An Evaluation of Current Java Bytecode Decompilers

James Hamilton, Sebastian Danicic
Department of Computing
Goldsmiths, University of London
United Kingdom
james.hamilton@gold.ac.uk, s.danicic@gold.ac.uk

***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 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*-java; bytecode; decompilation;**

I. Introduction

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 [1]. Java source code is compiled into an intermediate language known as Java bytecode. 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 [2] and updated in JSR202 [3]. The open specification allows tools other than Sun's Java compiler to generate and/or manipulate Java bytecode.

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[1]) or
3) by writing a class file by hand[2].

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 I 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 [5]. 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 [6].

Decompilation has many applications including legitimate uses, such as the recovery of lost source code for a crucial application [7] 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 hence 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.

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.

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 paper on a survey performed in 2003 [8] 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

[1]JGNAT is an open-source Ada compiler which compiles Ada source to Java bytecode

[2]The tedious task of hand-writing a Java class file can be made easier by using a Java assembler, such as Jasmin [4], which accepts a human readable form of Java bytecode instructions and generates a Java class file.

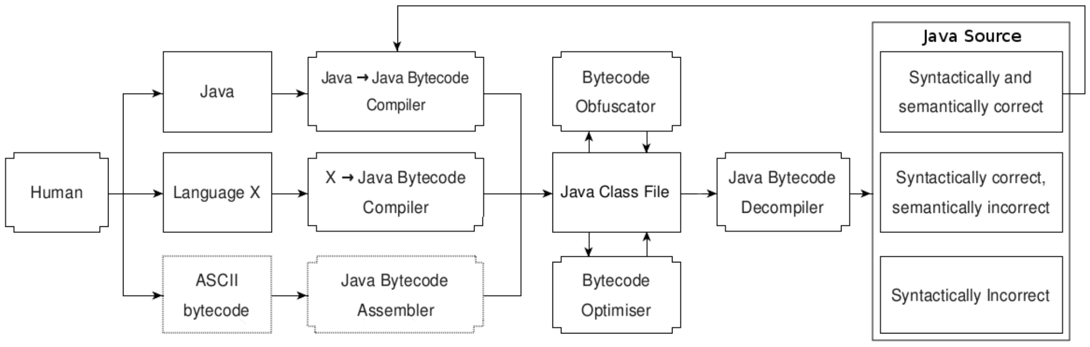


Figure 1. Compilation and decompilation of Java Bytecode

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 organised as follows: In Section II, the method of decompiler evaluation is described. In Section III, we present the results of our evaluation and in Section IV, we present our conclusions.

II. 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. In fact, many Java bytecode obfuscators rely on the fact that most decompilers fail when encountering unexpected, but valid, bytecode sequences [9].

Naeem *et al.* [10] suggest using software metrics [11], [12] 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. For each program-decompiler pair, we give a score between 0 and 9 (see Figure 2). A score of 0 is a perfect, or good, decompilation whereas 9 means that the decompiler failed and no output was produced.

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

Programs classified as 2, 3, 4 indicate varying levels of syntactically incorrect programs. A category 2 program indicates a program with small syntax errors, such as a missing variable declaration, that can be easily corrected to produce a semantically correct program. Category 3 programs contain syntactic errors which are harder to correct, such as programs with goto statements (which do not exist in Java), and category 4 programs contain extreme syntax errors which are very difficult or impossible to correct.

Programs classified as 5, 6, 7 are syntactically correct but are not semantically equivalent to the input program. These categories of programs are, 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 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.

A. Tests

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

The original survey showed that different decompilers are sometimes better in different areas but no decompiler passed all the tests [8]. Of the decompilers tested in the original survey, JODE performed the best by correctly decompiling 6 out of the 9 test programs, with Dava and Jad close behind.

