SootDiff

Bytecode Comparison Across Different Java Compilers

Andreas Dann
andreas.dann@upb.de
Heinz Nixdorf Institut
Paderborn University
Paderborn, Germany

Ben Hermann
ben.hermann@upb.de
Heinz Nixdorf Institut
Paderborn University
Paderborn, Germany

Eric Bodden
eric.bodden@upb.de
Heinz Nixdorf Institut
Paderborn University
Fraunhofer IEM
Paderborn, Germany

Abstract

Different Java compilers and compiler versions, e.g., `javac` or `ecj`, produce different bytecode from the same source code. This makes it hard to trace if the bytecode of an open-source library really matches the provided source code. Moreover, it prevents one from detecting which open-source libraries have been re-compiled and rebundled into a single jar, which is a common way to distribute an application. Such rebundling is problematic because it prevents one to check if the jar file contains open-source libraries with known vulnerabilities. To cope with these problems, we propose the tool SootDiff that uses Soot's intermediate representation Jimple, in combination with code clone detection techniques, to reduce dissimilarities introduced by different compilers, and to identify clones. Our results show that SootDiff successfully identifies clones in 102 of 144 cases, whereas bytecode comparison succeeds in 58 cases only.

CCS Concepts • **Security and privacy** → *Software and application security*; • **Software and its engineering**;

Keywords Intermediate Representation, Code Clone Detection, Static Analysis

ACM Reference Format:
Andreas Dann, Ben Hermann, and Eric Bodden. 2019. SootDiff: Bytecode Comparison Across Different Java Compilers. In *Proceedings of the 8th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP '19), June 22, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3315568.3329966

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SOAP '19, June 22, 2019, Phoenix, AZ, USA

© 2019 Association for Computing Machinery.

ACM ISBN 978-1-4503-6720-2/19/06…$15.00
https://doi.org/10.1145/3315568.3329966

1 Introduction

Java projects often include a considerable amount of open-source libraries from public repositories, e.g., Maven Central which contains more than 3.5 million[1] artifacts. Developers often include existing libraries by copying and pasting library source code or bytecode into their project [2, 3, 6, 11]. This raises license and security problems, as copyright terms may be violated or libraries with known vulnerabilities may become assembled into a project. A simple comparison of the project's bytecode with the library's code to identify which parts originate from the library is difficult, as different Java compilers or configured target versions produce slightly different bytecode. For instance, the bytecode generated by `javac` for version 1.5 differs from the bytecode generated for version 1.8. Thus, a simple comparison based on the bytecode will fail. Even if the source code of a library is openly available the exact compiler configuration must be known and used to recompile the source code. This information, however, is rarely available.

Developers have to trust that the vendors of an artifact compiled the source code without any modifications - as they do not know the exact compiler configuration to (re-)produce the same bytecode. There exists no option to trace and validate that the bytecode hosted in public repositories corresponds to the published source code.

Even under the assumption that the libraries are benign, developers may accidentally include known vulnerable libraries into their application [11]. To check if the project is susceptible to published vulnerabilities it is necessary to check if the project actually includes vulnerable library code.

Detecting library code inside a project becomes even more difficult by the fact that developers commonly bundle their software together, either by bundling all required libraries into a single jar (rebundling) or by embedding libraries recompiled source code or bytecode of libraries directly into their project using different package names [3, 5, 11]. However, not only local libraries, residing on the developer's machine, but also libraries hosted in repositories, e.g., Maven Central, are subject to repackaging and rebundling, and thus may contain foreign code.

