The Journal of Systems & Software 168 (2020) 110645

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Java decompiler diversity and its application to meta-decompilation

Nicolas Harrand *, César Soto-Valero, Martin Monperrus, Benoit Baudry

KTH Royal Institute of Technology, SE-100 44 Stockholm, Sweden

ARTICLE INFO

Article history:
Received 2 December 2019
Received in revised form 17 April 2020
Accepted 12 May 2020
Available online 20 May 2020

Keywords:
Java bytecode
Decompilation
Reverse engineering
Source code analysis

ABSTRACT

During compilation from Java source code to bytecode, some information is irreversibly lost. In other words, compilation and decompilation of Java code is not symmetric. Consequently, decompilation, which aims at producing source code from bytecode, relies on strategies to reconstruct the information that has been lost. Different Java decompilers use distinct strategies to achieve proper decompilation. In this work, we hypothesize that the diverse ways in which bytecode can be decompiled has a direct impact on the quality of the source code produced by decompilers.

In this paper, we assess the strategies of eight Java decompilers with respect to three quality indicators: syntactic correctness, syntactic distortion and semantic equivalence modulo inputs. Our results show that no single modern decompiler is able to correctly handle the variety of bytecode structures coming from real-world programs. The highest ranking decompiler in this study produces syntactically correct, and semantically equivalent code output for 84%, respectively 78%, of the classes in our dataset. Our results demonstrate that each decompiler correctly handles a different set of bytecode classes.

We propose a new decompiler called Arlecchino that leverages the diversity of existing decompilers. To do so, we merge partial decompilation into a new one based on compilation errors. Arlecchino handles 37.6% of bytecode classes that were previously handled by no decompiler. We publish the sources of this new bytecode decompiler.

1. Introduction

In the Java programming language, source code is compiled into an intermediate stack-based representation known as bytecode, which is interpreted by the Java Virtual Machine (JVM). In the process of translating source code to bytecode, the compiler performs various analyses. Even if most optimizations are typically performed at runtime by the just-in-time (JIT) compiler, several pieces of information residing in the original source code are already not present in the bytecode anymore due to compiler optimization (Miecznikowski and Hendren, 2002; Lindholm et al., 2014). For example the structure of loops is altered and local variable names may be modified (Jaffe et al., 2018).

Decompilation is the inverse process, it consists in transforming the bytecode instructions into source code (Nolan, 2004). Decompilation can be done with several goals in mind. First, it can be used to help developers understand the code of the libraries they use. This is why Java IDEs such as IntelliJ and Eclipse include built-in decompilers to help developers analyze the third-party classes for which the source code is not available. In this case, the readability of the decompiled code is paramount. Second, decompilation may be a preliminary step before another compilation pass, for example with a different compiler. In this case, the main goal is that the decompiled code is syntactically correct and can be recompiled. Some other applications of decompilation with slightly different criteria include clone detection (Ragkhitwetsagul and Krinke, 2017), malware analysis (Yakdan et al., 2016; Ďurfina et al., 2013) and software archeology (Robles et al., 2005).

Overall, the ideal decompiler is one that transforms all inputs into source code that faithfully reflects the original code: the decompiled code (1) can be recompiled with a Java compiler and (2) behaves the same as the original program. However, previous studies that compared Java decompilers (Hamilton and Danicic, 2009; Kostelanský and Dedera, 2017) found that this ideal Java decompiler does not exist, because of the irreversible data loss that happens during compilation. In this paper, we perform a comprehensive assessment of three aspects of decompilation: the syntactic correctness of the decompiled code (the decompiled code can recompile); the semantic equivalence with the original source (the decompiled code passes all tests); the syntactic similarity to the original source (the decompiled source looks like the original). We evaluate eight recent and notable decompilers on 2041 Java classes, making this study one order of magnitude larger than the related work (Hamilton and Danicic, 2009; Kostelanský and Dedera, 2017).

* Corresponding author.
E-mail addresses: harrand@kth.se (N. Harrand), cesarsv@kth.se (C. Soto-Valero), martin.monperrus@csc.kth.se (M. Monperrus), baudry@kth.se (B. Baudry).

https://doi.org/10.1016/j.jss.2020.110645
0164-1212/

Next, we isolate a subset of 157 Java classes that no state-of-the-art decompiler can correctly handle. The presence of generics and wildcards is a major challenge that prevents successful decompilation. Meanwhile, we note that different decompilers fail for diverse reasons. This raises the opportunity to merge the results of several incorrect decompiled sources to produce a version that can be recompiled. We call this process **meta-decompilation**. Meta-decompilation is a novel approach for decompilation: (1) it leverages the natural diversity of existing decompilers by merging the results of different decompilers (2) it is able to provide decompiled sources for classes that no decompiler in isolation can handle. We implement this approach in a novel meta-decompiler called Arlecchino.

Our results have important implications: (1) for all users of decompilation, our paper shows significant differences between decompilers and provide well-founded empirical evidence to choose the best ones; (2) for researchers in decompilation, our results show that the problem is not solved; (3) for authors of decompilers, our experiments have identified bugs in their decompilers (3 have already been fixed, and counting) and our methodology of semantic equivalence modulo inputs can be embedded in the QA process of all decompilers in the world.

In summary, this paper makes the following contributions:

- an empirical comparison of eight Java decompilers based on 2041 real-world Java classes, tested by 25 019 test cases, identifying the key strengths and limitations of Java bytecode decompilation;
- meta-decompilation, a novel approach to decompilation that leverages decompilers diversity to improve decompilation effectiveness;
- a tool and a dataset for future research on Java decompilers publicly available at https://github.com/castor-software/decompilercmp

2. Background

In this section, we present an example drawn from the Apache `commons-codec` library. We wish to illustrate information loss during compilation of Java source code, as well as the different strategies that bytecode decompilers adopt to cope with this loss when they generate source code from bytecode. Listing 1 shows the original source code of the utility class `org.apache.commons.codec.net.Utils`, while Listing 2 shows an excerpt of the bytecode produced by the standard *javac* compiler.[1] Here, we omit the constant pool as well as the table of local variables and replace references towards these tables with comments to save space and make the bytecode more human readable.

As mentioned, the key challenge of decompilation resides in the many ways in which information is lost during compilation. Consequently, Java decompilers need to make several assumptions when interpreting bytecode instructions, which can also be generated in different ways. To illustrate this phenomenon, Listing 3 and Listing 4 show the Java sources produced by the Fernflower and Dava decompilers when interpreting the bytecode of Listing 2. In both cases, the decompilation produces correct Java code (i.e., recompilable) with the same functionality as the input bytecode. Notice that Fernflower guesses that the series of `StringBuilder` (bytecode instruction 23 to 27) calls is the compiler's way of translating string concatenation and is able to revert it. On the contrary, the Dava decompiler does not reverse this transformation.

As we notice, the decompiled sources are different from the original in at least three points: (1) In the original sources, the local variable *i* was `final`, but *javac* lost this information during compilation. (2) The `if` statement had originally no `else` clause. Indeed, when an exception is thrown in a method that does not catch it, the execution of the method is interrupted. Therefore, leaving the `return` statement outside of the `if` is equivalent to putting it inside an `else` clause. (3) In the original code the String `"Invalid URL encoding: not a valid digit (radix 16): "` was actually computed with `"Invalid URL encoding: not a valid digit (radix " + URLCodec.RADIX + "): "`. In this case, `URLCodec.RADIX` is actually a final static field that always contains the value 16 and cannot be changed. Thus, it is safe for the compiler to perform this optimization, but the information is lost in the bytecode.

Besides, this does not include the different formatting choices made by the decompilers such as new lines placement and brackets usage for single instructions such as `if` and `else`.

```
1  class Utils {
2    private static final int RADIX = 16;
3     static int digit16(final byte b) throws
           DecoderException {
4        final int i = Character.digit((char) b, RADIX);
5        if (i == -1) {
6           throw new DecoderException("Invalid URL
                 encoding: not a valid digit (radix " +
                 RADIX + "): " + b);
7        }
8        return i;
9     }
10 }
```

Listing 1: Source code of Java class corresponding to `org.apache.commons.codec.net.Utils`.

```
1  class org.apache.commons.codec.net.Utils {
2   static int digit16(byte) throws
         org.apache.commons.codec.DecoderException;
3         0: ILOAD_0 //Parameter byte b
4         1: I2C
5         2: BIPUSH 16
6         4: INVOKESTATIC #19 //Character.digit:(CI)I
7         7: ISTORE_1 //Variable int i
8         8: ILOAD_1
9         9: ICONST_m1
10        10: IF_ICMPNE 37
11 //org/apache/commons/codec/DecoderException
12        13: NEW #17
13        16: DUP
14        17: NEW #25 //java/lang/StringBuilder
15        20: DUP
16 //"Invalid URL encoding: not a valid digit (radix 16):"
17        21: LDC #27
18 //StringBuilder."<init>":(Ljava/lang/String;)V
19        23: INVOKESPECIAL #29
20        26: ILOAD_0
21 //StringBuilder.append:(I)Ljava/lang/StringBuilder;
22        27: INVOKEVIRTUAL #32
23 //StringBuilder.toString:()Ljava/lang/String;
24        30: INVOKEVIRTUAL #36
25 //DecoderException."<init>":(Ljava/lang/String;)V
26        33: INVOKESPECIAL #40
27        36: ATHROW
28        37: ILOAD_1
29        38: IRETURN
30 }
```

Listing 2: Excerpt of disassembled bytecode from code in Listing 1.

3. Decompiler evaluation methodology

In this section, we introduce definitions, metrics and research questions. Next, we detail the framework to compare decompilers and we describe the Java projects that form the set of case studies for this work.

[1] There are various Java compilers available, notably Oracle *javac* and Eclipse *ecj*, which can produce different bytecode for the same Java input.

```
class Utils {
   private static final int RADIX = 16;
   static int digit16(byte b) throws DecoderException {
      int i = Character.digit((char)b, 16);
      if(i == -1) {
         throw new DecoderException("Invalid URL
              encoding: not a valid digit (radix 16): " +
              b);
      } else {
         return i;
      }
   }
}
```

Listing 3: Decompilation result of Listing 2 with Fernflower.

```
class Utils
{
    static int digit16(byte b)
        throws DecoderException
    {
        int i = Character.digit((char)b, 16);
        if(i == -1)
            throw new DecoderException((new
                 StringBuilder()).append("Invalid URL
                 encoding: not a valid digit (radix 16):
                 ").append(b).toString());
        else
            return i;
    }
    private static final int RADIX = 16;
}
```

Listing 4: Decompilation result of Listing 2 with Dava.

3.1. Definitions and metrics

The value of the results produced by decompilation varies greatly depending on the intended use of the generated source code. In this work, we evaluate the decompilers' capacity to produce a faithful retranscription of the original sources. Therefore, we collect the following metrics.

Definition 1 (*Syntactic Correctness*)**.** The output of a decompiler is syntactically correct if it contains a valid Java program, i.e. a Java program that is recompilable with a Java compiler without any error.

When a bytecode decompiler generates source code that can be recompiled, this source code can still be syntactically different from the original. We introduce a metric to measure the scale of such a difference according to the abstract syntax tree (AST) dissimilarity (Falleri et al., 2014) between the original and the decompiled results. This metric, called *syntactic distortion*, allows to measure the differences that go beyond variable names. The description of the metric is as follows:

Definition 2 (*Syntactic Distortion*)**.** The minimum number of atomic edits required to transform the AST of the original source code of a program into the AST of the corresponding decompiled version of it.

In the general case, determining if two program are semantically equivalent is undecidable. For some cases, the decompiled sources can be recompiled into bytecode that is equivalent to the original, modulo reordering of the constant pool. We call these cases *strictly equivalent* programs. We measure this equivalence with a bytecode comparison tool named Jardiff.[2]

Inspired by the work of Le et al. (2014) and Yang et al. (2019), we check if the decompiled and recompiled program are semantically equivalent modulo inputs. This means that for a given set of inputs, the two programs produce equivalent outputs. In our case, we select the set of relevant inputs and assess equivalence based on the existing test suite of the original program.