In our tests we include two types of Java class file: those generated by *javac* and those generated by other tools. Java source files are compiled with *javac* version 1.6.0_10. Arbitrary Java class files include two hand-written using

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

Figure 2. Decompilation correctness classification

the Jasmin assembler version 2.1, one optimised using the Soot Framework and one compiled by JGNAT. We use the test programs from the original survey, where possible, and also extend the evaluation to include more problem areas such as the correct decompilation of try-finally blocks and local variable slot re-use. The Java class files were decompiled using each decompiler and the result was classified into one of our ten categories of decompiler effectiveness. The class files considered were:

Fibonacci is a trivial test for a decompiler. It is a fairly simple program to output the Fibonacci number of a given input number.
Casting [13] is a simple program to test if a decompiler can correctly detect the need to cast a *char* to an *int*.
InnerClass [13] is a simple program containing inner classes.
TypeInference [14] is a program which tests a decompiler's ability to perform type inference for local variables. A specific variable in the program is difficult for a decompiler to type because it depends on the value of another variable. The original paper [14], written by the developers of Dava, tested three different decompilers against Dava. They showed that their decompiler could correctly type the variables whereas the other decompilers failed to do so.
TryFinally is a simple test to determine whether decompilers can decompile the implementation of try-finally blocks using inline code instead of Java bytecode subroutines. Many of the old decompilers expect the use of subroutines for try-finally blocks but *javac* 1.4.2+ generates inline code instead.
ControlFlow [14] is a program which tests a decompiler's handling of control flow.
Exceptions [14] contains two intersecting try-catch blocks. The intersecting try-catch block is allowed in Java bytecode but would not be generated by a Java compiler - the program here is created using Jasmin. The program used in the original tests is incorrect and Dava, which should be able to decompile the program, exits with a null pointer exception. A re-written version is used in our tests based on the call graph in the original paper [14].
Optimised was generated by using the Soot optimiser

on the *TypeInference* test program. An example of an optimisation that has been performed is the removal of the *dup* opcode (which duplicates the value on the top of the stack) and replacing it with *load* and *store* instructions.

VariableReUse re-uses the local variable slot 0 in the main method. At the start of the method local variable slot 0 is of type *String[]* but it is then re-used as type *int*. A compiler would not generate such code that we created using Jasmin. Multiple types for local variables is valid bytecode as long as the lifetimes of the two uses of the local variable does not overlap [15] i.e. a local variable only has the type (and value) of the last variable stored in it.

Ada [16] is an implementation of the game Connect Four originally written in Ada and compiled with JGNAT to Java bytecode. This program provides an example of a source language other than Java for a Java decompiler to handle. Such programs potentially contain code which cannot easily be decompiled to Java source due to unexpected bytecode sequences generated by a non-Java to Java bytecode compiler. The original survey used a different Ada program compiled to Java bytecode which we could not obtain.

III. The Evaluation

The following decompilers were tested:

Mocha [17], 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.

SourceTec (Jasmine) [18], also known as Jasmine, is another unmaintained old compiler which is a patch to Mocha. The installation process involves providing Mocha's class files which are then patched by SourceTec (Jasmine).

SourceAgain [19] was a commercial decompiler from Ahpah Software, Inc. It is no longer sold or supported though they keep a web-based version of their decompiler available on their website.

ClassCracker3 [20] is another commercial decompiler which seems not to have been updated for at least four years. An evaluation version of the program is available at the Mayon Software Research's website which states that it will decompile the first 5 methods of a Java class file.

Jad [21] is a popular decompiler that is free for non-commercial use but is no longer maintained. It is a closed source program written in C. The last update for Linux and Windows version for Jad was in 2001, while a small update added an OS X version in 2006. Jad is used as the back-end by many decompiler GUIs [21] including an Eclipse IDE plug-in named JadClipse [22].

JODE [23] is an open-source decompiler that also includes a bytecode optimiser. The latest version 1.1.2-pre1 was released February 24, 2004.

jReversePro [24] is an open-source disassembler and decompiler project which is currently at version 1.4.2 though hasn't been updated for several years.

