The Strengths and Behavioral Quirks of Java Bytecode Decompilers

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

KTH Royal Institute of Technology, Stockholm, Sweden

Email: {harrand, cesarsv, baudry}@kth.se, martin.monperrus@esc.kth.se

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, the decompilation process, which aims at producing source code from bytecode, must establish some strategies to reconstruct the information that has been lost. Modern Java decompilers tend to 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.

We study the effectiveness of eight Java decompilers with respect to three quality indicators: syntactic correctness, syntactic distortion and semantic equivalence modulo inputs. This study relies on a benchmark set of 14 real-world open-source software projects to be decompiled (2041 classes in total).

Our results show that no single modern decompiler is able to correctly handle the variety of bytecode structures coming from real-world programs. Even the highest ranking decompiler in this study produces syntactically correct output for 84% of classes of our dataset and semantically equivalent code output for 78% of classes.

Index Terms—Java bytecode, decompilation, reverse engineering, source code analysis

I. 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 [1]. For example the structure of loops is altered and local variable names may be modified [2].

Decompilation is the inverse process, it consists in transforming the bytecode instructions into source code [3]. 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 and grammatically correct and can be recomplied. Some other applications of decompilation with slightly different criteria include clone detection [4], malware analysis [5], [6] and software archaeology [7].

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 having compared Java decompilers [8], [9] found that this ideal Java decompiler does not exist, because of the irreversible data loss that happens during compilation. Yet, the experimental scale of this previous work is rather small to fully understand the state of decompilation for Java. There is a fundamental reason for this: this previous work relies on manual analysis to assess the semantic correctness of the decompiled code.

In this paper, we solve this problem by proposing a fully automated approach to study Java decompilation, based on equivalence modulo inputs (EMI) [10]. The idea is to automatically check that decompiled code behaves the same as the original code, using inputs provided by existing application test suites. In short, the decompiled code of any arbitrary class $x$ should pass all the tests that exercise $x$. To our knowledge, this is the first usage of EMI in the context of decompilation. With that instrument, 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 modulo input with the original source (the decompiled code passes all tests); the syntactic similarity to the original source (the decompiled source looks like the original). To our knowledge, this is the first deep study of those three aspects together.

Our study is based on 14 open-source projects totaling 2041 Java classes. We evaluate eight recent and notable decompilers on code produced by two different compilers. This study is at least one order of magnitude larger than the related work [8], [9]. Our results are important for different people: 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 shows that the problem is not solved, and we isolate a subset of 157 Java classes that no state-of-the-art decompiler can correctly handle: 3) for authors of decompilers, our experiments have identified bugs in their decompilers (2 have already been fixed, and counting) and our methodology of semantic equivalence modulo inputs can be
class Dava
final int and commons-codec DecoderException
an empirical comparison of eight Java decompilers based throw new
a fully automated pipeline to assess the syntactic and throws Fernflower
i;
27 (i == -1) {
25019 test \( \text{"Invalid URL }\)
25: INVOKEVIRTUAL #32
20: DUP
16: DUP
20: DUP
16: //"Invalid URL encoding: not a valid digit (radix 16):"
21: LDC #27
18: //StringBuilder."\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\text{ at java/lang/StringBuilder}"
29: //StringBuilder.append:(L\text{java/lang/String};V}
22: //StringBuilder.append:(L\text{java/lang/String};V}
24: //DecoderException."\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\:\\text{ at java/lang/String}"
30: //DecompilerException."
33: //DecompilerException."
36: ATHROW
38: IRETURN
36:
38:
36:
38:
36:
38:
36:
38:
Listing 1. Source code of org.apache.commons.codec.net.Utils.

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

II. MOTIVATING EXAMPLE

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. 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. 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 than 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.

1https://github.com/castor-software/decompilercmp
2There are various Java compilers available, notably Oracle javac and Eclipse ecj, which can produce different bytecode for the same Java input.

On the contrary, the Dava decompiler does not reverse this transformation. As we can 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 do 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 16) + 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
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.

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.