[1] by March 2019

To cope with repackaging, rebundling, different compilers, and unavailable library source code, we propose to use the intermediate representation Jimple [18] of the static analysis framework Soot [12] to identify the libraries and versions a project includes, instead of using the source code or bytecode directly. To do so SootDiff[2] integrates and enriches Soot with an implementation of Myers' Diff algorithm tailored for Jimple and additional optimization steps to reduce dissimilarities between different Java compilers. In contrast to Java bytecode which uses more than 200 different instructions, Jimple uses only 15 distinct instructions. As we show, for this reason many different but functionally equivalent code constructs are likely to coincide on the Jimple level. In contrast to techniques that identify libraries using statistical information that is orthogonal to the choice of compiler, e.g., hashes of method signatures, SootDiff considers low-level details such as method bodies, which is required to identify the library's exact version - which in turn is required to identify the published vulnerabilities affecting the particular version.

To validate the feasibility of our approach, we compare how SootDiff performs compared to an approach solely based on bytecode comparison. Therefore, we check for 144 different classes in bytecode format, which we generated with different compilers and target version, originating from 17 Java source files, how often SootDiff and a bytecode comparison correctly identifies that two bytecode classes originate from the same source. In total, SootDiff successfully identifies for 102 test cases that they originate from the same source code, whereas a comparison based on bytecode succeeds only for 58 test cases. Thus, our results show that SootDiff and Jimple reduce dissimilarities introduced by different Java compiler and Java target versions. Consequently, Jimple eases the detection of similarities and differences between different bytecode artifacts.

In summary, this paper makes the following contributions:

- We present an approach to compare the bytecode produced by different Java compilers based on the intermediate representation Jimple in Section 3.
- We compare the performance of our approach against a naïve bytecode comparison in Section 4.

2 Background

Jimple The static analysis framework Soot [12] uses an intermediate representation named Jimple [18] to produce a format for representing Java source and bytecode for static code analysis. Jimple serves as an abstraction layer by drastically reducing the number of instructions needed to represent bytecode. Jimple uses only 15 different instructions whereas Java bytecode is composed of over 200 instructions. The reduced number of instructions mitigates the dissimilarities introduced by different Java versions and compilers, as the 200 bytecode instructions are mapped to 15 Jimple instructions. Thus, Jimple helps to compare the bytecode generated by different compilers and eases the detection of code clones.

Figures 1a-2 show an example of Java source code, its bytecode, and Jimple code. Similar to bytecode, Jimple transforms Java's control structures, e.g., if-else, loops, to goto instructions and program labels. For instance, the if-condition in the source code, shown in Figure 1a in Line 4, is translated to the branch instruction `ifeq` in Figure 1b in Line 7. The majority of lines in the bytecode, namely the Lines 9-17, constructs the string `"Debug:"+s` using `java.lang.StringBuilder` referring to methods and strings (#9-#14) stored in the class' constant pool.

In contrast to the bytecode, the Jimple code declares all variables explicitly, shown in Figure 2. The instructions to constructs the string and invoke the `StringBuilder` are fully resolved in Lines 17-20. Moreover, the condition on the variable `debug` is stated explicitly in Line 14.

```
1 class Point2d {
2 private boolean debug;
3 public void dprint(String s){
4  if (debug)
5    System.out.println("Debug:"+s);
6  }
7 }
```

(a) Java Source Code

```
1 class Point2d {
2 private boolean debug;
3 public void dprint(java.lang.String);
4 Code:
5  0: aload_0
6  1: getfield #4
7  4: ifeq 32
8  7: getstatic #8
9 10: new #9
10 13: dup
11 14: invokespecial #10
12 17: ldc #11
13 19: invokevirtual #12
14 22: aload_1
15 23: invokevirtual #12
16 26: invokevirtual #13
17 29: invokevirtual #14
18 32: return }
```

(b) Bytecode generated with `javac` for target version 1.8 (human-readable format)

Figure 1. Java Source Code vs. Bytecode

Java Compilers Several compilers to produce JVM-compatible bytecode from Java source code exist. The most common are Oracle's `javac`, the Eclipse Compiler for Java (ECJ)[3], IBM Jikes[4], or the GNU Compiler for the Java programming