Definition 3 (*Semantic Equivalence Modulo Inputs*)**.** We call a decompiled program semantically equivalent modulo inputs to the original if it passes the set of tests from the original test suite.

In the case where the decompiled and recompiled program produce non-equivalent outputs, that demonstrates that the sources generated by the decompiler express a different behavior than the original. As explained by Hamilton and colleagues (Hamilton and Danicic, 2009), this is particularly problematic as it can mislead decompiler users in their attempt to understand the original behavior of the program. We refer to these cases as *deceptive decompilation* results.

Definition 4 (*Deceptive Decompilation*)**.** Decompiler output that is syntactically correct but not semantically equivalent to the original input.

3.2. Research questions

We elaborated five research questions to guide our study on the characteristics of modern Java decompilers.

RQ1: **To what extent is decompiled Java code syntactically correct?** In this research question, we investigate the effectiveness of decompilers for producing syntactically correct and hence recompilable source code from bytecode produced by the *javac* and *ecj* compilers.

RQ2: **To what extent is decompiled Java code semantically equivalent modulo inputs?** Le and colleagues (Le et al., 2014) propose to use equivalence modulo inputs assessment as a way to test transformations that are meant to be semantic preserving (in particular compilation). In this research question, we adapt this concept in the context of decompilation testing. In this paper we rely on the existing test suite instead of generating inputs.

RQ3: **To what extent do decompilers produce deceptive decompilation results?** In this research question, we investigate the cases where we observe semantic differences between the original source code and the outputs of the decompilers.

RQ4: **What is the syntactic distortion of decompiled code?** Even if decompiled bytecode is ensured to be syntactically and semantically correct, syntactic differences may remain as an issue when the purpose of decompilation is human understanding. Keeping the decompiled source code free of syntactic distortions is essential during program comprehension, as many decompilers can produce human unreadable code structures. In this research question, we compare the syntactic distortions produced by decompilers.

RQ5: **To what extent do the successes and failures of decompilers overlap?** In this research question we investigate the intersection of classes for which each decompiler produce semantically equivalent modulo input sources.

3.3. Study protocol

Fig. 1 represents the pipeline of operations conducted on every Java source file in our dataset. For each triplet <*decompiler, compiler, project*>, we perform the following:

1. Compile the source files with a given compiler.
2. Decompile each class file with a decompiler (there might be several classes if the source defines internal classes). If the decompiler does not return any error, we mark the source file as decompilable. Then, (a) we measure syntactic distortion by comparing the AST of the original source with the AST of the decompiled source.

[2] https://github.com/scala/jardiff.

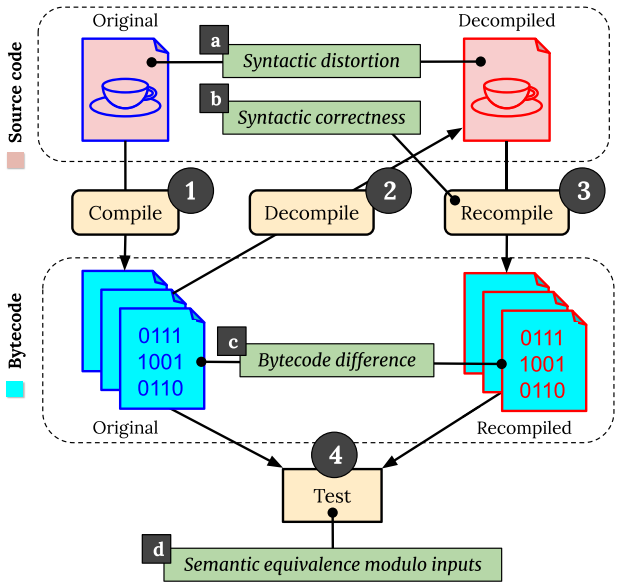

Fig. 1. Java decompiler assessment pipeline with four evaluation layers: syntactic distortion, bytecode difference, syntactic correctness, and semantic equivalence modulo input.

3. Recompile the class files with the given compiler. If the compilation is successful, we know that the decompiler produces (b) syntactically correct code. Then, we measure (c) the textual difference between the original and the recompiled bytecode. If there are none, the decompiler produced semantically equivalent code, otherwise we cannot assess anything in that regard yet.
4. Run the test cases on the recompiled bytecode. If the tests are successful, we mark the source as *passTests* for the given triplet, showing that the decompiler produces (d) semantically equivalent code modulo inputs.

If one of these steps fails we do not perform the following steps and consider all the resulting metrics not available. As decompilation can generate a program that does not stop, we set a 20 min timeout for the execution of the test suite (the original test suites run under a minute on the hardware used for this experiment, a Core i5-6600K with 16 GB of RAM).

The tests used to assess the semantic equivalence modulo inputs are those of the original project that cover the given Java file.[3] We manually excluded the tests that fail on the original project (either flaky or because versioning issue). The list of excluded tests is available as part of our experiments.

3.4. Study subjects

Decompilers. Table 1 shows the set of decompilers under study. We have selected Java decompilers that are (i) freely available, and (ii) have been active in the last two years. We add Jode in order to compare our results with a legacy decompiler, and because the previous survey by Hamilton and colleagues considers it to be one of the best decompilers (Hamilton and Danicic, 2009).

The column Version shows the version used (some decompilers do not follow any versioning scheme). We choose the latest release if one exists, if not the last commit available the 09-05-2019. The column Status indicates the date of the last commit or "Active" if the last commit was more recent than 30 days. The column #Commits represents the number of commits in the decompiler project, in cases where the decompiler is a submodule of a bigger project (e.g. Dava and Fernflower) we count only commits affecting the submodule. The column #LOC is the number of lines of code in all Java files (and Python files for Krakatau) of the decompiler, including sources, test sources and resources counted with *cloc*.[4]

Note that different decompilers are developed for different usages and, therefore, are meant to achieve different goals. CFR (Benfield, 2019) for Java 1[5] to 14, for code compiled with *javac* (note that since we performed our first experiments, it is now tested with *ecj* generated classes). Procyon (Strobel, 2019) from Java 5 and beyond and *javac*, shares its test suite with CFR. Fernflower (JetBrains, 2019) is the decompiler embedded in IntelliJ IDE. Krakatau (Storyyeller, 2019) up to Java 7 does not currently support Java 8 or `invokedynamic`. JD-Core (Dupuy, 2019) is the engine of JD-GUI. It supports Java 1.1.8 to Java 12.0. The version we study in this work is the first version released since the complete rewrite of JD-Core. While older versions were based on a simple bytecode pattern recognition engine, JD-Core now includes a CFG analysis layer. JADX (skylot, 2019) is a decompiler that originally targeted dex files (bytecode targeting the android platform) but can also target class files, as in our experiments. Dava (Jerome Miecznikowski and Naeem, 2019) is a decompiler built on top of the Soot Framework (Vallée-Rai et al., 1999). It does not target Java bytecode produced by any specific compiler nor from any specific language, but produces decompiled sources in Java. Soot supports bytecode and source code up to Java 7. Jode (Hoenicke, 2019) is a legacy decompiler that handles Java bytecode up to Java 1.4.

Projects. In order to get a set of real world Java projects to evaluate the eight decompilers, we reuse the set of projects of Pawlak and colleagues (Pawlak et al., 2015). To these 13 projects we added a fourteenth one named `DcTest` made out of examples collected from previous decompiler evaluations (Hamilton and Danicic, 2009; Kostelanský and Dedera, 2017).[6] Table 2 shows a summary of this dataset: the Java version in which they are written, the number of Java source files, the number of unit tests as reported by Apache Maven, and the number of Java lines of code in their sources.

As different Java compilers may translate the same sources into different bytecode representations (Dann et al., 2019),[7] we experiment with the two most used Java compilers: *javac* and *ecj* (versions 1.8.0_17 and 13.13.100, respectively). We compiled all 14 projects with both compilers (except `commons-lang` which failed to build with *ecj*). Our dataset includes 3928 bytecode classes, 1887 of which compiled with *ecj*, and 2041 compiled with *javac*. As we study the influence of the compiler, in RQ1, we limit our datasets to the 1887 classes that compiled with both compilers. As semantic equivalence modulo inputs is based on test suites, for RQ2 and RQ3 we focus on the classes that contain code executed by test suites: 2397 classes generated by the two compilers. These classes covered by the test suites exclude interfaces as they do not contain executable code. Most enum declarations fall under the same category. Test coverage is assessed through bytecode instrumentation with a tool named yajta.[8]

[3] Coverage was assessed using yajta https://github.com/castor-software/yajta.

[4] http://cloc.sourceforge.net/.

[5] https://github.com/leibnitz27/cfr/blob/33216277ae3b61a9d2b3f912d9ed91a3e698d536/src/org/benf/cfr/reader/entities/attributes/AttributeCode.java#L49.

[6] http://www.program-transformation.org/Transform/JavaDecompilerTests.

[7] https://www.benf.org/other/cfr/eclipse-differences.html.

[8] https://github.com/castor-software/yajta.

Table 1
Characteristics of the studied decompilers.

Decompiler	Version	Status	#Commits	#LOC
CFR (Benfield, 2019)	0.141	Active	1433	52 098
Dava (Jerome Miecznikowski and Naeem, 2019)	3.3.0	2018-06-15[a]	14	22 884
Fernflower (JetBrains, 2019)	NA[b]	Active	453	52 118
JADX (skylot, 2019)	0.9.0	Active	970	55 335
JD-Core (Dupuy, 2019)	1.0.0	Active	NA[c]	36 730
Jode (Hoenicke, 2019)	1.1.2-pre1	2004-02-25[a]	NA[c]	30 161
Krakatau (Storyyeller, 2019)	NA[b]	2018-05-13[a]	512	11 301
Procyon (Strobel, 2019)	0.5.34	Active	1080	122 147

[a]Date of last update.
[b]Not following any versioning scheme.
[c]CVS not available at the date of the present study.

Table 2
Characteristics of the projects used to evaluate decompilers.

Project name	Java version	#Classes	#Tests	#LOC
Bukkit	1.6	642	906	60 800
Commons-codec	1.6	59	644	15 087
Commons-collections	1.5	301	15 067	62 077
Commons-imaging	1.5	329	94	47 396
Commons-lang	1.8	154	2581	79 509
DiskLruCache	1.5	3	61	1206
JavaPoet[a]	1.6	2	60	934
Joda time	1.5	165	4133	70 027
Jsoup	1.5	54	430	14 801
JUnit4	1.5	195	867	17 167
Mimecraft	1.6	4	14	523
Scribe Java	1.5	89	99	4294
Spark	1.8	34	54	4089
DcTest[b]	1.5–1.8	10	9	211
Total		2041	25 019	378 121

[a]Formerly named JavaWriter.
[b]Examples collected from previous decompilers evaluation.

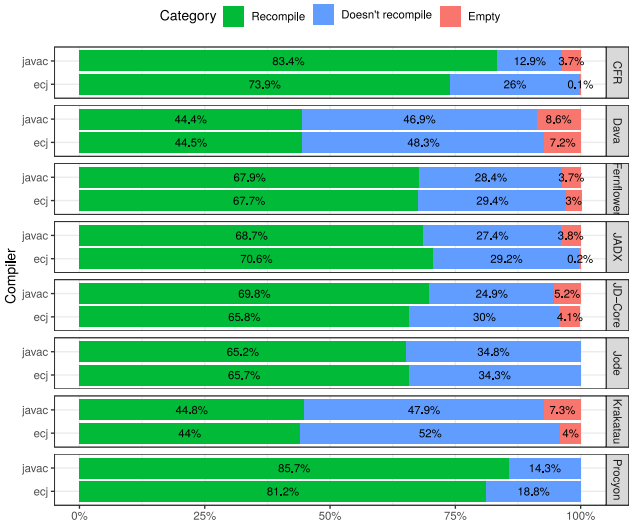

Fig. 2. Outcome of the decompilation for each pair of compiler and decompiler for the 1887 classes compilable by both *ecj* and *javac*. From left to right are presented the percentages of classes that were syntactically correct (in green), syntactically incorrect (in blue), and empty (in red). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

4. Experimental results

4.1. RQ1: (syntactic correctness) To what extent is decompiled Java code syntactically correct?

This research question investigates to what extent the source code produced by the different decompilers is syntactically correct, meaning that the decompiled code compiles. We also investigate the effect of the compiler that produces the bytecode on the decompilation results. To do so, in this section, we focus on the 1887 classes that compile with both *javac* and *ecj*.

