void main(String args[])
    {
        System.out.println("try");
        System.out.println("finally");
        break MISSING_BLOCK_LABEL_30;
        Exception exception;
        exception;
        System.out.println("finally");
        throw exception;
    }
}
```

Listing 43: Test Result: Jad - Args test program

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

import java.io.PrintStream;

class Args
{

    Args()
    {
    }

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

Listing 45: Test Result: Java Decompiler - Casting test program

```
import java.io.PrintStream;

public class Casting
{
  public static void main(String[] paramArrayOfString)
  {
    for (char c = '\000'; c < '?'; c = (char)(c + '\001'))
      System.out.println("ascii " + c + " character " + c);
  }
}
```

Listing 46: Test Result: Java Decompiler - InnerClass test program

```
import java.io.PrintStream;
public class Usa {
  public String name;
  private final void jdMethod_this() {
    this.name = "Detroit";
  }
  public Usa() {
    jdMethod_this();
  }
  public class England { public String name;
    private final void jdMethod_this() { this.name = "London"; }
    public England() {
      jdMethod_this();
    }
    public class Ireland {
      public String name;
      public void print_names() { System.out.println(this.name); }
      private final void jdMethod_this() {
        this.name = "Dublin";
      }
      public Ireland() {
        jdMethod_this();
      }
    }
  }
}
```

Listing 52: Test Result: Java Decompiler - Extract 1 from connectfour test program

```
standard.access_string tmp10_7 = new jgnat/adalib/standard$access_string;
tmp10_7.<init>();
```

Listing 53: Test Result: Java Decompiler - Extract 2 from connectfour test program

```
public static void _elabb() throws {
```

Listing 47: Test Result: Java Decompiler - Sable test program

```
public class Sable {
  public static void f(short paramShort) {
    Object localObject;
    if (paramShort > 10) {
    localObject = new Rectangle;
      ((Rectangle)localObject).<init>(paramShort, paramShort);
      paramShort = ((Rectangle)localObject).isFat();
      localObject = localObject;
    } else {
      localObject = new Circle;
      ((Circle)localObject).<init>(paramShort);
      paramShort = ((Circle)localObject).isFat();
      localObject = localObject;
    }
    if (paramShort != 0)
      return;
    ((Drawable)localObject).draw();
  }

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

Listing 48: Test Result: Java Decompiler - ControlFlow test program

```
public class ControlFlow {
  public static int foo(int paramInt1, int paramInt2) {
    while (true) {
      try {
        while (paramInt1 < paramInt2)
          paramInt1 = paramInt2++ / paramInt1;
      } catch (RuntimeException localRuntimeException) {
        paramInt1 = 10;
        break label35:
      }
      label35: break;
    }
    return paramInt2;
  }

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

Listing 61: Test Result: jdec - Extract 1 from connectfour test program

```
int[] JdecGenerated173 = new int[]{    _aggr =(JdecGenerated173)
     ;71,143,214,286,357,429,500};
```

Listing 49: Test Result: Java Decompiler - Exceptions test program

```
import java.io.PrintStream;

class Exceptions {
  public static void main(String[] paramArrayOfString) {
    new Exceptions();
  }

  public void Exceptions() {
    System.out.println("a");
    try {
      System.out.println("b");
    } catch (java.lang.RuntimeException this) {
      try {
        System.out.println("c");
        System.out.println("d");
        label32: System.out.println("f");
        return;
      } catch (java.lang.Exception this) {
        System.out.println("e");
        break label32:
        this = this;
        System.out.println("g");
      }
    }
  }
}
```

Listing 50: Test Result: Java Decompiler - Optmised test program

```
public class Optimised
{
  public static void f(short arg0)
  {
    Object localObject;
    if (arg0 > 10)
    {
      localObject = new Rectangle;
      ((Rectangle)localObject).<init>(arg0, arg0);
      arg0 = ((Rectangle)localObject).isFat();
      localObject = localObject;
    }
    else
    {
      localObject = new Circle;
      ((Circle)localObject).<init>(arg0);
      arg0 = ((Circle)localObject).isFat();
      localObject = localObject;
    }
    if (arg0 == 0)
      ((Drawable)localObject).draw();
  }

  public static void main(String[] arg0)
  {
    f(11);
  }
}
```

Listing 51: Test Result: Java Decompiler - Args test program

```
import java.io.PrintStream;