III. 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.

A. 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 [11] between the original and the decompiled results. This metric, called syntactic distortion, allows to measure the differences that goes beyond variable names. The description of the metric is as follows:

Definition 2. Syntactic distortion. 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.3

Inspired by the work of [10] and [12], we check if the decompiled and recompiled program is semantically equivalent modulo inputs. This means that for a given set of inputs, the two program 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 [8], this is particularly problematic as it can mislead decompiler users in their attempt to understand the original behavior of the program. We refer to theses cases as deceptive decompilation results.

Definition 4. Deceptive decompilation: Decompiler output that is syntactically correct but not semantically equivalent to the original input.

B. 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? In this research question, we investigate on the semantic differences between the original source code and the outputs of the decompilers.

RQ3: To what extent do decompilers produce deceptive decompilation results? Le and colleagues [10] 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.

RQ4: What is the syntactic distortion of decomplied 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.

3https://github.com/scala/jardiff
**RQ5: To what extent the behavioral diversity of decompilers can be leveraged to improve the decompilation of Java bytecode?** As we observed during their comparison, each decompiler have their pros and cons. We evaluate if this diversity of features can be leveraged to boost the overall decompilation results.

**C. Study Protocol**

Figure 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. 
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 difference between the original and the recomplied bytecode.
4) Run the test cases on the recomplied 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 as not available. As decompilation can sometime produce a program that does not stop, we set a 20 minutes timeout on the test execution (the original test suites run under a minute on the hardware used for this experiment, a Core i5-6600K with 16Go of RAM).

**D. Study Subjects**

**Decompilers.** Table I 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 [8]. The column VERSION shows the version used (some decompilers do not follow any versioning scheme). We choose the latest release if one exist, if note the last commit available the 09-05-2019. The column #COMMENTS 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 line of code in all Java files (and Python files for **Krákatau**) of the decompiler, including sources, test sources and resources counted with cloc.

**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 [21]. To these 13 projects we added a fourteenth one named **DcTest** made out of examples collected from previous decompiler evaluations [8], [9]. Table II 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, we employed the two most used Java compilers: **javac** and **ecj** (we use versions 1.8.0_17 and 13.13.100, respectively). We compiled all the 14 projects with both compilers (except commons-lang that we failed to build it with **ecj**) This represents 1887 class files for each compiler that we use to evaluate syntactic correctness.

---

4 Coverage was assessed using yajta https://github.com/castor-software/yajta
5 http://cloc.sourceforge.net/
6 http://www.program-transformation.org/Transform/JavaDecompilerTests
7 https://www.benf.org/other/cfr/eclipse-differences.html

---

![Diagram of decompiler assessment pipeline](image-url)
of decompiler outputs in RQ1 and syntactic distortion in RQ4. We select only those that contain code executed by tests (2397 grouping files generated by the two compilers) to evaluate semantic correctness in RQ2 and RQ3.

IV. EXPERIMENTAL RESULTS

A. 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.

Figure 2 shows the ratio of compiled 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 the decompiler on the right. For example, Procyon, shown in the last row, is able to produce a 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 (82.2%) of the 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.

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). 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, as in the case of our experiments.

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.

Intuitively, it seems that the compiler has an impact on decompilation effectiveness. To verify this, we use a $\chi^2$ test on the ratio of classfile decompiled into syntactically correct source code depending on the used compiler, javac versus ecj. The compiler variable has an impact for three decompilers and no impact for the remaining five at 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 will be discussed in more details in Section IV-C.

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 do require some additional manual effort. Only few cases make all decompiler fail, which suggest that using several decompilers in conjunction could help to achieve better results.

B. 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 that are covered by at least one test case. When decompilers produce sources that compile, we investigate the semantic equivalence of the decompiled source and their original. To do so, 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 constant pool, as explained in Section III-A), (ii) semantically equivalent modulo inputs: the output is recompilable and passes the original project 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 interesting example of semantic equivalence of decompiled code. Listing 5 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 (IFNE becomes IFEQ), 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 is still equivalent code.

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

Listing 5. Excerpt of org/apache/commons/codec/language/bm/Lang.class bytecode compiled with javac and decompiled with CFR. Lines in red are in the original byte code, while lines in green are from the recompiled sources.

Answer to RQ2: The number of classes for which the decompiler produces EMI semantically equivalent varies a lot from one decompiler to another. The source code generated by the decompilers is usually not strictly identical to the original, still many of the decompiled classes are semantically equivalent modulo inputs. 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 in the decompiled code.