Fig. 2 shows the ratio of decompiled classes that are syntactically correct per pair of compiler and decompiler. The horizontal axis shows the ratio of syntactically correct output in green, the ratio of syntactically incorrect output in blue, and the ratio of empty output in red (an empty output occurs, e.g. when the decompiler crashes). The vertical axis shows the compiler on the left and decompiler on the right. For example, Procyon, shown in the last row, is able to produce syntactically correct source code for 1609 (85.3%) class files compiled with *javac*, and produce a non-empty syntactically incorrect output for 278 (14.7%) of them. On the other hand, when sources are compiled with *ecj*, Procyon generates syntactically correct sources for 1532 (81.2%) of class files and syntactically incorrect for 355 (18.8%) sources. In other words, Procyon is slightly more effective when used against code compiled with *javac*. It is interesting to notice that not all decompiler authors have decided to handle error the same way. Both Procyon and Jode's developers have decided to always return source files, even if incomplete (for our dataset). Additionally, when CFR and Procyon detect a method that they cannot decompile properly, they may replace the body of the method by a single `throw` statement and comment explaining the error. This leads to syntactically correct code, but not semantically equivalent.

```
1 public interface Bag<E> extends Collection<E> {
2 +   @Override
3 +   public boolean add(E var1);
4 -   public boolean add(E Object);
5     [...]
6 }
```

Listing 5: Excerpt of differences in Bag original (in red marked with a -) and decompiled with CFR (in green marked with a +).

The ratio of syntactically correct decompiled code ranges from 85.7% for Procyon on *javac* inputs (the best), down to 44% for Krakatau on *ecj* (the worst). All decompilers failed to produce syntactically correct output for 137 classes. Overall, no decompiler is capable of correctly handling the complete dataset. This illustrates the challenges of Java bytecode decompilation, even for bytecode that has not been obfuscated.

We note that syntactically incorrect decompilation can still be useful for reverse engineering. However, an empty output is useless: the ratio of class files for which the decompilation completely fails is never higher than 8.6% for Dava on *javac* bytecode.

In the following paragraphs we investigate the impact of the compiler on decompilation effectiveness. Over the 10912 syntactically incorrect decompilation (over all decompilers), 712 can

```
//Bytecode
   NEW Lang$LangRule
   DUP
   ALOAD 10 //pattern
   NEW HashSet
   DUP
   ALOAD 11 //langs
   INVOKESTATIC Arrays.asList
   invokespecial HashSet.<init>
   ILOAD 12 // accept
   ACONST_NULL
-  INVOKESPECIAL Lang$LangRule.<init>
-    (LPattern;LSet;Z LLang$1)V
+  INVOKESPECIAL Lang$LangRule.<init>
+    (LPattern;LSet;Z LLang$LangRule)V

//Decompiled sources
//Usage of private static inner class LangRule
      constructor's wrapper generated by the compiler
-  new LangRule(pattern, new HashSet<String>(
-        Arrays.asList(langs)), accept)
+  new LangRule(pattern, new HashSet(
+        Arrays.asList(langs)), accept, null)
```

Listing 6: Excerpt of differences when compiling `org/apache/commons/codec/language/bm/Lang` with *javac* (in red marked with a -) and with *ecj* (in green marked with a +). Top part of listing illustrates differences in bytecode, while bottom part shows differences in source code decompiled by Procyon.

be attributed to compiler differences because the decompiler produces syntactically incorrect sources for one compiler and not for the other. These cases break down as follows: 596 failures occur only on *ecj* bytecode and 116 cases only on *javac* code.

Listing 5 shows an excerpt of the differences between the original source code of the `Bag` interface from `commons-collections` and its decompiled sources produced by CFR. This is an example where both *javac* and *ecj* produce the same bytecode, yet recompilation of the sources produced by CFR succeed with *javac* and fail with *ecj*. The `commons-collections` library is compiled targeting Java 1.5, in our experiment. In these conditions, *ecj* fails in the presence of the `@Override` annotation to override a method inherited from an interface such as `Collection`. It is only accepted for inheritance from a class (abstract or concrete). Meanwhile, *javac* compiles the decompiled sources without any error. Note that specifying Java 1.6 as target solves the error with *ecj*. This illustrates how the notion of syntactic correctness depends on the actual compiler as well as on the targeted Java version.

Listing 6 shows an excerpt of the bytecode generated by *javac* and *ecj* for class `org/apache/commons/codec/language/bm/Lang` as well as the corresponding decompiled sources generated by Procyon in both cases. The excerpt shows a call to the `private` constructor of a `private static` nested class of `Lang` called `LangRule`. Since the nested class is `static` the outer class and the inner class interact as if both were top-level classes[9] in the bytecode. But as the constructor of the nested class is `private`, the enclosing class cannot access it. To bypass this problem, both *javac* and *ecj* create a synthetic `public` wrapper for this constructor. In Java bytecode, a synthetic element is an element created by the compiler that does not correspond to any element present in the original sources (implicitly or not). This wrapper is a public constructor for the nested class `LangRule`. As the signature for this wrapper cannot be the same as the private constructor, it must have different parameters. *javac* and *ecj* handle this case differently. *javac* creates a synthetic anonymous class `Lang$1` and adds an additional parameter typed with this anonymous class to the wrapper parameters. Since the value of this parameter is never used, null is passed as additional parameter when the wrapper is called. *ecj* does almost the same thing, but the additional parameter is of type `Lang$LangRule`. Procyon is able to reverse the *javac* transformation correctly, but not the *ecj* one. It decompiles the *ecj* version literally, conserving the null parameter. Yet, the synthetic wrapper does not exist in the decompiled sources. Consequently, the decompiled code that refers to an absent constructor is syntactically incorrect. This example illustrates the decompilation challenge introduced by synthetic elements that are generated by a compiler. Note that synthetic elements in bytecode do carry a flag indicating their nature. But, (i) it does not change the difficulty of reversing an unforeseen pattern and (ii) this flag could be abused by an obfuscator as it does not change the semantic of the bytecode but rather gives indication to drive potential tools modifying the bytecode. In the case of Procyon and CFR, their common test suite did not include code generated by *ecj* at the time of this experiment. Since then, the author of CFR has updated the test suite common to CFR and Procyon with test covering bytecode from *ecj*[10] We have shown that the compiler impact decompilation effectiveness for two reasons: (i) it changes the oracle for syntactic correctness; (ii) it produces different bytecode structure that decompilers might not expect.

To assess the significance of this impact, we use a χ^2 test on the ratio of classfiles decompiled into syntactically correct source code depending on the compiler, *javac* versus *ecj*. The compiler variable has an impact for three decompilers and no impact for the remaining five, with 99% confidence level. The test rejects that the compiler has no impact on the decompilation syntactic correctness ratio for CFR, Procyon and JD-Core (p-value 10^{-14}, 0.00027 and 0.006444). For the five other decompilers we do not observe a significant difference between *javac* and *ecj* (p-values: Dava 0.15, Fernflower 0.47, JADX 0.17, Jode 0.50, and Krakatau 0.09). Note that beyond syntactic correctness, the compiler may impact the correctness of the decompiled code, this is discussed in more details in Section 4.3.

To sum up, Procyon and CFR are the decompilers that score the highest on syntactic correctness. The three decompilers ranking the lowest are Jode, Krakatau and Dava. It is interesting to note that those three are no longer actively maintained.

> **Answer to RQ1:** No single decompiler is able to produce syntactically correct sources for more than 85.7% of class files in our dataset. The implication for decompiler users is that decompilation of Java bytecode cannot be blindly applied and does require some additional manual effort. Only few cases make all decompilers fail, which suggests that using several decompilers in conjunction could help to achieve better results.

4.2. RQ2: (semantic equivalence) To what extent is decompiled Java code semantically equivalent modulo inputs?

To answer this research question, we focus on the 2397 class files, regrouping bytecode generated by both *javac* and *ecj*, that are covered by at least one test case. This excludes all types which contain no executable code, such as interfaces. When decompilers produce sources that compile, we investigate the semantic equivalence of the decompiled source and their original. To do so,

[9] https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html.

[10] Commit: https://github.com/leibnitz27/cfr_tests/commit/b4f0b01e34a953a1fd57e52f508c9c02c58e6deeDiscussion: https://github.com/leibnitz27/cfr/issues/50.

```
-    IFEQ L2
+    IFNE L2
+    GOTO L0
+ L2
     ALOAD 5
     INVOKESTATIC Lang$LangRule.access$100
          (LLang$LangRule;)Z
     IFEQ L3
     ALOAD 3
     ALOAD 5
     INVOKESTATIC Lang$LangRule.access$200
          (Lang$LangRule;)Ljava/util/Set;
-    INVOKEINTERFACE Set.retainAll (LCollection;)Z
+    INVOKEVIRTUAL HashSet.retainAll (LCollection;)Z
     (itf)
     POP
-    GOTO L2
+    GOTO L0
```

Listing 7: Excerpt of bytecode from class `org/apache/commons/codec/language/bm/Lang.class`, compiled with *javac* and decompiled with CFR: Lines in red, marked with a -, are in the original bytecode, while lines in green, marked with a +, are from the recompiled sources.

we split recompilable outputs in three categories: (i) *semantically equivalent:* the code is recompiled into bytecode that is strictly identical to the original (modulo reordering of the constant pool, as explained in Section 3.1), (ii) *semantically equivalent modulo inputs:* the output is recompilable and passes the original project's test suite (i.e. we cannot prove that the decompiled code is semantically different), and (iii) *semantically different:* the output is recompilable but it does not pass the original test suite (deceptive decompilation, as explained in Definition 4).

Let us first discuss an example of semantic equivalence of decompiled code. Listing 7 shows an example of bytecode that is different when decompiled–recompiled but equivalent modulo inputs to the original. Indeed, we can spot two differences: the control flow blocks are not written in the same order (`L2` becomes `L0`) and the condition evaluated is reversed (`IFEQ` becomes `IFNEQ`), which leads to an equivalent control flow graph. The second difference is that the type of a variable originally typed as a `Set` and instantiated with an `HashSet` has been transformed into a variable typed as an `HashSet`, hence once `remainAll` is invoked on the variable `INVOKEINTERFACE` becomes directly `INVOKEVIRTUAL`. This example illustrates how bytecode may be changed (here with a slightly different but equivalent control flow graph and a change in the precision of the type information) yet still be semantically equivalent modulo inputs.

Now we discuss the results globally. Fig. 3 shows the recompilation outcomes of decompilation regarding semantic equivalence for the 2397 classes under study. The horizontal axis shows the eight different decompilers. The vertical axis shows the number of classes decompiled successfully. Strictly equivalent output is shown in blue, equivalent classes modulo input are shown in orange. For example, CFR (second bar) is able to correctly decompile 1713 out of 2397 classes (71%), including 1114 classes that are recompilable into strictly equivalent bytecode, and 599 that are recompilable into equivalent bytecode modulo inputs. When the decompiler fails to produce an output that is semantically equivalent modulo inputs, it is either because the decompiled sources were not syntactically correct and did not recompile, or because they did recompile but did not pass the original test suite (deceptive decompilation). Table 3 gives this information for each decompiler.