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

Listing 62: Test Result: jdec - Extract 3 from connectfour test program

```
 public  static   int[][] connectfour$Tboard_array_typeB_deep_copy(  int
      [][] _target,  int _trgstart,  int [][] _source,  int _srccount,
     int _srcstart  int [][] _result  int _loop_index  int _trgindex  int
      _srcindex  int [][] _trg_tmp  int [][] _src_tmp  int _loop_index
     int _index_max) { ... }

 public static void initialize_board(  int [][] board  int _loop_param
     int _loop_limit  int _loop_param  int _loop_limit  Throwable
     _exc_var) { ... }

 public static void place_disk(  int [][] board,  int column,  Int row,
     int who  Throwable _exc_var) { ... }

 public static void check_won(  int [][] board,  int who,  Int won  int
     _loop_param  int _loop_limit  int _loop_param  int _loop_limit
     Throwable _exc_var) { ... }

 public static void check_tie(  int [][] board,  Int is_tie  int
     _loop_param  int _loop_limit  Throwable _exc_var) { ... }

 public static void computer_turn(  int [][] board,  Int column
     standard$access_string [] value_typeT  standard$access_string
     _alloc_tmp  byte [] _str_literal  byte [] _str_literal  byte []
     _str_literal  byte [] _str_literal  byte [] _str_literal  byte []
     _str_literal  __AR_connectfour$computer_turn __AR  int [][]
     new_board  int [] evaluations  int [] moves_to_unknown  int
     count_unknowns  int value  int max_value  int best_move  int
     _loop_param  int _loop_limit  int _loop_param  int _loop_limit  int
     _loop_param  int _loop_limit  Int _out_tmp  int R41b  int
     _loop_param  int _loop_limit  Throwable _exc_var) { ... }
```

Listing 54: Test Result: jdec - Casting test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File 'COPYING' For more details.

/**** List of All Imported Classes ***/

import java.io.PrintStream;
import java.lang.Object;
import java.lang.StringBuilder;
import java.lang.System;

// End of Import

public  class  Casting

{


//  CLASS: Casting:
 public     Casting( )
 {
    super();
    return;

 }

//  CLASS: Casting:
 public  static    void main(  String [] aString1)
 {
    char char1= 0;
    char1=0;
    while(true)
    {
      if(char1 >= 128)
     {
        break;
      }
      if(char1 < 128)
      {
        StringBuilder JdecGenerated14 = new StringBuilder();
        System.out.println(JdecGenerated14.append("ascii ").append(char1)
             .append(" character ").append(char1).toString());
        char1=(char)((char1 + 1));
        continue  ;

      }

    }
    return;

 }


}
```

Listing 55: Test Result: jdec - Usa test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File 'COPYING' For more details.

/**** List of All Imported Classes ***/

import java.lang.Object;

// End of Import

public class Usa {

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

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

Listing 56: Test Result: jdec - Sable test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File 'COPYING' For more details.
// class file compiled with jikes, javac wouldn't decompile
/**** List of All Imported Classes ***/

import java.lang.Object;

// End of Import

public class Sable {
//  CLASS: Sable:
 public Sable() {
    super();
    return;
 }

//  CLASS: Sable:
 public static void f(short short2) {
    Circle aCircle1= null;
    Rectangle aRectangle1= null;
    Rectangle aRectangle2= null;

    int int1= 0;
    if(short2 > 10) {
      Rectangle JdecGenerated8 = new Rectangle(short2,short2);
      aRectangle1=JdecGenerated8;
      int1=aRectangle1.isFat();
      aRectangle2=aRectangle1;
    }else{
      Circle JdecGenerated29 = new Circle(short2);
      aCircle1=JdecGenerated29;
      int1=aCircle1.isFat();
      aCircle2 =aCircle1;
    }
    if(int1==0) {
      aCircle2.draw();
      return ;
    }
 }

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

Listing 57: Test Result: jdec - ControlFlow test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File 'COPYING' For more details.

/**** List of All Imported Classes ***/

import java.lang.Object;
import java.lang.RuntimeException;

// End of Import

public class ControlFlow {

//  CLASS: ControlFlow:
 public ControlFlow() {
    super();
    return;
 }

//  CLASS: ControlFlow:
 public  int foo(int int4,  int int5) {
    int int4= 0;
    while(true) {
      try {
    if(int4 < int5) {
      int4 = (int5++) / (int4);
      continue  ;
    }else{

        }catch(RuntimeException  aRuntimeException1) {

          }
        int4=10;
        continue  ;

      }

    }
    return int5;
 }
}
```

Listing 58: Test Result: jdec - Exceptions test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File 'COPYING' For more details.