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

As explained by Hamilton and colleagues [8], 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. 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 as decompilation bugs.
Figure 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 it 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 Figure 4 represents 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.

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 correspond to less 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 of them to the authors of decompilers. 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.

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 6. 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 7 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 compiled, with an excerpt shown in Listing 6 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 7, 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 (LOAD, 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.

2) Case study: Down cast error: Listing 8 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
Our empirical results indicate that no changes the behavior of `StringBuffer applyRules` and `StringBuffer decompiler` have a median of and `StringBuffer into Appendable buf` `StringBuffer RQ4: (ASTs difference)`

```java
protected StringBuffer applyRules(final Calendar calendar, final StringBuffer buf) {
    return (StringBuffer) applyRules(calendar, (Appendable) buf);
}

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

Listing 8. Excerpt of differences in FastDatePrinter original (in red) and decompiled with Procyon sources (in green).

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. 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 other have been acknowledged.

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

The quality of decompilation depends not only on its syntactic compilability and semantic equivalence but also on how well a human can understand the behavior of the decompiler 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 program and introduces a bug in it.

**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. Our results suggest that syntactic distortion can be used by decompiler developers to improve the alignment between the decompiled sources and the original. Also, decompiler users can use this analysis when deciding which decompiler to employ.
RQ5: (Multi-decompiler evaluation) Krakatau because they Dava In this section, we investigate what and classes are handled correctly by all of