The three decompilers that are not actively maintained any more (Jode, Dava and Krakatau) handle less than 50% of the cases correctly (recompilable and pass tests). On the other hand, Procyon and CFR have the highest ratio of equivalence modulo inputs of 78% and 71%, respectively.

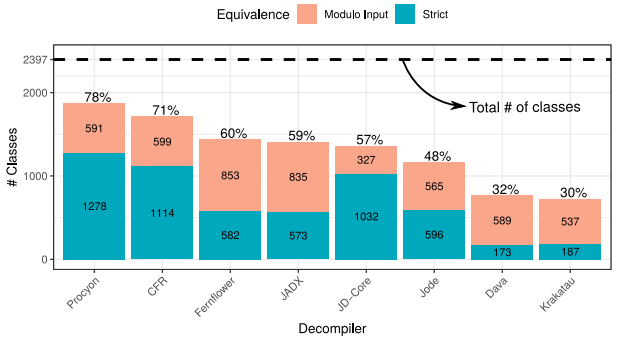

Fig. 3. Equivalence results for each decompiler on all the 2397 classes of the studied projects covered by at least one test, (both compilers combined). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Table 3
Cause of non-equivalence for each decompiler.

	Procyon	CFR	Fernflower	JADX	JD-Core	Jode	Dava	Krakatau
#Deceptive	33	22	21	78	44	142	36	97
#!Recompile	495	662	941	911	994	1094	1599	1576
#Failures	528	684	962	989	1038	1236	1635	1673

Answer to RQ2: The number of classes for which the decompiler produces equivalent sources modulo input varies significantly from one decompiler to another. The result of decompilation is usually not strictly identical to the original source code. Five decompilers generate equivalent modulo input source code for more than 50% of the classes. For end users, it means that the state of the art of Java decompilation does not guarantee semantically correct decompilation, and care must be taken not to blindly trust the behavior of decompiled code.

4.3. RQ3: (bug finding) To what extent do decompilers produce deceptive decompilation results?

As explained by Hamilton and colleagues (Hamilton and Danicic, 2009), while a syntactically incorrect decompilation output may still be useful to the user, syntactically correct but semantically different output is more problematic. Indeed, this may mislead the user by making her believe in a different behavior than the original program has. We call this case *deceptive decompilation* (as explained in Definition 4). When such cases occur, since the decompiler produces an output that is semantically different from what is expected, they may be considered decompilation bugs.

Fig. 4 shows the distribution of bytecode classes that are deceptively decompiled. Each horizontal bar groups deceptive decompilation per decompiler. The color indicates which compiler was used to produce the class file triggering the error. In blue is the number of classes leading to a decompilation error only when compiled with *javac*, in green only when compiled with *ecj*, and in pink is the number of classes triggering a decompilation error with both compilers. The sum of these classes is indicated by the total on the right side of each bar. Note that the bars in Fig. 4 represent the number of bug manifestations, which are not necessarily distinct bugs: the same decompiler bug can be triggered by different class files from our benchmark. Also, Fig. 4

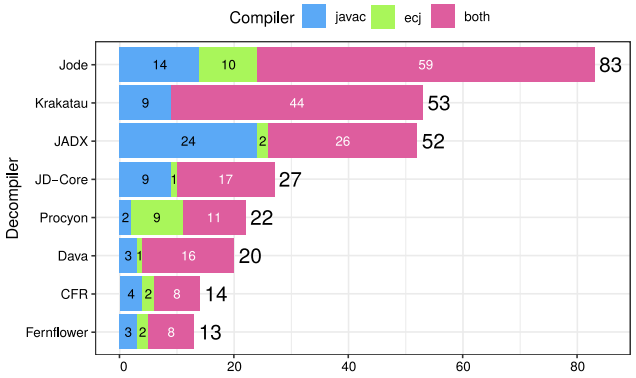

Fig. 4. Deceptive decompilation results per decompiler. From left to right, deceptive decompilation occurring after *javac* compilation, *ecj* and both. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

plots the same classes referred as `Deceptive` in Table 3, but in Table 3 classes leading to a deceptive decompilation for both compilers are counted twice.

Overall, Jode is the least reliable decompiler, with 83 decompilation bug instances in our benchmark. While Fernflower produces the least deceptive decompilations on our benchmark (13), it is interesting to note that CFR produces only one more deceptive decompilation (14) but that corresponds to fewer bugs per successful decompilation. This makes CFR the most reliable decompiler on our benchmark.

We manually inspected 10 of these bug manifestations. 2 of them were already reported by other users. We reported the other 8 to the authors of decompilers.[11] The sources of errors include incorrect cast operation, incorrect control-flow restitution, auto unboxing errors, and incorrect reference resolution. Below we detail two of these bugs.

4.3.1. Case study: incorrect reference resolution

We analyze the class `org.bukkit.Bukkit` from the Bukkit project. An excerpt of the original Java source code is given in Listing 8. The method `setServer` implements a setter of the static field `Bukkit.server`. This is an implementation of the common Singleton design pattern. In the context of method `setServer`, `server` refers to the parameter of the method, while `Bukkit.server` refers to the static field of the class `Bukkit`.

When this source file is compiled with *javac*, it produces a file `org/bukkit/Bukkit.class` containing the bytecode translation of the original source. Listing 9 shows an excerpt of this bytecode corresponding to the `setServer` method (including lines are filled in red, while excluding lines are filled in green)

When using the JADX decompiler on `org/bukkit/Bukkit.class` it produces decompiled sources, of which an excerpt is shown in Listing 8. In this example, the decompiled code is not semantically equivalent to the original version. Indeed, inside the `setServer` method the references to the static field `Bukkit.server` have been simplified into `server` which is incorrect in this scope as the parameter `server` overrides the local scope. In the bytecode of the recompiled version (Listing 9, including lines are filled in green), we can observe that instructions accessing and writing the static field (`GETSTATIC`, `PUTSTATIC`) have been replaced by instructions accessing and writing the local variable instead (`ALOAD`, `ASTORE`).

When the test suite of *Bukkit* runs on the recompiled bytecode, the 11 test cases covering this code fail, as the first access to `setServer` will throw an exception instead of normally initializing the static field `Bukkit.server`. This is clearly a bug in JADX.

[11] https://github.com/castor-software/decompilercmp/tree/master/funfacts.

```
public final class Bukkit {
    private static Server server;
        [...]
    public static void setServer(Server server) {
-       if (Bukkit.server != null) {
+       if (server != null) {
            throw new UnsupportedOperationException(
            "Cannot redefine singleton Server");
        }
-       Bukkit.server = server;
+       server = server;
        [...]
    }
```

Listing 8: Excerpt of differences in `org.bukkit.Bukkit` original (in red marked with a -) and decompiled with JADX sources (in green marked with a +).

```
public static setServer(Lorg/bukkit/Server;)V
-    GETSTATIC org/bukkit/Bukkit.server :
-    Lorg/bukkit/Server;
+    ALOAD 0
     IFNULL L0
     NEW java/lang/UnsupportedOperationException
     DUP
     ATHROW
    L0
     ALOAD 0
-    PUTSTATIC org/bukkit/Bukkit.server :
-    Lorg/bukkit/Server;
+    ASTORE 0
     ALOAD 0
     INVOKEINTERFACE org/bukkit/Server.getLogger
          ()Ljava/util/logging/Logger; (itf)
     NEW java/lang/StringBuilder
```

Listing 9: Excerpt of bytecode from class `org/bukkit/Bukkit.class` compiled with *javac*: Lines in red, marked with a -, are in the original bytecode, while lines in green, marked with a +, are from the recompiled sources (decompiled with JADX).

```
    protected StringBuffer applyRules(final Calendar
         calendar, final StringBuffer buf) {
-       return (StringBuffer) applyRules(calendar,
-                       (Appendable) buf);
+       return this.applyRules(calendar, buf);
    }

    private <B extends Appendable> B applyRules(final
         Calendar calendar, final B buf) {...}
```

Listing 10: Excerpt of differences in `FastDatePrinter` original (in red marked with a -) and decompiled with Procyon sources (in green marked with a +).

4.3.2. Case study: Down cast error

Listing 10 illustrates the differences between the original sources of `org/apache/commons/lang3/time/FastDatePrinter` and the decompiled sources produced by Procyon. The line in red is part of the original, while the line in green is from the decompiled version. In this example, method `applyRules` is overloaded, i.e. it has two implementations: one for a `StringBuffer` parameter and one for a generic `Appendable` parameter (`Appendable` is an interface that `StringBuffer` implements). The implementation for `StringBuffer` down casts `buf` into `Appendable`, calls the method handling `Appendable` and casts the result back to `StringBuffer`. In a non-ambiguous context, it is perfectly valid to call a method which takes `Appendable` arguments on an instance of a class that implements that interface. But in this context, without the down cast to `Appendable`, the Java compiler will resolve the method call `applyRules` to the most concrete method. In this case, this will lead `applyRules` for `StringBuffer` to call itself instead of the other method.

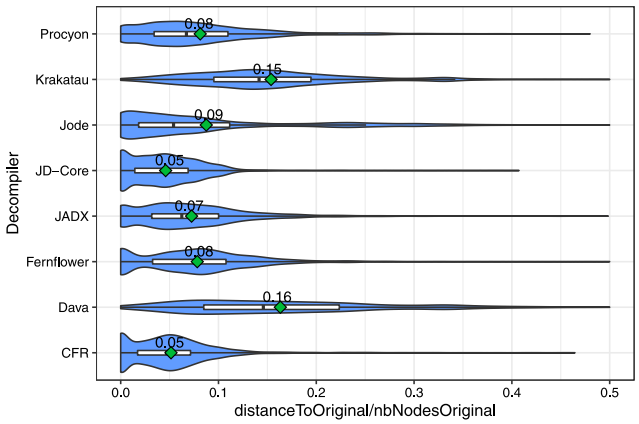

Fig. 5. Distribution of differences between the original and the decompiled source code ASTs. Green diamonds indicate average.

When executed, this will lead to an infinite recursion ending in a StackOverflowError. Therefore, in this example, Procyon changes the behavior of the decompiled program and introduces a bug in it.

> **Answer to RQ3:** Our empirical results indicate that no decompiler is free of deceptive decompilation bugs. The developers of decompilers may benefit from the equivalent modulo input concept to find bugs in the wild and extend their test base. Two bugs found during our study have already been fixed by the decompiler authors, and three others have been acknowledged.

4.4. RQ4: (ASTs difference) What is the syntactic distortion of decompiled code?

The quality of decompilation depends not only on its syntactic correctness and semantic equivalence but also on how well a human can understand the behavior of the decompiled program. The code produced by a decompiler may be syntactically and semantically correct but yet hard to read for a human. In this research question, we evaluate how far the decompiled sources are from the original code. We measure the syntactic distortion between the original and the decompiled sources as captured by AST differences (Definition 2).

Fig. 5 shows the distribution of syntactic distortion present in syntactically correct decompiled code, with one violin plot per decompiler. The green diamond marks the average syntactic distortion. For example, the syntactic distortion values of the Jode decompiler have a median of 0.05, average of 0.09, 1st-Q and 3rd-Q of 0.01 and 0.11, respectively. In this figure, lower is better: a lower syntactic distortion means that the decompiled sources are more similar to their original counterparts.