/**** List of All Imported Classes ***/

import java.io.PrintStream;
import java.lang.Object;
import java.lang.RuntimeException;
import java.lang.System;

// End of Import

class  Exceptions2

{


//  CLASS: Exceptions2:
 Exceptions2( )
 {
    super();
    return;

 }

//  CLASS: Exceptions2:
 public  static    void main(  String [] aString1)
 {

    Exceptions2 JdecGenerated2 = new Exceptions2();
    return;

 }

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


}
```

Listing 59: Test Result: jdec - Optimised test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File 'COPYING' For more details.

/**** List of All Imported Classes ***/

import java.lang.Object;

// End of Import

public  class  Optimised {
//  CLASS: Optimised:
 public     Optimised( )
 {
    super();
    return;
 }
//  CLASS: Optimised:
 public  static    void f()
 {
    Circle Var_1= null;
    Object Var_1= null;
    Rectangle Var_1= null;

    if(arg0 > 10)
    {
      Rectangle JdecGenerated8 = new Rectangle(      Var_1=JdecGenerated8
           ;
arg0,arg0);
      arg0=this.isFat();
      Var_1=this;
    }
    else
    {
      Circle JdecGenerated28 = new Circle(      Var_1=JdecGenerated28;
arg0);
      arg0=this.isFat();
      Var_1=this;
    }
    if(arg0==0)
    {
      this.draw();
      return ;
    }
 }
//  CLASS: Optimised:
 public  static    void main()  {
    f(11);
    return;
 }
}
```

Listing 60: Test Result: jdec - Args test program

```
//  Decompiled by jdec
//  DECOMPILER HOME PAGE: jdec.sourceforge.net
//  Main HOSTING SITE: sourceforge.net
//  Copyright (C)2006,2007,2008 Swaroop Belur.
//  jdec comes with ABSOLUTELY NO WARRANTY;
//  This is free software, and you are welcome to redistribute
//  it under certain conditions;
//  See the File 'COPYING' For more details.

/**** List of All Imported Classes ***/

import java.io.PrintStream;
import java.lang.Object;
import java.lang.System;

// End of Import

class  Args

{


//  CLASS: Args:
 Args( )
 {
    super();
    return;

 }

//  CLASS: Args:
 public  static    void main(  java.lang.String [] aString1)
 {
    java.lang.String [] aString1= 0;
    aString1=0;
    System.out.println(aString1);
    return;

 }


}
```

Listing 63: Test Result: jdec - Extract 4 from connectfour test program

```
if(board[(_loop_param- 1)][((_loop_param + 1)- 1)] != who) {

}else{

}
```

Listing 64: Test Result: JODE - TryFinally test program

```
/* TryFinally - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */

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

Listing 65: Test Result: JODE - ControlFlow test program

```
/* ControlFlow - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */

public class ControlFlow
{
    public int foo(int i, int i_0_) {
  for (;;) {
      try {
    for (/**/; i < i_0_; i = i_0_++ / i) {
        /* empty */
    }
    break;
      } catch (RuntimeException runtimeexception) {
    i = 10;
      }
  }
  return i_0_;
    }

}
```

Listing 66: Test Result: JODE - Exceptions test program

```
Exception while decompiling:jode.AssertError: Exception handlers ranges
     are intersecting: [16, 32] and [8, 24].
  at jode.flow.TransformExceptionHandlers.checkTryCatchOrder(
       TransformExceptionHandlers.java:914)
  at jode.flow.TransformExceptionHandlers.analyze(
       TransformExceptionHandlers.java:928)
  at jode.decompiler.MethodAnalyzer.analyzeCode(MethodAnalyzer.java:577)
  at jode.decompiler.MethodAnalyzer.analyze(MethodAnalyzer.java:652)
  at jode.decompiler.ClassAnalyzer.analyze(ClassAnalyzer.java:352)
  at jode.decompiler.ClassAnalyzer.dumpJavaFile(ClassAnalyzer.java:624)
  at jode.decompiler.Decompiler.decompile(Decompiler.java:192)
  at jode.swingui.Main.run(Main.java:204)
  at java.lang.Thread.run(Thread.java:619)
```

Listing 67: Test Result: JODE - Optmised test program