[2] https://github.com/secure-software-engineering/sootdiff

[3] https://www.eclipse.org/jdt/core/

[4] https://sourceforge.net/projects/jikes/

```
1 class Point2d extends java.lang.Object{
2 private boolean debug;
3 public void dprint(java.lang.String)
4 { Point2d r0;
5  java.lang.String r1, $r6;
6  boolean $z0;
7  java.lang.StringBuilder $r2, $r4, $r5;
8  java.io.PrintStream $r3;
9
10  r0 := @this: Point2d;
11  r1 := @parameter0: java.lang.String;
12  $z0 = r0.<Point2d: boolean debug>;
13
14  if $z0 == 0 goto label1;
15  $r3 = <java.lang.System: java.io.PrintStream out>;
16  $r2 = new java.lang.StringBuilder;
17  specialinvoke $r2.<java.lang.StringBuilder: void <init>()>();
18  $r4 = virtualinvoke $r2.<java.lang.StringBuilder:
        java.lang.StringBuilder
        append(java.lang.String)>("Debug:");
19  $r5 = virtualinvoke $r4.<java.lang.StringBuilder:
        java.lang.StringBuilder append(java.lang.String)>(r1);
20  $r6 = virtualinvoke $r5.<java.lang.StringBuilder:
        java.lang.String toString()>();
21  virtualinvoke $r3.<java.io.PrintStream: void
        println(java.lang.String)>($r6);
22 label1:
23  return;} }
```

Figure 2. Jimple parsed from bytecode generated with `javac` target version 1.8

,

language (GCJ)[5]. Since the Java language specification [15] does not state if and how a compiler should optimize certain Java language features during compile-time, optimizations are a design decision of the particular compiler vendor. Thus, the bytecode created by different compilers differ.

A small example for the different optimizations is given in the figures below. Figure 3a and Figure 3b show the bytecode and Jimple code generated by the ECJ for the class `Point2d`[6]. In contrast to Oracle's compiler, shown in Figure 2, ECJ produces slightly different bytecode as the constant string `"Debug:"` is inlined directly into the `StringBuilder` constructor in Line 18, and thus requires one variable and invoke instruction less. Consequently, the bytecode generated by Oracle's Java compiler and the bytecode generated by ECJ for the source class `Point2d` differ.

Moreover, Java compilers support the option to generate bytecode for different Java language versions, as the different versions of the Java language version support different bytecode instructions. Thus, a compiler may generate different bytecode for different target language versions, e.g., Java 1.8 or Java 1.5. Consequently, simply comparing the bytecode is insufficient as different Java compiler produce slightly different bytecode, even one compiler may produce different bytecode for different Java target versions. On top

```
1 class Point2d {
2   private boolean debug;
3   descriptor: Z
4
5 public void
      dprint(String);
6  descriptor:
        (Ljava/lang/String;)V
7 Code:
8  0: aload_0
9  1: getfield #20
10  4: ifeq 29
11  7: getstatic #35
12 10: new #41
13 13: dup
14 14: ldc #43
15 16: invokespecial #45
16 19: aload_1
17 20: invokevirtual #47
18 23: invokevirtual #51
19 26: invokevirtual #55
20 29: return }
```

(a) (Decompiled) Bytecode `ecj` target version 1.8

```
1 class Point2d extends
        java.lang.Object{
2 private boolean debug;
3
4 public void dprint(String){
5  Point2d r0;
6  java.lang.String r1, $r6;
7  boolean $z0;
8  java.lang.StringBuilder $r2, $r4;
9  java.io.PrintStream $r3;
10
11  r0 := @this: Point2d;
12  r1 := @parameter0:
        java.lang.String;
13  $z0 = r0.<Point2d: boolean debug>;
14
15  if $z0 == 0 goto label1;
16  $r3 = <java.lang.System:
        java.io.PrintStream out>;
17  $r2 = new java.lang.StringBuilder;
18  specialinvoke
        $r2.<java.lang.StringBuilder:
        void <init>()>("Debug:");
19  $r4 = virtualinvoke
        $r2.<java.lang.StringBuilder:
        java.lang.StringBuilder
        append(java.lang.String)>(r1);
20  $r5 = virtualinvoke
        $r5.<java.lang.StringBuilder:
        java.lang.String
        toString()>();
21  virtualinvoke
        $r3.<java.io.PrintStream:
        void
        println(java.lang.String)>($r5);
22 label1:
23  return;} }
```