CFR and JD-Core introduce the least syntactic distortion, with high proportion of cases with no syntactic distortion at all (as we exclude renaming). Their median and average syntactic distortion are close to 0.05, which corresponds to 5 edits every 100 nodes in the AST of the source program. On the other extreme, Dava and Krakatau introduce the most syntactic distortion with average of 16 (resp. 15) edits per 100 nodes. They also have almost no cases for which they produce sources with no syntactic distortion. It is interesting to note that Dava makes no assumptions on the source language nor the compiler used to produce the bytecode it decompiles (Miecznikowski and Hendren, 2002). This partly explains the choice of its author to not reverse some optimizations made by Java compilers. (See example introduced in Section 2.)

```
1  public class Foo {
2    public int foo(int i, int j) {
3      while (true) {
4        try {
5          while (i < j) i = j++ / i;
6  +       return j;
7        } catch (RuntimeException re) {
8          i = 10;
9  -       continue;
10       }
11 -     break;
12     }
13 -   return j;
14   }
15 }
```

Listing 11: Excerpt of differences in Foo original (in red marked with a -) and decompiled with Fernflower (in green marked with a +) sources.

Listing 11 shows the differences on the resulting source code after decompiling the Foo class from DcTest with Fernflower. As we can observe, both Java programs represent a semantically equivalent program. Yet, their ASTs contain substantial differences. For this example, the edit distance is 3/104 as it contains three tree edits: MOVE the return node, and DELETE the break node and the continue node (the original source's AST contained 104 nodes).

Note that some decompilers perform some transformations on the sources they produce on purpose to increase readability. Therefore, it is perfectly normal to observe some minimal syntactic distortion, even for decompilers producing readable sources. But as our benchmark is composed of non obfuscated sources, it is expected that a readable output will not fall too far from the original.

> **Answer to RQ4:** All decompilers present various degrees of syntactic distortion between the original source code and the decompiled bytecode. This reveals that all decompilers adopt different strategies to craft source code from bytecode. We propose a novel metric to quantify the distance between the original source code and its decompiled counterpart. Also, decompiler users can use this analysis when deciding which decompiler to employ.

4.5. RQ5: (Decompiler Diversity) To what extent do the successes and failures of decompilers overlap?

In the previous research questions, we observe that different decompilers produce source code that varies in terms of syntactic correctness, semantic equivalence and syntactic distortion. Now, we investigate the overlap in successes and failures of the different decompilers considered for this study.

Fig. 6 shows a Venn Diagram of semantically equivalent classes modulo inputs for decompiled/recompiled classes. We exclude Dava and Krakatau because they do not handle correctly any class file unique to them. We see that 6/8 decompilers have cases for which they are the only decompiler able to handle it properly. These cases represent 276/2397 classes. Only 589/2397 classes are handled correctly by all of these 6 decompilers. Furthermore, 157/2397 classes are not correctly handled by any of the considered decompilers.

Listing 12 is an excerpt of AbstractHashedMap, which is incorrectly decompiled by all decompilers. While the complete set of syntactic errors for the decompiled sources is different for each decompiler, it always includes one call of the constructor of the super class of KeySetIterator<K>. Either a constructor

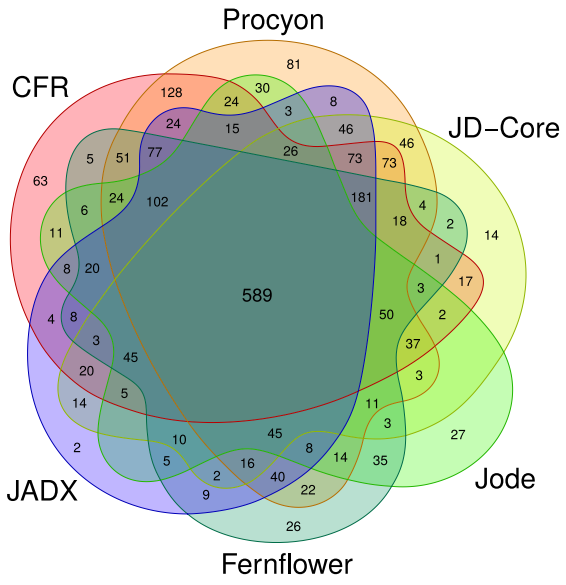

Fig. 6. Venn diagram of syntactically and semantically equivalent modulo inputs decompilation results.

```
protected static class KeySetIterator<K> extends
      HashIterator<K, Object> implements Iterator<K> {
-   @SuppressWarnings("unchecked")
        protected KeySetIterator(final
              AbstractHashedMap<K, ?> parent) {
-     super((AbstractHashedMap<K, Object>) parent);
+     super(parent);
        }
    [...]
}
```

Listing 12: Excerpt of differences in `AbstractHashedMap` original (in red marked with a -) and decompiled with Procyon (in green marked with a +).

with the correct signature is not resolved or the cast in front of `parent` is missing. The fundamental problem behind this decompilation lies in the fact that the JVM does not directly support generics (Amin and Tate, 2016). While bytecode do keep meta information about types in signatures, the actual type manipulated in this example for ? is an `Object`. Therefore, contrarily to the original sources, no `CHECKCAST` instruction is required in the bytecode. This does not make the task of decompilation impossible to perform in theory, as both the type of `parent` and the type required by the `super` constructor can be resolved, but, it does make it more challenging to decompilers in practice.

A manual analysis of these classes shows common issues among the studied decompilers. (i) Generics is a feature that causes many decompilers to fail in particular when combined with ternary operators, wildcards or type bounds. Another example of such a case is detailed in Section 5. (ii) As mentioned in Sections 4.1 and 4.3, compilers producing the bytecode do play a role. In particular, synthetic elements created by a compiler, which the decompiler does not expect. (iii) Overall, the diversity of independent corner cases cannot be completely captured under one concise explanation. Even for Procyon, the best performing decompiler in our study, among the 528 classes for which it does not produce semantically equivalent modulo inputs sources, only 157 are also not decompilable by any other decompiler.

Table 4 summarizes the quantitative results obtained from the previous research questions. Each line corresponds to a decompiler. Column `#Recompilable` shows the number of cases (and ratio) for which the decompiler produced a recompilable

Table 4
Summary results of the studied decompilers.

Decompiler	#Recompilable	#PassTest	#Deceptive
CFR	3097 (0.79)	1713 (0.71)	22
Dava	1747 (0.44)	762 (0.32)	36
Fernflower	2663 (0.68)	1435 (0.60)	21
JADX	2736 (0.70)	1408 (0.59)	78
JD-Core	2726 (0.69)	1375 (0.57)	44
Jode	2569 (0.65)	1161 (0.48)	142
Krakatau	1746 (0.44)	724 (0.30)	97
Procyon	3281 (0.84)	1869 (0.78)	33
Union	3734 (0.95)	2240 (0.93)	342
Total	3928 (1.00)	2397 (1.00)	–

output among all classes of our dataset (3928 in total: 2041 for *javac* and 1887 for *ecj*). Column `#PassTest` shows the number of cases where the decompiled code passes those tests among the 2397 classes covered by tests and regrouping both compiler. Column `#Deceptive` indicates the number of cases that were recompilable but did not pass the test suite (i.e. a decompilation bug). The line 'Union' shows the number of classes for which at least 1 decompiler succeeds to produce `Recompilable` sources and respectively sources that `pass tests`. The column `#Deceptive` indicates the number of classes for which at least 1 decompiler produced a deceptive decompilation. This means that for 2240 classes out of the 2397 (93%), there is at least 1 decompiler that produces semantically equivalent sources modulo inputs. This number must be taken with a grain of salt, as it does not mean that someone who looks for a successful decompilation of one of these classes could find one trivially. Overall, 342 out of 2397 classes have at least 1 decompiler that produce a deceptive decompilation. Assuming that one can merge the successful decompilation results together, we would obtain a better decompiler overall, this is what we explore in Section 5.

> **Answer to RQ5:** The classes for which each decompiler produce semantically equivalent source code modulo input do not overlap completely. For 6 out of 8 decompilers, there exists at least 1 class for which the decompiler is the only one to produce semantic equivalence modulo inputs sources. In theory, a union of the best features of each decompiler would cover 2240 out of the 2397 (93%) classes of the dataset. This suggests to combine multiple decompilers to improve decompilation results.

5. Meta decompilation

In this section, we present an original concept for decompilation.

5.1. Overview

In 1995, Selberg and Etzioni (1997) noticed that different web search engines produced different results for the same input query. They exploited this finding in a tool called MetaCrawler, which delegates a user query to various search engines and merges the results. This idea of combining diverse tools that have the same goal has been explored since then. For example, Foster and Somayaji (2010) explore how a genetic algorithm can recombine related programs at the object file level to produce correct variants of C programs. Persaud et al. (2016) combines cryptographic libraries together for software security. Chen et al. (2018) rely on various fuzzers to build an ensemble based fuzzer that gets better performance and generalization ability than that of any constituent fuzzer alone.

```
1  public class YamlConfiguration extends
       FileConfiguration {
2      protected static final String COMMENT_PREFIX = "# ";
3      protected static final String BLANK_CONFIG = "{}
4  ";
5      private final DumperOptions yamlOptions;
6      private final Representer yamlRepresenter;
7      private final Yaml yaml;
8
9      public YamlConfiguration()
10     {
11         DumperOptions r7;
12         YamlRepresenter r8;
13         YamlConstructor r9;
14         Yaml r10;
15         BaseConstructor r11;
16         r7 = new DumperOptions();
17         yamlOptions = r7;
18         r8 = new YamlRepresenter();
19         yamlRepresenter = r8;
20         r9 = new YamlConstructor();
21         r11 = (BaseConstructor) r9;
22         r10 = new Yaml(r11, yamlRepresenter,
               yamlOptions);
23         yaml = r10;
24     }
25   [...]
26 }
```

Listing 13: Excerpt of `org.bukkit.configuration.file.YamlConfiguration` decompiled with Dava.

```
1  public class YamlConfiguration extends
       FileConfiguration {
2      protected static final String BLANK_CONFIG = "{}\n";
3      protected static final String COMMENT_PREFIX = "# ";
4      private final Yaml yaml =
5          new Yaml(new YamlConstructor(),
6          this.yamlRepresenter, this.yamlOptions);
7      private final DumperOptions yamlOptions = new
           DumperOptions();
8      private final Representer yamlRepresenter = new
           YamlRepresenter();
9    [...]
10 }
```

Listing 14: Excerpt of `org.bukkit.configuration.file.YamlConfiguration` decompiled with JADX.

In this paper, we apply a similar approach to improve Java decompilation. Each decompiler has its strengths and weaknesses, and the subset of JVM bytecode sequences they correctly handle is diverse (cf. Section 4.5). Therefore, our idea is to combine decompilers in a meta-decompiler.

In this paper, we propose a tool called Arlecchino, that implements such a 'meta-decompilation' approach. Arlecchino merges partially incorrect decompilation results from diverse decompilers in order to produce a correct one.

5.2. Example

The class `org.bukkit.configuration.file.YamlConfiguration` of the project Bukkit is an example of a class file that is incorrectly handled by both JADX and Dava. While both decompilers produce syntactically incorrect Java code for this class, the error that prevents successful recompilation is not located at the same place in both decompiled classes.

Listing 13 shows an excerpt of the decompiled sources produced by Dava for `YamlConfiguration`. The static field `BLANK_CONFIG` is initialized with an incorrect string literal that contains a non escaped line return. When attempting to recompile these sources, *javac* produces an `unclosed string literal` error for both line 3 and 4.