```java
public class Foo {
    public int foo(int i, int j) {
        try {
            while (true) {
                while (i < j) i = j++ / i;
            }
            catch (RuntimeException re) {
                i = 10;
            }
            break;
        }
        return j;
    }
}
```

Listing 9. Excerpt of differences in Foo original and decompiled with Fernflower sources.

These 6 decompilers. Furthermore, 157/2397 classes are not correctly handled by any of the considered decompilers.

To assess the benefit of using multiple decompilers instead of one, we have implemented a naive Multi-DC that uses each decompiler one by one until it finds a syntactically correct decompilation result. The order of decompiler tried follows a ranking by decreasing success rate according to the six most successful decompilers in terms of semantic equivalence modulo inputs rates. In this manner, we can compare the effectiveness of this naive meta-decompiler with respect to the other decompilers taken in isolation.

Table III summarizes the quantitative results obtained from the previous research questions, and adds the effectiveness of the Multi-DC as the last row. Each line corresponds to a decompiler. Column #Recompilable shows the number of cases (and ratio) for which the decompiler produced a recompilable output; column #PassTest shows the number of cases where the decompiled code passes those tests; column #Deceptive indicate the number of cases that were recompilable but did not pass the test suite (i.e. a compilation bug); column #ASTDist indicate the average syntactic distortion among successfully decompiled cases.

Table III: Summary results of the studied decompilers plus Multi-DC

| Decomplier | #Recompilable | #PassTest | #Deceptive | #ASTDist |
|------------|---------------|-----------|------------|----------|
| CFR        | 2057 (0.59)   | 1714 (0.71) | 22         | 0.05     |
| Dava       | 1747 (0.44)   | 762 (0.32)  | 36         | 0.17     |
| Fernflower | 2663 (0.68)   | 1435 (0.60)| 21         | 0.08     |
| JADX       | 2736 (0.70)   | 1408 (0.59)| 78         | 0.07     |
| JCore      | 2726 (0.69)   | 1375 (0.57)| 82         | 0.06     |
| Jode       | 2569 (0.65)   | 1161 (0.48)| 142        | 0.09     |
| Krakatau   | 1746 (0.44)   | 724 (0.30)  | 97         | 0.20     |
| Procyon    | 3281 (0.84)   | 1869 (0.78)| 33         | 0.08     |
| Multi-DC   | 3734 (0.95)   | 2174 (0.91)| 45         | 0.08     |

Overall, the naive Multi-DC implementation performs the best in terms of both syntactically correct and semantically equivalent modulo inputs criteria. However, it is not the best in #Deceptive, because it accumulates all bugs from Procyon and bugs from other decompilers that affect cases not handled by Procyon. Note that an user ready to give up performance could reorganize the order of decompilers tried by the Multi-DC in order to optimize either #Deceptive or #ASTDist (with no impact on the number of syntactically correct cases). This shows that a decompiler user who would use the Multi-DC approach would obtain syntactically correct sources more frequently by 11 points and semantically equivalent modulo inputs sources by 13 points.

As observed, the decompilation of Java is a non trivial task with no clear systematic solution. In order to produce useful results, decompiler developers make various assumptions about the source code that produced the bytecode, or about the compiler. For example Dava does not assume that the bytecode was produced from Java sources [22]. CFR does not trust information contained in the Local Variable Type Table, as an obfuscation tool could change it without altering the behavior of the program. All these assumptions, and various

9The website http://www.javadecompilers.com indeed proposes to leverage this multiplicity of decompilers.

10https://www.benf.org/other/cfr/faq.html
strategies implemented by the decompilers lead to a situation where the collection of implementations successfully covers significantly more input cases that any individual implementation.

**Answer to RQ5:** By leveraging the diversity of features present in existing decompilers, the naïve Multi-DC decompiler outperforms the other decompilers in terms of syntactic correctness (by 11 percentage points compared to the best) and semantic equivalence modulo inputs (by 13 percentage points). This quantitatively illustrates the benefit for decompiler users to try different decompiler instead of a single one. It also suggests research opportunities to approach the decompilation problem with a set of various decompilation strategies instead of a single one.

V. THREATS TO VALIDITY

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

a) 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.

b) External validity: The external threats refer to what extend 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.

c) 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.  

VI. 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.

The evaluation of decompilers is closely related to the assessment of compilers. In particular, Le et al. [10] 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. Naeem et al. [23] propose a set of software quality metrics aimed at measuring

| V. T | ALENT TO VALIDITY |
|-----|-------------------|
| a) 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. |
| b) External validity: | The external threats refer to what extend 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. |
| c) 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. |

VII. CONCLUSION