```
/* Optimised - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */

public class Optimised
{
    public static void f(short arg0) {
  boolean bool;
  Drawable drawable;
  if (arg0 > 10) {
      Rectangle rectangle = new Rectangle;
      ((UNCONSTRUCTED)rectangle).Rectangle(arg0, arg0);
      bool = rectangle.isFat();
      drawable = rectangle;
  } else {
      Circle circle = new Circle;
      ((UNCONSTRUCTED)circle).Circle(arg0);
      bool = circle.isFat();
      drawable = circle;
  }
  if (!bool)
      drawable.draw();
    }

    public static void main(String[] arg0) {
  f((short) 11);
    }
}
```

Listing 68: Test Result: JODE - connectfour test program

```
Exception while decompiling:java.lang.IllegalArgumentException: stack
    length differs
  at jode.flow.VariableStack.merge(VariableStack.java:77)
  at jode.flow.LoopBlock.mergeContinueStack(LoopBlock.java:454)
  at jode.flow.LoopBlock.mapStackToLocal(LoopBlock.java:440)
  at jode.flow.SequentialBlock.mapStackToLocal(SequentialBlock.java:73)
  at jode.flow.SequentialBlock.mapStackToLocal(SequentialBlock.java:73)
  at jode.flow.FlowBlock.mapStackToLocal(FlowBlock.java:1531)
  at jode.flow.FlowBlock.mapStackToLocal(FlowBlock.java:1515)
  at jode.decompiler.MethodAnalyzer.analyzeCode(MethodAnalyzer.java:580)
  at jode.decompiler.MethodAnalyzer.analyze(MethodAnalyzer.java:652)
  at jode.decompiler.ClassAnalyzer.analyze(ClassAnalyzer.java:355)
  at jode.decompiler.ClassAnalyzer.dumpJavaFile(ClassAnalyzer.java:624)
  at jode.decompiler.Decompiler.decompile(Decompiler.java:192)
  at jode.swingui.Main.run(Main.java:204)
  at java.lang.Thread.run(Thread.java:619)
```

Listing 69: Test Result: jReversePro - Fibo test program

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index:
    2304, Size: 68
  at java.util.ArrayList.RangeCheck(ArrayList.java:547)
  at java.util.ArrayList.get(ArrayList.java:322)
  at jreversepro.reflect.JConstantPool.getCpValue(JConstantPool.java:381)
  at jreversepro.reflect.JConstantPool.getUtf8String(JConstantPool.java
      :457)
  at jreversepro.parser.JClassParser.readMethodAttributes(JClassParser.
      java:677)
  at jreversepro.parser.JClassParser.readMethods(JClassParser.java:659)
  at jreversepro.parser.JClassParser.parse(JClassParser.java:199)
  at jreversepro.parser.JClassParser.parse(JClassParser.java:166)
  at jreversepro.parser.JClassParser.parse(JClassParser.java:119)
  at jreversepro.revengine.JSerializer.loadClass(JSerializer.java:80)
  at jreversepro.JCmdMain.loadClass(JCmdMain.java:241)
  at jreversepro.JCmdMain.process(JCmdMain.java:186)
  at jreversepro.JCmdMain.main(JCmdMain.java:159)
```

Listing 70: Test Result: jReversePro - Casting test program

```
// JReversePro v 1.4.1 Sun Feb 01 14:23:58 GMT 2009
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions;See the File 'COPYING' for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSiON: 50.0
// SOURCEFILE: null

import java.io.PrintStream;


public class Casting{


    public Casting()
    {
         ;
         return;
    }


    public static void main(String[] stringArr)
    {
         int i = 0;
         for (;i < 128;) {
              System.out.println(new StringBuilder().append("ascii ").
                   append(i).append(" character ").append(i).toString());
              char j = (char)(i + 1);
         }
         return;
    }

}
```

Listing 76: Test Result: jReversePro - Extract 1 from connectfour test program

```
else if (k == 0) {
}
else if (0 & 1 != 0) {
}
else if (j != 1) {
}
else if (k != 1) {
}
else if (j != 2) {
}
else if (1 | 0 & 1 != 0) {
}
else if (1 & (o ^ 1) != 0) {
}
else if (k != 1) {
}
else if (1 & (o ^ 1) != 0) {
}
```