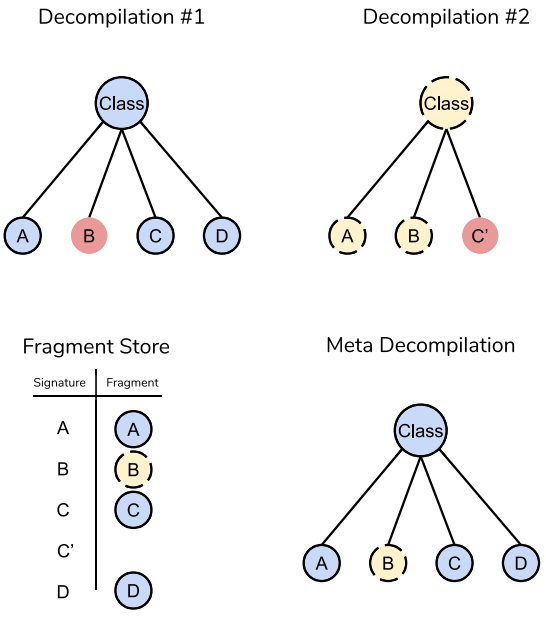

Fig. 7. Meta decompilation: Merger of different partial decompilation. Node in blue with plain border originates from Decompiler #1, nodes in yellow with dashed border originate from Decompiler #2. Borderless nodes in red contains compilation errors. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

Listing 14 shows an excerpt of the decompiled sources produced by JADX for the same class. The static field `BLANK_CONFIG` is correctly initialized with `"{}\n"`, but the initialization of `yaml`, `yamlOptions` and `yamlRepresenter` are conducted out of order, which lead to a compilation error as `yamlOptions` and `yamlRepresenter` are still `null` when `yaml` is initialized. Intuitively, one can see that Dava's solution could be fixed by replacing lines 3 and 4 with line 2 from JADX's solution. This is an example of successful meta-decompilation, merging the output of two decompilers.

Fig. 7 illustrates how two erroneous decompilations can be merged into one that is correct, when the error is not located at the same place. This figure represents different versions of the abstract syntax tree (AST) of a Java class. The root node corresponds to the class itself, while its children represent type members of the class. A type member is either a method, a field, a nested type (class, or enum), or a static initialization block. `Decompilation #1` represents the AST of the sources produced by one decompiler, it includes 4 type members (A,B,C and D), and one compilation error located in B. `Decompilation #2` represents the sources produced by a different decompiler for the same class. It contains only 3 type members (A,B an C') and one compilation error located in C'. The fragment store is a dictionary containing an error free AST fragment for each type member when such a fragment exists. `Meta Decompilation` shows an example of error free AST that can be built based on `Decompilation #1` and the store that combines AST fragments from both decompilations. Note that different decompilers may produce sources that do not exactly contain the same type members. This is illustrated here by `Decompilation #2` not having a type member D and having a different signature for C.

5.3. Algorithm

Algorithm 1 describes the process of meta decompilation as implemented by Arlecchino. Arlecchino takes as input a bytecode

```
Data: bytecode A bytecode file,
Decompilers A set of decompilers
Result: The decompiled java sources corresponding
1  Solutions ← {}
2  FragmentStore ← {}
3  foreach dc ∈ Decompilers do
4      solution ← AST(decompile(dc, bytecode))
5      Fragments ← fragmentsOf(solution)
6      foreach f ∈ Fragments do
7          if ¬problem(f) ∧ signature(f) ∉ Store then
8              FragmentStore ←
               FragmentStore ∪ {signature(f) → f}
9          end
10     end
11     Solutions ← Solutions ∪ {solution}
12     foreach s ∈ Solutions do
13         if completable(s, FragmentStore) then
14             if recompile(complete(s, FragmentStore)) then
15                 return print(s);
16             else
17                 remove(s, FragmentStore)
18             end
19         end
20     end
21 end
```

Algorithm 1: Meta decompilation procedure.

file, and an ordered list of bytecode decompilers. The process starts with an empty set of solutions and an empty fragment store of correct fragments. This fragment store is a dictionary that associates a type member signature to a fragment of AST free of compilation error corresponding to the type member in question.

For each decompiler, the meta-decompilation goes through the following steps.

The bytecode file is passed to the decompiler *d*. An AST is built from the decompiled sources (line 4). While building the AST, the compilation errors and their location are gathered (if any) and the type members containing errors are annotated as such. A class abstract syntax tree includes a node for the class itself as the root, as well as children representing class information (super class, super interfaces, formal type parameters) and type members. Type members include fields, methods, constructors, inner classes, enum values, and static blocks. These type members' source locations are recorded and compared with the compiler error locations. If an error is located between a type member start and end location, the type member is annotated as errored. For example, the element corresponding to the field `BLANK_CONFIG` is annotated as errored in Dava's solution for `YamlConfiguration`. This annotated AST, that we call solution, is added to the set of remaining solutions.

Additionally, for all type members in the current solution, if the fragment store does not already contain an error free fragment with the same signature, the type member is added to the fragment store (line 8). The signature of a type member is a character string that identifies it uniquely. For example, the signature of the field `BLANK_CONFIG` is `org.bukkit.configuration.file.YamlConfiguration#BLANK_CONFIG` and the signature of `YamlConfiguration`'s constructor is `org.bukkit.configuration.file.YamlConfiguration()`.

Each solution in the set of solutions is checked for completion with the current store (line 12). A solution is "completable" with the members in a given fragment store, if all the solution's type members annotated with an error are present in the fragment store. Indeed, these type members' AST can be replaced with an error free variant present in the fragment store. If a solution is completable with the current fragment store, all its type members annotated as errored are replaced with a fragment from the fragment store. The solution is then passed to the compiler to check if it compiles. If it does, it is printed, and the meta decompilation stops. If not, the solution is removed from the set of solutions. As the first solution that satisfies the oracle (syntactic correctness) stops the process, and this oracle is imperfect, the order of the decompilers matters. More details are given in the following section.

By attempting to repair each solution and its given set of type members with a minimum of transplanted fragments from those available in the fragment store, Arlecchino does not favor any type member set. This allows Arlecchino to deal with cases where the different solutions do not contain the same type members. This occurs with implicit constructor declarations such as the one present in Listing 14 with `YamlConfiguration`. It also makes it possible to handle cases where element signatures might differ depending on how type erasure is dealt with by each decompiler. And finally, it handles cases where elements might not be in the same order (and the order of type members is meaningful as seen in Listing 14).

Table 5
Arlecchino results on classes with no correct decompilation from state of the art decompilers.

	Arlecchino	Union
#PassTest	59 (37.6%)	0 (0%)
#Deceptive	11 (7.0%)	23 (14.6%)
#!Recompile	87 (55.4%)	134 (85.4%)
Total	157 (100%)	157 (100%)

5.4. Experimental results about meta-decompilation

The following section evaluates the effectiveness of Arlecchino. It is organized as follows. First, we gather the 157 classes of our dataset for which no decompilers produced semantically equivalent modulo input sources and assess the results produced by Arlecchino. Second, we run Arlecchino on the complete dataset of classes in this study. We then evaluate the results with regards to semantic equivalence modulo inputs. Finally, we study the origin of fragments produced by Arlecchino and discuss the consequences on the number of deceptive decompilations.

Table 5 shows the results of meta decompilation on the 157 classes of our dataset that led to decompilation errors for all decompilers in the study and were covered by at least one test.[12] Arlecchino produces semantically equivalent results for 59 out of 157 (37.6%) classes. It produces deceptive decompilation for 11 (7.0%) classes and fails to produce recompilable results for 87 out of 157 (55.4%) classes. The success case where Arlecchino produces correct output is when: (1) at least one compiler is able to read the correct signature for all type members of a class and, (2) an error free decompilation exists for all of these type members. However, when no decompiler is able to decompile a specific type member or that no decompiler reads correctly the signature of all type members, no meta decompilation can be successful. These results demonstrate that successful decompilation (in terms of both syntactic correctness and semantic equivalence modulo inputs) can be found by Arlecchino for classes where no other decompilers can.

Table 6 shows the results obtained when running Arlecchino on the whole dataset presented in Section 3 and compares it

[12] 3 of the 137 classes that led to syntactically incorrect outputs for all decompilers are not covered by any tests.

Table 6
Comparison of Arlecchino results with state of the art.

Decompiler	#Recompilable	#PassTest	#Deceptive	ASTDiff
CFR	3097 (79%)	1713 (71%)	22 (1.27%)	0.05
Procyon	3281 (84%)	1869 (78%)	33 (1.74%)	0.08
Arlecchino	3479 (89%)	2087 (87%)	30 (1.42%)	0.06
Total	3928 (100%)	2397 (100%)	–	–

with Procyon and CFR. Procyon is the decompiler that scores the highest in terms of syntactic correctness as well as semantic equivalence modulo inputs, while CFR scores the lowest in deceptive decompilation rate and syntactic distortion. The first column indicates the number of classes for which each decompiler produced syntactically correct sources, among the 3928 from the dataset. The second column shows the number of classes for which each decompiler produced semantically correct modulo inputs sources among the 2397 classes covered by tests. The third column indicates the number of deceptive decompilations produced by each decompiler. The percentage of deceptive decompilation is computed with `#Deceptive / (#Deceptive + #PassTests)`. The last column shows the median syntactic distortion in number of edits per nodes in the original AST.

Arlecchino produced syntactically correct sources for 3479 classes (89%). It produces semantically equivalent modulo inputs sources for 2087 (87%) classes, and 30 deceptive decompilations. Compared with Procyon, Arlecchino produces syntactically correct sources for 198 more classes, semantically correct modulo inputs sources for 218 more. It also produces 3 less deceptive decompilations, and has a lower syntactic distortion. Compared with CFR, Arlecchino produces 8 more deceptive decompilations but it produces semantically correct modulo inputs sources for 374 more classes. In percentage of deceptive decompilation among recompilable decompilation, Arlecchino produces 1.42% of deceptive decompilation which is lower than Procyon's 1.74% but slightly higher than CFR's 1.27%.

Overall, Arlecchino scores higher than all studied decompilers in terms of semantic correctness as well as semantic equivalence modulo inputs, and ranks second in deceptive decompilation rate by a small margin. The rate of semantically equivalent decompilation modulo inputs is higher because Arlecchino produces, by design, more syntactically correct decompilations. On the other hand, the rate of deceptive decompilation is slightly higher than CFR, as Arlecchino aggregates some of the deceptive decompilations from all used decompilers. This is, to our knowledge, the first implementation of this meta-decompilation approach. It demonstrates the validity of the approach and adds a new state of the art tool that practitioners can use to decompile Java bytecode.

Note that Arlecchino also has its implementation flaws and may fail where other decompilers may succeed. In particular, not all AST nodes transplantation produce syntactically correct code. But it may be used in conjunction of other decompilers. The union of classes for which at least one decompiler (including Arlecchino) produces semantically equivalent modulo inputs sources, presented in RQ5, now covers 2299 out of 2397 classes (96%) of our dataset.

5.4.1. Remaining deceptive decompilations

In order to investigate deceptive decompilations produced by Arlecchino, we need to investigate the origins of the AST fragments used in each decompilation.

Fig. 8 shows the distribution of the number of decompilers used by Arlecchino for each of the 2397 classes of our dataset for which we have tests. Arlecchino finds no solution for 166 classes. For 1922 classes, only one decompiler was used, meaning that there is no need for meta-decompilation. For 287 classes,

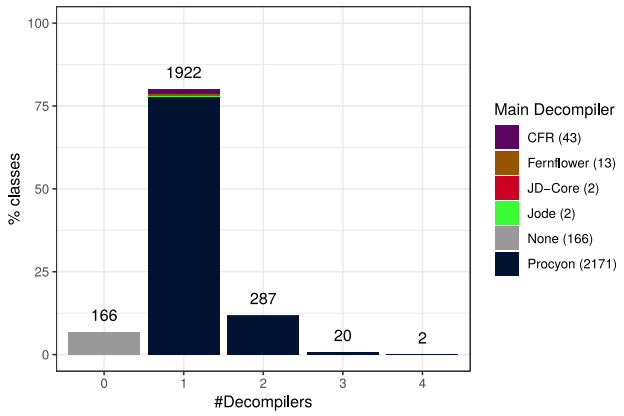

Fig. 8. Distribution of the number of decompilers used by Arlecchino.

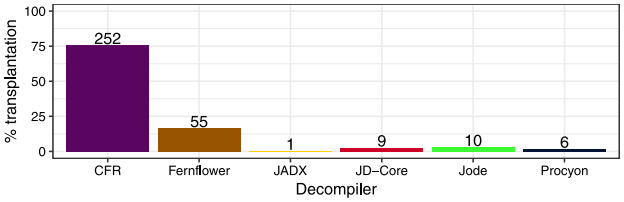

Fig. 9. Distribution of the origin of transplanted fragments in Arlecchino results.