Java bytecode decompilation is used for multiple purposes, ranging from reverse engineering to source recovery and understanding. In this work we proposed a fully automated pipeline to evaluate the Java bytecode decompilers’ capacity to produce compilable, semantically equivalent and readable code. We proposed to use the concept of semantic equivalence modulo inputs to compare decompiled sources to their original counterpart. We applied this approach on 8 available decompilers through a set of 2041 classes from 14 open-source projects compiled with 2 different decompilers. 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%. Meanwhile the diversity of implementation of these decompiler allow an user to combine several of them with significantly better results than by using a single one. In future work, we will explore the possibility to exploit this diversity of decompiler implementation automatically by merging the results of different decompilers via source code analysis and manipulations.

---

https://github.com/castor-software/decompilercmp/tree/master/notebooks
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 financed by the Swedish Foundation for Strategic Research.

REFERENCES

[1] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, The Java Virtual Machine Specification. Pearson Education, 2014.
[2] A. Jaffe, J. Lacomis, E. J. Schwartz, C. L. Goues, and B. Vasilisecu, “Meaningful Variable Names for Decompiled Code: A Machine Translation Approach,” in 26th Conference on Program Comprehension (ICPC), (New York, NY, USA), pp. 20–30, ACM, 2018.
[3] G. Nolan, Decompiler Design, pp. 121–157. Berkeley, CA: Apress, 2004.
[4] C. Ragkhiwetsagul and J. Krinke, “Using Compilation/Decompilation to Enhance Clone Detection,” in 11th International Workshop on Software Clones (IWSC), pp. 1–7, Feb 2017.
[5] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, “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, May 2016.
[6] L. Durlina, J. Klouftek, and P. Zemek, “PsyBoT Malware: A Step-By-Step Decomposition Case Study,” in 20th Working Conference on Reverse Engineering (WCRE), pp. 449–456, Oct 2013.
[7] G. Robles, J. M. Gonzalez-Barahona, and I. Herranz, “An Empirical Approach to Software Archaeology,” in 21st International Conference on Software Maintenance (ICSM), pp. 47–50, 2005.
[8] J. Hamilton and S. Dunicic, “An Evaluation of Current Java Bytecode Decompilers,” in 9th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 129–136, Sep. 2009.
[9] J. Kostelanský and L. Dedera, “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, Oct 2017.
[10] V. Le, M. Afshar, and Z. Su, “Compiler Validation via Equivalence Modulo Inputs,” in 25th Conference on Programming Language Design and Implementation (PLDI), (New York, NY, USA), pp. 216–226, ACM, 2014.
[11] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and Accurate Source Code Differencing,” in 29th International Conference on Automated Software Engineering (ASE), (New York, NY, USA), pp. 313–324, ACM, 2014.
[12] Y. Yang, Y. Zhou, H. Sun, Z. Su, Z. Zuo, L. Xu, and B. Xu, “Hunting for Bugs in Code Coverage Tools via Randomized Differential Testing,” in 41st International Conference on Software Engineering (ICSE), (New York, NY, USA), pp. 129–140, ACM, 2019.
[13] L. Benfield, “CFR.” https://www.benf.org/other/cfr/, 2019. [Online; accessed 19-July-2019].
[14] L. J. H. Jerome Miecznikowski, Nomair A. Naeem, “Dava.” http://www.sable.mcgill.ca/dava/, 2019. [Online; accessed 19-July-2019].
[15] “Fernflower.” https://github.com/NetBrains/intellij-community/tree/master/plugins/java-decompiler/engine, 2019. [Online; accessed 19-July-2019].
[16] skylot, “JADX.” https://github.com/skylot/jadx, 2019. [Online; accessed 19-July-2019].
[17] E. Dupuy, “Java Decompiler.” https://http://java-decompiler.github.io/, 2019. [Online; accessed 19-July-2019].
[18] J. Hoenicke, “JODE.” http://jode.sourceforge.net/, 2019. [Online; accessed 19-July-2019].
[19] Storyyeller, “Krakatau.” https://github.com/storyyeller/Krakatau, 2019. [Online; accessed 19-July-2019].
[20] M. Strobel, “Procyon.” https://bitbucket.org/mstrobel/procyon, 2019. [Online; accessed 19-July-2019].
[21] R. Pawlik, M. Monperrus, N. Petitprez, C. Nogueira, and L. Seinturier, “Spoon: A Library for Implementing Analyses and Transformations of Java Source Code,” Software: Practice and Experience, vol. 46, pp. 1155–1179, 2015.
[22] J. Miecznikowski and L. Hendren, “Decompiling Java Bytecode: Problems, Traps and Pitfalls,” in Compiler Construction (R. N. Horspool, ed.), (Berlin, Heidelberg), pp. 111–127. Springer Berlin Heidelberg, 2002.
[23] N. A. Naeem, M. Batchelder, and L. Hendren, “Metrics for Measuring the Effectiveness of Decompilers and Obfuscators,” in 15th IEEE International Conference on Program Comprehension (ICPC), pp. 253–258, June 2007.
[24] K. Gusarovs, “An Analysis on Java Programming Language Decompiler Capabilities,” Applied Computer Systems, vol. 23, no. 2, pp. 109–117, 2018.
[25] L. Vinciguerra, L. Wills, N. Kejriwal, P. Martino, and R. Vinciguerra, “An Experimentation Framework for Evaluating Disassembly and Decompilation Tools for C++ and Java,” in 10th Working Conference on Reverse Engineering (WCRE), (Washington, DC, USA), pp. 14–, IEEE Computer Society, 2003.
[26] M. A. B. Khuda, D. Stoffel, and W. Kunz, “Speculative disassembly of binary code,” in International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES), pp. 1–10, Oct 2016.
[27] O. Katz, Y. Olshaker, Y. Goldberg, and E. Yahav, “Towards Neural Decompilation,” arXiv e-prints, p. arXiv:1905.08325, May 2019.
[28] K. Troshina, Y. Derevenets, and A. Chernov, “Reconstruction of Composite Types for Decompilation,” in 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 179–188, Sep. 2010.
[29] M. J. Van Emmerik, Static Single Assignment for Decompilation, University of Queensland, 2007.
[30] D. S. Katz, J. Ruchti, and E. Schulte, “Using Recurrent Neural Networks for Decompilation,” in 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 346–356, March 2018.
[31] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Loginov, “Evolving exact decompilation,” in Workshop on Binary Analysis Research (Y. Shoshitaishvili and R. F. Wang, eds.), (San Diego, CA, USA), Feb. 18–21 2018.
[32] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, Z. Su, and X. Jiao, “EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers,” arXiv e-prints, p. arXiv:1807.00182, Jun 2018.