Dava [14], [25]–[28] is a decompiler which is part of the Soot Java Optimisation Framework [29] from the Sable Research Group[3] at McGill University in Montreal, Quebec, Canada. Soot is under constant development at the Sable Research Group and the latest release was version 2.3.0 on June 3, 2008.

jdec [30] is an open-source decompiler written in Java which was last released at version 2.0 in May, 2008. jdec is aimed at the decompilation of bytecode generated by Sun's *javac* compiler and therefore will probably have problems decompiling the arbitrary code.

Java Decompiler [31] is a free Java decompiler aimed at decompiling Java 5 and above class files. It is in its early stages, at only version 0.2.7, and has been in development for about a year.

NMI Code Viewer was included in the original survey with results 'startlingly similar' to Jad [8]. We do not include this (unmaintained) decompiler as, in actual fact, it is a front-end for Jad [21].

jAscii was included in the original survey, with poor results [8], but it is now obsolete and unavailable for our evaluation.

A. Results

ClassCracker3 did not decompile any of our test programs completely with just the method signatures in the resulting Java source file.

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 and reverse it. 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.

Jad produced some pleasing results, with a similar score to JODE. 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

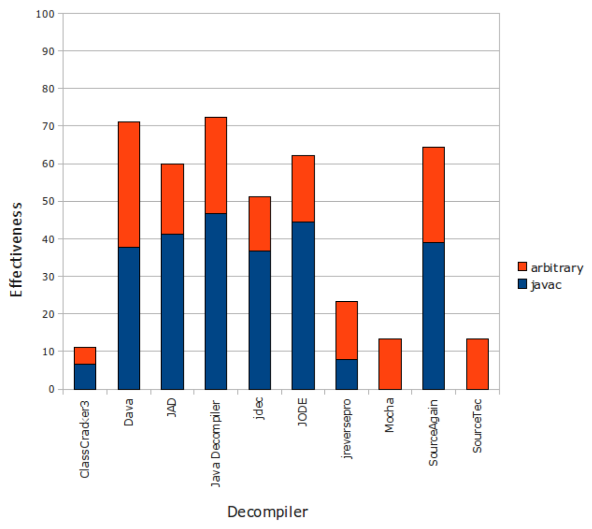
[3]http://www.sable.mcgill.ca/

decompiler	type	status	2003 version	current version	last update
Mocha	free	obsolete	0.1b	0.1b	1996
SourceTec	commercial	obsolete	1.1	1.1	1997
SourceAgain	commercial	obsolete	1.10j	1.1	2004
Jad	free	unmaintained	1.5.8e	1.5.8e	2001
JODE	open-source	unmaintained	unknown	1.1.2-pre1	2004
ClassCracker3	commercial	obsolete	3.01	3.02	2005
jReversePro	open-source	unmaintained	1.4.1	1.4.2	2005
Dava	open-source	current	2.0.1	2.3.0	2008
jdec	open-source	current	N/A	2.0	2008
Java Decompiler	free	current	N/A	0.2.7	2008

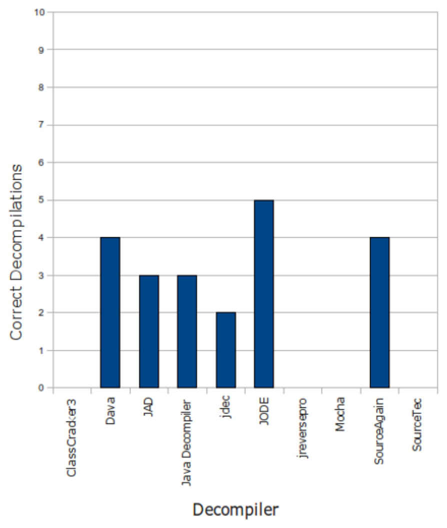
Figure 3. Decompilers: Some of the currently available decompilers have, since the original survey [8] in 2003, been upgraded, become unmaintained or obsolete. There are two new decompilers, jdec and Java Decompiler which have become available since 2003. Many commercial Java bytecode decompilers have become obsolete and the currently maintained decompilers are open-source and/or free. In the original survey JODE was determined to be the best by decompiling 6 out of 9 programs correctly, closely followed by Jad and Dava.