Arlecchino combines the output of 2 decompilers. It uses 3 and 4 decompilers for 20 classes and 2 classes respectively. The color indicates which decompiler's base solution was used. In the overwhelming majority, the Procyon solution is used.

Fig. 9 shows the distribution of transplanted fragments' origin for the 309 classes where several decompilers are used. For 252 classes, one or more fragments from CFR's solution were transplanted to build Arlecchino's solution. 55 classes have fragments coming from Fernflower, and the rest of the distribution is negligible. Note that Arlecchino stops as soon as it finds an admissible solution. Thus, the order of decompilers when building a solution largely impacts this distribution.

Arlecchino produces a deceptive decompilation either when the first recompilable solution of a given type member is a deceptive one, or the assembly of different fragments introduces an error.

In order to minimize these problems, Arlecchino uses Procyon as the first decompiler and orders the other decompilers by their deceptive decompilation rate.

Therefore, most of the decompilers' deceptive decompilations are for the same classes as Procyon's one. In a lesser way, deceptive decompilation originating from type members decompiled with CFR affect Arlecchino when those type member are decompiled with syntactic errors by Procyon. Note that, as no software is free of bugs, the implementation of Arlecchino could also add new sources of error. In practice, as shown by Table 6, the number of deceptive decompilations (30) corresponds to a better deceptive decompilation rate than all decompilers of this study except CFR.

5.4.2. Case studies

Here we discuss two examples in details: one successful and one failed meta decompilation.

Success: Request. Listing 15 shows the decompiled sources for `org.junit.runner.Request` produced by Procyon. In this example, there are ambiguous references because two types share the same simply qualified name: both `org.junit.runners.model` and `org.junit.internal.runners` contain a type

```
import org.junit.runners.model.*;
import org.junit.internal.runners.*;
public abstract class Request {
       [...]
    public static Request classes(final Computer
         computer, final Class<?>... classes) {
        try {
            final AllDefaultPossibilitiesBuilder
                 builder = new
                 AllDefaultPossibilitiesBuilder(true);
            final Runner suite =
                 computer.getSuite(builder, classes);
            return runner(suite);
        }
        catch ( InitializationError e) {
            throw new RuntimeException("Bug in saff's
                 brain: Suite constructor, called as
                 above, should always complete");
        }
    }

    public static Request runner(final Runner runner) {
        return new Request() {
            @Override
            public Runner getRunner() {
                return runner;
            }
        };
    }
}
```

Listing 15: Excerpt of `org.junit.runner.Request` decompiled with Procyon.

```
import org.junit.runners.model.InitializationError;

public abstract class Request {
 [...]

    public static Request classes(Computer computer,
         Class<?> ... classes) {
        try {
            AllDefaultPossibilitiesBuilder builder =
                 new
                 AllDefaultPossibilitiesBuilder(true);
            Runner suite = computer.getSuite(builder,
                 classes);
            return Request.runner(suite);
        }
        catch (InitializationError e) {
            throw new RuntimeException("Bug in saff's
                 brain: Suite constructor, called as
                 above, should always complete");
        }
    }

    public static Request runner(Runner runner) {
        return new Request(){

            public Runner getRunner() {
                return Runner.this;
            }
        };
    }
}
```

Listing 16: Excerpt of `org.junit.runner.Request` decompiled with CFR.

named `InitializationError`, therefore the decompiled sources generated by Procyon lead to a compilation error.

Listing 16 shows the decompiled sources for `org.junit.runner.Request` produced by CFR. These sources contain an error in the body of the static method `runner(Runner)`. Since this method contains an anonymous class, when the original sources are compiled, a synthetic field `runner` is created, by the compiler, for the anonymous class. This field contains the

```
    private final Closure<? super E> iDefault;

    private SwitchClosure(final boolean clone,
        final Predicate<? super E>[] predicates,
        final Closure<? super E>[] closures,
        final Closure<? super E> defaultClosure) {
        super();
        iPredicates = clone ?
             FunctorUtils.copy(predicates) :
             predicates;
        iClosures = clone ? FunctorUtils.copy(closures)
             : closures;
-       iDefault = (Closure<? super E>) (defaultClosure
-           == null ? NOPClosure.<E>nopClosure() :
-           defaultClosure);
+       this.iDefault = (defaultClosure == null ?
+           NOPClosure.nopClosure() : defaultClosure);
    }
```

Listing 17: Excerpt of `org.apache.commons.collections4.functors.SwitchClosure`, original (in red marked with a -) and decompiled (in green marked with a +).

parameter `runner` from the enclosing method. When CFR decompiles the bytecode, it incorrectly replaces the statement that returns the parameter of the enclosing method by a statement that returns a field that does not exist in the sources. This leads to a compilation error when attempting to recompile. Since our report, CFR's author has fixed this bug.[13]

While both Procyon and CFR's solutions contain an error, these errors are not located on the same type member. Hence, CFR's fragment for the method `classes(Computer, Class<?>[])` is transplanted on Procyon's solution. Since the pretty printer used by Arlecchino only lists imports at a type granularity, and CFR's fragment contains references that are non-ambiguous, the combined solution is recompilable and semantically equivalent modulo input.

Failure: Switchclosure. There are Java constructs for which all decompilers struggle. In these cases, all decompilers may produce an error on the same type member, and this leads to a failed meta-decompilation. The following example illustrates the problem of generic type lower bounds, which challenges all decompilers.

Listing 17 shows an excerpt of the original sources for `org.apache.commons.collections4.functors.SwitchClosure`. The line highlighted in red is the original line. The line highlighted in green is the corresponding line as decompiled by Procyon, CFR and JD-Core. None of them is able to correctly reproduce the cast to `Closure<? super E>`. This leads to a compilation error as the method `NOPClosure.<E>nopClosure()` return type is *Closure<E>*, which is not a subtype of `Closure<? super E>` in the general case.

As the decompiled sources for `SwitchClosure` produced by all decompilers contain at least one error on this constructor, no solution is completable with the fragment store at the end of Algorithm 1. Therefore, the meta decompilation fails to produce recompilable sources.

5.5. Discussion

In this section we discuss how alternative design decisions might be applied for meta decompilation. In particular, we discuss the use of different oracles to choose among decompiled fragments and the order of decompilers in Algorithm 1.

A benefit of embedding a compiler in the meta decompiler is that it allows to use many different oracles to pick among

[13] https://github.com/leibnitz27/cfr/issues/50.

the decompiled (and optionally recompiled) fragments. In this paper, we use the syntactic correctness assessment done by the compiler. But it would be possible to use other oracles. For certain decompilation use cases, such as source recovery, tests covering the original bytecode could be available. In the case of reverse engineering, it is realistic to assume that one has access to a set of inputs with known outputs for the reversed program. They may be used as a test suite. The challenge with this approach is the granularity of the oracle provided by test cases. For meta decompilation to work, the granularity of the oracle needs to be finer or equal to the one of transplantation. In this work, we use *ecj* errors as we are able to map them to code fragments. This allows us to label a fragment as correct and incorrect.

Another oracle can be based on the bytecode distance between the decompiled-then-recompiled fragment and its original bytecode counterpart. This could be considered as a heuristic to minimize the likelihood of semantic differences between both fragments. In this work we measure bytecode distance with `JarDiff`, but `SootDiff` (Dann et al., 2019) could also be used, as its authors announce that it tolerates some control flow graph equivalent transformation.

Furthermore, depending on the metric that a decompiler user favors, the order of the decompilers used through meta decompilation may change. In this work we rank decompilers according to the number of classes for which they produce semantically equivalent modulo inputs sources. If a user favors the rate of deceptive decompilation to be as low as possible, CFR could be put first. Inverting the order of Procyon and CFR for Arlecchino, on the 157 classes presented in Section 4.5, yields only 38 decompiled classes that are semantically equivalent modulo inputs. But it produces only 4 deceptive decompilations.

Highlights about meta-decompilation: To summarize, we have devised and implemented a novel approach to merge results from different decompilers, called meta-decompilation. This tool handles 59 of the 157 cases (37.6%) previously not handled by any decompiler. Meta-decompilation is, to our knowledge, a radically new idea that has never been explored before. Our experiments demonstrate the feasibility and effectiveness of the idea.

6. Threats to validity

In this section, we report about internal, external and reliability threats against the validity of our results.

Internal validity. The internal threats are related to the metrics employed, especially those used to compare the syntactic distortion and semantic equivalence modulo inputs between the original and decompiled source code. Moreover, the coverage and quality of the test suite of the projects under study influences our observations about the semantic equivalence of the decompiled bytecode. To mitigate this threat, we select a set of mature open-source projects with good test suites as study subjects, and rely on state-of-the-art AST and bytecode differencing tools.

External validity. The external threats refer to what extent the results obtained with the studied decompilers can be generalized to other Java projects. To mitigate this threat, we reuse an existing dataset of Java programs which we believe is representative of the Java world. Moreover, we added a handmade project which is a collection of classes used in previous decompilers evaluations as a baseline for further comparisons.

Reliability validity. Our results are reproducible, the experimental pipeline presented in this study is publicly available online. We provide all necessary code to replicate our analysis, including AST metric calculations and statistical analysis via R notebooks.[14]

7. Related work

This paper is related to previous works on bytecode analysis, decompilation and program transformations. In this section, we present the related work on Java bytecode decompilers along these lines.

Kerbedroid (Jang et al., 2019) is the closest related work. The work focuses on decompilers for Android and starts from the same observation as ours: decompilers perform differently with varying applications due to the various strategies to handle information lost in compilation. Kerbedroid is a meta-decompiler that stitches together results from multiple decompilers. Our current work shares the same observation, while contributing two key novel points. We perform an in-depth assessment of the different strategies implemented in 8 decompilers, with respect to three quality attributes, including equivalence modulo-input to compare the behavior of decompiled bytecode. Arlecchino leverages the partial results from 8 decompilers instead of 3 for Kerbedroid, which increases the coverage of various corner cases in the bytecode.

The evaluation of decompilers is closely related to the assessment of compilers. In particular, Le et al. (2014) introduce the concept of semantic equivalence modulo inputs to validate compilers by analyzing the interplay between dynamic execution on a subset of inputs and statically compiling a program to work on all kind of inputs. Blackburn et al. (2006) propose a set of benchmarking selection and evaluation methodologies, and introduces the DaCapo benchmarks, a set of open source, client-side Java benchmarks. Naeem et al. (2007) propose a set of software quality metrics aimed at measuring the effectiveness of decompilers and obfuscators. In 2009, Hamilton and Danicic (2009) show that decompilation is possible for Java, though not perfect. In 2017, Kostelanský and Dedera (2017) perform a similar study on updated decompilers. In 2018, Gusarovs (2018) performed a study on five Java decompilers by analyzing their performance according to different handcrafted test cases. All those works demonstrate that Java bytecode decompilation is far from perfect.

Decompilers are disassemblers are closely related, and each pair of binary format, target language poses specific challenges (see Vinciguerra et al., 2003 for C++, Khadra et al., 2016 for ThumbISA, Grech et al., 2019 for Ethereum bytecode). With disassembling, types must be reconstructing (Troshina et al., 2010), as well as assignment chains (Van Emmerik, 2007). As we do in this paper, some researchers focus on reassembling disassembled binary code (Wang et al., 2015; Emamdoost et al., 2019; Flores-Montoya and Schulte, 2019).