Listing 71: Test Result: jReversePro - Usa test program

```
// JReversePro v 1.4.1 Sun Feb 01 14:25:55 GMT 2009
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions;See the File 'COPYING' for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSiON: 50.0
// SOURCEFILE: Usa.java



public class Usa{

  public String name;

    public Usa()
    {
         ;
         name = "Detroit";
         return;
    }

}
```

Listing 77: Test Result: jReversePro - Extract 2 from connectfour test program

```
for (;n <= m;) {
      else if (iArr[n - 1] == 0) {
      }
      else if (0 & 1 != 0) {
      }
}
```

Listing 79: Test Result: Mocha - Extract 1 from connectfour test program

```
public connectfour() throws {
    $this.<init>();
}
```

Listing 72: Test Result: jReversePro - TryFinally test program

```
// JReversePro v 1.4.1 Sun Feb 01 14:21:55 GMT 2009
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions;See the File 'COPYING' for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSiON: 50.0
// SOURCEFILE: null

import java.io.PrintStream;


public class TryFinally{


    public TryFinally()
    {
         ;
         return;
    }


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

}
```

Listing 73: Test Result: jReversePro - Exceptions test program

```
jreversepro.revengine.RevEngineException: Block overlap   false 16 16  35
     TYPE_TRY    with    true 8 8  27 TYPE_TRY
  at jreversepro.runtime.JRunTimeContext.pushControlEntry(JRunTimeContext
      .java:302)
  at jreversepro.runtime.JRunTimeContext.getBeginStmt(JRunTimeContext.
      java:209)
  at jreversepro.revengine.JDecompiler.genSource(JDecompiler.java:476)
  at jreversepro.revengine.JDecompiler.genCode(JDecompiler.java:215)
  at jreversepro.reflect.JClassInfo.processMethods(JClassInfo.java:406)
  at jreversepro.reflect.JClassInfo.reverseEngineer(JClassInfo.java:592)
  at jreversepro.JCmdMain.process(JCmdMain.java:323)
  at jreversepro.JCmdMain.process(JCmdMain.java:198)
  at jreversepro.JCmdMain.main(JCmdMain.java:159)
```

Listing 74: Test Result: jReversePro - Optmised test program

```
// JReversePro v 1.4.1 Tue Feb 17 18:33:46 GMT 2009
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions;See the File 'COPYING' for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSiON: 46.0
// SOURCEFILE: Optimised.java



public class Optimised{


    public Optimised()
    {
         ;
         return;
    }


    public static void f(short i)
    {
         Drawable drawable;
         if (i > 10) {
              Rectangle rectangle = new Rectangle;
              rectangle(i , i);
              i = rectangle.isFat();
              rectangle = rectangle;
         }
         else {
              drawable = new Circle;
              drawable(i);
              i = drawable.isFat();
              drawable = drawable;
         }
         if (i == 0)
              drawable.draw();

         return;
    }


    public static void main(String[] stringArr)
    {
         Optimised.f(11);
         return;
    }

}
```

Listing 75: Test Result: jReversePro - Args test program

```
// JReversePro v 1.4.1 Sun Feb 01 14:22:50 GMT 2009
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions;See the File 'COPYING' for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSiON: 45.3
// SOURCEFILE: Args.jasmin

import java.io.PrintStream;


class Args{


    Args()
    {
         ;
         return;
    }


    public static void main(String[] stringArr)
    {
         stringArr = 0;
         System.out.println(stringArr);
         return;
    }

}
```

Listing 78: Test Result: Mocha - Args test program

```
/* Decompiled by Mocha from Args.class */
/* Originally compiled from Args.jasmin */

import java.io.PrintStream;

synchronized class Args
{
    Args()
    {
    }

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

Listing 80: Test Result: Mocha - Extract 2 from connectfour test program

```
public static void initialize_board(int board[][]) throws {
    int _loop_param;
    int _loop_limit;
    int _loop_param;
    int _loop_limit;
    expression 1
    _loop_limit = 6;
    pop _loop_param
    if (_loop_param > _loop_limit)
        expression 1
    _loop_limit = 7;
    pop _loop_param
    for (; _loop_param <= _loop_limit; _loop_param++)
        board[_loop_param - 1][_loop_param - 1] = 0;
    _loop_param++;
}
```

Listing 81: Test Result: Mocha - Extract 3 from connectfour test program