		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	0	8	9	8	9	8	9
	TypeInference	8	2	1	2	4	0	9	9	1	9
	TryFinally	8	4	8	0	0	4	8	9	4	9
	ControlFlow	8	1	1	5	4	1	8	9	1	9
arbitrary	Exceptions	8	0	4	4	8	9	9	9	7	9
	Optimised	8	2	4	3	4	2	3	9	3	9
	VariabeReUse	8	1	2	2	3	0	2	2	0	2
	Ada	8	3	9	4	8	9	8	4	3	4

Figure 4. 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.



(a) Decompiler Effectiveness



(b) Correct Decompilations

Figure 5. Decompiler Effectiveness

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.

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 InnerClass test programs, which are both *javac* generated. Java Decompiler has some trouble decompiling arbitrary bytecode but does this better than most of the other decompilers, except Dava. Java Decompiler outperforms Jad and JODE by performing slightly better with arbitrary bytecode. Java Decompiler decompiles 3 programs correctly, two less than JODE. Java Decompiler produces the same mistake in the TypeInference test program as Dava - a missing typecast. Java Decompiler and Dava fall behind in the number of correct decompilations because of this (easily corrected) syntactic mistake.

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.

JODE has a similar overall result to Dava 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 decompile 3 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.

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.

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*.

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. The product is no longer sold or supported and is only available online to decompiler single class files. SourceAgain is therefore obsolete.

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

No decompiler was able to decompile all test programs with JODE decompiling the most programs correctly. JODE only managed to decompile 5 programs correctly while four (unmaintained) decompilers could not decompile any of the test programs correctly. The top three *javac* oriented decompilers were JODE, Jad and Java Decompiler 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, unsurprisingly, was better at decompiling the arbitrary bytecode test programs than the *javac* test programs. Dava is the second best decompiler based on our effectiveness measures, but is the best arbitrary bytecode decompiler. 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, and one less than JODE.

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 two more programs correctly than Java Decompiler and Jad though. 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. JODE performs similarly to Jad and Java Decompiler with *javac* generated bytecode in our tests.

IV. 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 [7], 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 [14], [25]–[28] decompiler, part of the Soot Optimisation Framework [29], which is a research project carried out by the Sable Research Group at McGill University. Dava differs from other decompilers in that it aims to decompile arbitrary bytecode whereas other decompilers rely on known patterns produced by Java compilers (and this is usually *javac*). Dava is better at decompiling arbitrary bytecode whereas other decompilers are better at decompiling *javac* generated bytecode.

A decompiler aimed at decompiling arbitrary bytecode, like Dava, can be more useful in some instances than a decompiler aimed at bytecode generated by a specific compiler. Java bytecode can be generated by tools other than a Java decompiler and many decompilers are aimed at patterns produced by Java decompilers and some specifically *javac*. Knowing the patterns that a compiler will produce makes decompilation of bytecode easier and it can sometimes be just a 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. [15], [28], [32]) and machine code (e.g. [33]). 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 [34], 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 [32].

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 two less programs 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 closed-source, 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 - the same number as Dava. SourceAgain performed similarly to Jad 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 3 out of 10 programs. This decompiler performs best at *javac* generated bytecode and may improve in the future even more as it is in development.

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

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

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

References

[1] W. Caudle, "On inverse of compiling," *Sperry-UNIVAC*, Apr. 1980. [Online]. Available: http://www.program-transformation.org/view/Transform/OnInverseOfCompiling?skin=print.pattern

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

[3] A. Buckley, E. Rose, A. Coglio, B. S. Corporation, I. Sun Microsystems, I. Tmax Soft, S. Technologies, and E. AG, "Jsr 202: Javatm class file specification update," 2006. [Online]. Available: http://jcp.org/en/jsr/detail?id=202

[4] J. Meyer, D. Reynaud, I. Kharon *et al.*, "Jasmin," sourceforge.net, 2004. [Online]. Available: http://jasmin.sourceforge.net/

[5] M. V. Emmerik, "Static single assignment for decompilation," PhD Thesis, The University of Queensland, 2007.