A recent trend in decompilation is to use neural networks (Katz et al., 2018; Li et al., 2019; Fu et al., 2019). For example, Katz et al. (2018) present a technique for decompiling binary code snippets using a model based on Recurrent Neural Networks, which produces source code that is more similar to human-written code and therefore more easy for humans to understand. This a remarkable attempt at driving decompilation towards a specific goal. Lacomis et al. (2019) propose a probabilistic technique for variable name recovery. Schulte et al. (2018) use evolutionary search to improve and recombine a large population of candidate decompilations by applying source-to-source transformations gathered from a database of human-written sources. Miller and colleagues (Miller et al., 2019) model the uncertainty

[14] https://github.com/castor-software/decompilercmp/tree/master/notebooks.

due to the information loss during compilation using probabilities and propose a novel disassembly technique, which computes a probability for each address in the code space, indicating its likelihood of being a true positive instruction.

8. Conclusion

In this work, we presented a fully automated pipeline to assess Java bytecode decompilers with respect to their capacity to produce compilable, equivalent modulo-input, and readable code. We assessed eight decompilers with a set of 2041 classes from 14 open-source projects compiled with two different compilers. The results of our analysis show that bytecode decompilation is a non-trivial task that still requires human work. Indeed, even the highest ranking decompiler in this study produces syntactically correct output for 84% of classes of our dataset and semantically equivalent modulo inputs output for 78%. We extract 157 classes for which no decompiler produces semantically equivalent sources. These classes illustrate how generics and, in particular, generic with wildcards and type bounds are challenging for all decompilers. Yet the Java language with its diversity of compilers and versions makes room for many corner cases that require extensive testing and development effort from decompilers authors. Meanwhile, the diversity of implementation of these decompilers allows to merge their different results to bypass the shortcomings of single decompilers. We called this approach 'meta decompilation' and implemented it in a tool called Arlecchino. Our experimental results show that Arlecchino can produce semantic equivalence modulo inputs sources for 37.6% of classes for which, previously, no single decompiler could.

CRediT authorship contribution statement

Nicolas Harrand: Conceptualization, Methodology, Software, Investigation, Data curation, Writing - original draft. **César Soto-Valero:** Conceptualization, Methodology, Investigation, Data curation, Writing - original draft, Visualization. **Martin Monperrus:** Conceptualization, Methodology, Writing - review & editing, Supervision. **Benoit Baudry:** Conceptualization, Methodology, Writing - review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work has been partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation and by the TrustFull project funded by the Swedish Foundation for Strategic Research.

References

Amin, N., Tate, R., 2016. Java and scala's type systems are unsound: The existential crisis of null pointers. In: Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. Association for Computing Machinery, New York, NY, USA, pp. 838–848. http://dx.doi.org/10.1145/2983990.2984004.

Benfield, L., 2019. CFR. https://www.benf.org/other/cfr/. [Online; accessed 19-July-2019].

Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B., 2006. The dacapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications. Association for Computing Machinery, New York, NY, USA, pp. 169–190. http://dx.doi.org/10.1145/1167473.1167488.

Chen, Y., Jiang, Y., Ma, F., Liang, J., Wang, M., Zhou, C., Su, Z., Jiao, X., 2018. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. arXiv e-prints, arXiv:1807.00182, arXiv:1807.00182.

Dann, A., Hermann, B., Bodden, E., 2019. Sootdiff: Bytecode comparison across different java compilers. In: Proceedings of the 8th ACM SIGPLAN International Workshop on State of the Art in Program Analysis. Association for Computing Machinery, New York, NY, USA, pp. 14–19. http://dx.doi.org/10.1145/3315568.3329966.

Dupuy, E., 2019. Java Decompiler. https://http://java-decompiler.github.io/. [Online; accessed 19-July-2019].

Emamdoost, N., Sharma, V., Byun, T., McCamant, S., 2019. Binary mutation analysis of tests using reassembleable disassembly. http://dx.doi.org/10.14722/bar.2019.23058.

Falleri, J.R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M., 2014. Fine-grained and accurate source code differencing. In: 29th International Conference on Automated Software Engineering (ASE). ACM, New York, NY, USA, pp. 313–324. http://dx.doi.org/10.1145/2642937.2642982, URL: http://doi.acm.org/10.1145/2642937.2642982.

Flores-Montoya, A., Schulte, E.M., 2019. Datalog disassembly. CoRR arXiv:abs/1906.03969. URL: http://arxiv.org/abs/1906.03969, arXiv:1906.03969.

Foster, B., Somayaji, A., 2010. Object-level recombination of commodity applications. In: Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation. pp. 957–964.

Fu, C., Chen, H., Liu, H., Chen, X., Tian, Y., Koushanfar, F., Zhao, J., 2019. Coda: An end-to-end neural program decompiler. In: Advances in Neural Information Processing Systems. pp. 3703–3714.

Grech, N., Brent, L., Scholz, B., Smaragdakis, Y., 2019. Gigahorse: thorough, declarative decompilation of smart contracts. In: International Conference on Software Engineering. IEEE, pp. 1176–1186.

Gusarovs, K., 2018. An analysis on Java programming language decompiler capabilities. Appl. Comput. Syst. 23, 109–117.

Hamilton, J., Danicic, S., 2009. An evaluation of current Java bytecode decompilers. In: 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 129–136. http://dx.doi.org/10.1109/SCAM.2009.24.

Hoenicke, J., 2019. JODE. http://jode.sourceforge.net/. [Online; accessed 19-July-2019].

Jaffe, A., Lacomis, J., Schwartz, E.J., Goues, C.L., Vasilescu, B., 2018. Meaningful variable names for decompiled code: A machine translation approach. In: 26th Conference on Program Comprehension (ICPC). ACM, New York, NY, USA, pp. 20–30. http://dx.doi.org/10.1145/3196321.3196330, URL: http://doi.acm.org/10.1145/3196321.3196330.

Jang, H., Jin, B., Hyun, S., Kim, H., 2019. Kerberoid: A practical android app decompilation system with multiple decompilers. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. pp. 2557–2559.

Jerome Miecznikowski, Nomair A., Naeem, L.J.H., 2019. Dava. http://www.sable.mcgill.ca/dava/. [Online; accessed 19-July-2019].

JetBrains, 2019. Fernflower. https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine. [Online; accessed 19-July-2019].

Katz, D.S., Ruchti, J., Schulte, E., 2018. Using recurrent neural networks for decompilation. In: 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 346–356. http://dx.doi.org/10.1109/SANER.2018.8330222.

Khadra, M.A.B., Stoffel, D., Kunz, W., 2016. Speculative disassembly of binary code. In: International Conference on Compliers, Architectures, and Sythesis of Embedded Systems (CASES). pp. 1–10. http://dx.doi.org/10.1145/2968455.2968505.

Kostelanský, J., Dedera, L., 2017. An evaluation of output from current java bytecode decompilers: Is it android which is responsible for such quality boost?. In: Communication and Information Technologies (KIT). pp. 1–6. http://dx.doi.org/10.23919/KIT.2017.8109451.

Lacomis, J., Yin, P., Schwartz, E.J., Allamanis, M., Goues, C.L., Neubig, G., Vasilescu, B., 2019. Dire: A neural approach to decompiled identifier naming. arXiv:1909.09029.

Le, V., Afshari, M., Su, Z., 2014. Compiler validation via Equivalence Modulo inputs. In: 35th Conference on Programming Language Design and Implementation (PLDI). ACM, New York, NY, USA, pp. 216–226. http://dx.doi.org/10.1145/2594291.2594334, URL: http://doi.acm.org/10.1145/2594291.2594334.

Li, Z., Wu, Q., Qian, K., 2019. Adabot: Fault-Tolerant Java Decompiler. Technical Report 1908.06748, arXiv.

Lindholm, T., Yellin, F., Bracha, G., Buckley, A., 2014. The Java Virtual Machine Specification. Pearson Education.

Miecznikowski, J., Hendren, L., 2002. Decompiling Java bytecode: Problems, traps and pitfalls. In: Horspool, R.N. (Ed.), Compiler Construction. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 111–127.

Miller, K., Kwon, Y., Sun, Y., Zhang, Z., Zhang, X., Lin, Z., 2019. Probabilistic disassembly. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, pp. 1187–1198.

Naeem, N.A., Batchelder, M., Hendren, L., 2007. Metrics for measuring the effectiveness of decompilers and obfuscators. In: 15th IEEE International Conference on Program Comprehension (ICPC). pp. 253–258. http://dx.doi.org/10.1109/ICPC.2007.27.

Nolan, G., 2004. Decompiler Design. Apress, Berkeley, CA, pp. 121–157. http://dx.doi.org/10.1007/978-1-4302-0739-9_5.

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L., 2015. Spoon: A library for implementing analyses and transformations of Java source code. Softw. - Pract. Exp. 46, 1155–1179. http://dx.doi.org/10.1002/spe.2346, URL: https://hal.archives-ouvertes.fr/hal-01078532/document.

Persaud, B., Obada-Obieh, B., Mansourzadeh, N., Moni, A., Somayaji, A., 2016. Frankenssl: Recombining cryptographic libraries for software diversity. In: Proceedings of the 11th Annual Symposium On Information Assurance. pp. 19–25.

Ragkhitwetsagul, C., Krinke, J., 2017. Using compilation/decompilation to enhance clone detection. In: 11th International Workshop on Software Clones (IWSC). pp. 1–7. http://dx.doi.org/10.1109/IWSC.2017.7880502.

Robles, G., Gonzalez-Barahona, J.M., Herraiz, I., 2005. An empirical approach to software archaeology. In: 21st International Conference on Software Maintenance (ICSM). pp. 47–50.

Schulte, E., Ruchti, J., Noonan, M., Ciarletta, D., Loginov, A., 2018. Evolving exact decompilation. In: Shoshitaishvili, Y., Wang, R.F. (Eds.), Workshop on Binary Analysis Research. San Diego, CA, USA. URL: http://www.cs.unm.edu/eschulte/data/bed.pdf.

Selberg, E., Etzioni, O., 1997. The metacrawler architecture for resource aggregation on the web. IEEE Expert 12, 11–14. http://dx.doi.org/10.1109/64.577468.

skylot, 2019. JADX. https://github.com/skylot/jadx. [Online; accessed 19-July-2019].

Storyyeller, 2019. Krakatau. https://github.com/Storyyeller/Krakatau. [Online; accessed 19-July-2019].

Strobel, M., 2019. Procyon. https://bitbucket.org/mstrobel/procyon. [Online; accessed 19-July-2019].

Troshina, K., Derevenets, Y., Chernov, A., 2010. Reconstruction of composite types for decompilation. In: 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 179–188. http://dx.doi.org/10.1109/SCAM.2010.24.

Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V., 1999. Soot - a java bytecode optimization framework. In: Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research. IBM Press, p. 13.

Van Emmerik, M.J., 2007. Static Single Assignment for Decompilation. University of Queensland.

Ďurfina, L., Křoustek, J., Zemek, P., 2013. PsybOt malware: A step-by-step decompilation case study. In: 20th Working Conference on Reverse Engineering (WCRE). pp. 449–456. http://dx.doi.org/10.1109/WCRE.2013.6671321.

Vinciguerra, L., Wills, L., Kejriwal, N., Martino, P., Vinciguerra, R., 2003. An experimentation framework for evaluating disassembly and decompilation tools for C++ and Java. In: 10th Working Conference on Reverse Engineering (WCRE). IEEE Computer Society, Washington, DC, USA, p. 14, URL: http://dl.acm.org/citation.cfm?id=950792.951361.

Wang, S., Wang, P., Wu, D., 2015. Reassembleable disassembling. In: 24th USENIX Security Symposium (USENIX Security 15). USENIX Association, Washington, D.C., pp. 627–642, URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-shuai.

Yakdan, K., Dechand, S., Gerhards-Padilla, E., Smith, M., 2016. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In: IEEE Symposium on Security and Privacy (SP). pp. 158–177. http://dx.doi.org/10.1109/SP.2016.18.

Yang, Y., Zhou, Y., Sun, H., Su, Z., Zuo, Z., Xu, L., Xu, B., 2019. Hunting for bugs in code coverage tools via randomized differential testing. In: 41st International Conference on Software Engineering (ICSE). ACM.