```
if (who != 1) goto 82 else 10;
```

Listing 82: Test Result: SourceAgain - Casting test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

import java.io.PrintStream;

public class Casting {

    public static void main(String[] as)
    {
        char c = 0;

        for( c = (char) 0; c < 128; c = (char) (c + 1) )
            System.out.println( new StringBuilder().append( "ascii " ).
                 append( c ).append( " character " ).append( c ).toString
                 () );
    }
}
```

Listing 83: Test Result: SourceAgain - Usa test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

public class Usa {

    public String name = "Detroit";
}
```

Listing 89: Test Result: SourceAgain - Extract 1 from connectfour test program

```
public connectfour() {
    $this();
}
```

Listing 84: Test Result: SourceAgain - Sable test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

public class Sable {
Warning #2008: Inheritance relationship can not be determined because
     Circle can not be found.
Warning #2008: Inheritance relationship can not be determined because
     Rectangle can not be found.
Warning #2008: Inheritance relationship can not be determined because
     Drawable can not be found.

    public static void f(short si)
    {
        Object obj = null;
        boolean bool = false;

        if( si > 10 )
        {
            Object obj1 = new Rectangle( si, si );

            bool = ((Rectangle) obj1).isFat();
            obj = obj1;
        }
        else
        {
            Object obj2 = new Circle( si );

            bool = ((Circle) obj2).isFat();
            obj = obj2;
        }
        if( !bool )
            ((Drawable) obj).draw();
    }

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

Listing 90: Test Result: SourceAgain - Extract 2 from connectfour test program

```
public static void initialize_board(int[][] board) {
    int _loop_limit = 6;
    int _loop_param = 0;

    for( _loop_param = 1; _loop_param <= _loop_limit; ++_loop_param ) {
        int _loop_limit = 7;
        int _loop_param = 0;

        for( _loop_param = 1; _loop_param <= _loop_limit; ++_loop_param )
           board[_loop_param - 1][_loop_param - 1] = 0;
    }
}
```

Listing 85: Test Result: SourceAgain - TryFinally test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

import java.io.PrintStream;

public class TryFinally {

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

Listing 86: Test Result: SourceAgain - ControlFlow test program

```
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

public class ControlFlow {

    public int foo(int i, int j)
    {
        for( ;; )
        {
            try
            {
                if( i < j )
                {
                    i = j++ / i;
                    continue;
                }
            }
            catch( RuntimeException runtimeexception1 )
            {
                i = 10;
                continue;
            }
            return j;
        }
    }
}
```

Listing 87: Test Result: SourceAgain - Exceptions test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

import java.io.PrintStream;

class Exceptions {

    public void Exceptions()
    {
        System.out.println( "a" );
label_15:
        {
            try
            {
                System.out.println( "b" );
                try
                {
                    System.out.println( "c" );
                    break label_15;
                }
                catch( RuntimeException runtimeexception1 )
                {
                    System.out.println( "e" );
                }
            }
            catch( RuntimeException runtimeexception2 )
            {
                System.out.println( "g" );
            }
            System.out.println( "f" );
            return;
        }
        System.out.println( "d" );
    }

    public static void main()
    {
        Exceptions exceptions1 = new Exceptions();

    }
}
```

Listing 88: Test Result: SourceAgain - Optmised test program

```
Warning #2002: Environment variable CLASSPATH not set.
//
// SourceAgain (TM) Professional v1.10k (C) 2003 Ahpah Software Inc
//

public class Optimised {
Warning #2008: Inheritance relationship can not be determined because
    Circle can not be found.
Warning #2008: Inheritance relationship can not be determined because
    Rectangle can not be found.
Warning #2008: Inheritance relationship can not be determined because
    Drawable can not be found.

    public static void f(short arg0)
    {
        Object obj = null;
        boolean bool = false;

        if( arg0 > 10 )
        {
            obj = new Rectangle;
            (Rectangle) obj( arg0, arg0 );
            bool = ((Rectangle) obj).isFat();
            obj = obj;
        }
        else
        {
            obj = new Circle;
            (Circle) obj( bool );
            bool = ((Circle) obj).isFat();
            obj = obj;
        }
        if( !bool )
            ((Drawable) obj).draw();
    }

    public static void main(String[] arg0)
    {
        f( (short) 11 );
    }
}
```