[6] C. Cifuentes, "Reverse compilation techniques," PhD Thesis, Queensland University of Technology, 1994. [Online]. Available: http://citeseer.ist.psu.edu/cifuentes94reverse.html

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

[8] M. V. Emmerik, “Java decompiler tests,” 2003. [Online]. Available: http://www.program-transformation.org/Transform/JavaDecompilerTests

[9] M. Batchelder and L. J. Hendren, “Obfuscating java: the most pain for the least gain,” in *CC ’07: Proceedings of Compiler Construction, 16th International Conference*, 2007, pp. 96–110.

[10] N. A. Naeem, M. Batchelder, and L. Hendren, “Metrics for measuring the effectiveness of decompilers and obfuscators,” in *ICPC ’07: Proceedings of the 15th IEEE International Conference on Program Comprehension*. Washington, DC, USA: IEEE Computer Society, 2007, p. 253258.

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

[12] Kearney, Sedlmeyer, Thompson, Gray, and Adler, “Software complexity measurement,” *Commun. ACM*, vol. 29, no. 11, p. 10441050, 1986.

[13] G. Nolan, *Decompiling Java*. APress, 2004.

[14] J. Miecznikowski and L. J. Hendren, “Decompiling java bytecode: Problems, traps and pitfalls,” in *CC ’02: Proceedings of the 11th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2002, p. 111?127.

[15] T. B. Knoblock and J. Rehof, “Type elaboration and subtype completion for java bytecode,” *ACM Trans. Program. Lang. Syst.*, vol. 23, no. 2, pp. 243–272, 2001.

[16] B. Fagin and M. Carlisle, “Connect Four(TM) game, written in ada,” Department of Computer Science, 2005. [Online]. Available: http://webdiis.unizar.es/asignaturas/EDA/gnat/jgnat/connect_four/test.html

[17] H. van Vliet, “Mocha, the java decompiler,” 1996. [Online]. Available: http://www.brouhaha.com/~eric/computers/mocha.html

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

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

[20] Mayon Software Research, “ClassCracker 3,” http://mayon.actewagl.net.au/, 2005.

[21] P. Kouznetsov, “Jad - the fast java decompiler,” http://www.kpdus.com/jad.html, Mar. 2006. [Online]. Available: http://www.kpdus.com/jad.html

[22] V. Grishchenko and J. Gyger, “JadClipse,” http://jadclipse.sourceforge.net/wiki/index.php/Main_Page, 2009.

[23] J. Hoenicke, “JODE,” http://jode.sourceforge.net/, 2004.

[24] K. Kumar, “JReversePro - java decompiler / disassembler,” http://jreversepro.blogspot.com, 2005, JReversePro is a java decompiler / disassembler written in Java.

[25] J. Miecznikowski, “New algorithms for a java decompiler and their implementation in soot,” Masters Thesis, McGill University, 2003.

[26] N. A. Naeem and L. Hendren, “Programmer-Friendly decompiled java,” in *ICPC ’06: Proceedings of the 14th IEEE International Conference onProgram Comprehension (ICPC’06)*. IEEE Computer Society, 2006, p. 327—336.

[27] N. A. Naeem, “Programmer-Friendly decompiled java,” Masters Thesis, 2007. [Online]. Available: http://www.sable.mcgill.ca/publications/thesis/#nomairMastersThesis

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

[29] R. Valle-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “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, 1999, p. 13.

[30] S. Belur and K. Bettadapura, “Jdec: Java decompiler,” http://jdec.sourceforge.net/, 2008.

[31] E. Dupuy, “Java decompiler,” http://java.decompiler.free.fr/, 2008.

[32] B. Bellamy, P. Avgustinov, O. de Moor, and D. Sereni, “Efficient local type inference,” *SIGPLAN Not.*, vol. 43, no. 10, pp. 475–492, 2008.

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

[34] E. Gagnon, L. J. Hendren, and G. Marceau, “Efficient inference of static types for java bytecode,” in *Static Analysis Symposium*, 2000, pp. 199–219. [Online]. Available: www.sable.mcgill.ca/publications