(b) Jimple parsed from bytecode generated with `ecj` target version 1.8

Figure 3. Bytecode vs. Jimple

of that, the comparison of Figure 2 and Figure 3b shows that even the (unoptimized) Jimple code parsed from the bytecode generated by `javac` and `ecj` differs.

3 Design: SootDiff

The idea of SootDiff is to use Jimple to compare two classes. This allows us to match classes with equivalent behavior even when they are produced with different Java compilers or different configurations. Additionally, this reduces the dissimilarities introduced by rebundling.

Figure 4 shows SootDiff's approach in a nutshell. SootDiff's result is a set of diff chunks for the class files using clone detection algorithms. Our approach is open to use with different traditional and established code clone detection tools. We currently use Myers' algorithm [14] to compare methods' bodies using `java-diff-utils`[7]. This greedy algorithm, which is for instance used in GNU DiffUtils, calculates the differences between two strings and a sequence of edits to

[5] https://web.archive.org/web/20070509055923/http://gcc.gnu.org/java/

[6] Sample Java Class University Illinois https://www.cs.uic.edu/~sloan/CLASSES/java/

[7] https://github.com/java-diff-utils/java-diff-utils

convert one string to the other. The algorithm recursively finds for two sequences the longest common subsequence with the smallest edit sequence. Once this is done, the algorithm compares recursively two subsequences preceding and following the matched sequence until there are no more sequences left for comparison. We only use Myers' algorithm for comparing method bodies, whereas we compare the signatures of classes, methods, and fields semantically using `apache:commons:DiffBuilder`[8] directly in Soot, and thus independent of their ordering in the code, e.g., the sequence in which the class declares its members or the sequence of method parameters.

In a first step, we use the Soot framework [12] to produce Jimple from the bytecode classes to compare, which may differ if they have been generated by different Java compiler. Therefore, we pass the bytecode files to Soot to transform them to Jimple. In a second step, SootDiff optimizes the Jimple representation using Soot's internal Jimple optimizer [18]. In this optimization step, SootDiff applies constant-propagation, dead-code-elimination, and unconditional-branch folder to the Jimple code [12, 18]. Thereby, these optimization steps reduce potential dissimilarities between the Jimple code caused by differences in the bytecode produced by different Java compiler.

In a third step, SootDiff further optimizes the created Jimple representation, and thus reduces dissimilarities. Currently, SootDiff contains an optimization step to reduce dissimilarities when constructing strings in Java using the `java.lang.StringBuilder` API, as shown in the Figure 2 in Section 2. In the future, one can add further steps for optimizing and evaluating simple arithmetic expressions. For instance, an optimization step may transforms an instruction of the form `int val = 7 + 5 * 3 + x` to `int val = 22 + x` by evaluating the arithmetic expression, which is, partially done by the `ecj`. Finally, we compare the optimized Jimple representations using Myers' Diff algorithm [14] and report the differences in the form of diff chunks using `apache:commons:DiffBuilder`.

4 Evaluation

In the following, we present the results when using SootDiff to check if two bytecode classes originate from the same Java source code. Therefore, we apply SootDiff on the bytecode generated by different Java compilers and for different Java versions. As Java source code examples, we use the Java sample programs provided by the University of Illinois [7], which cover most features of the Java library classes. To generate the test cases shown in Table 1, we compile the Java source code using the compilers `javac`, `ecj`, `gcj` for the language versions 1.5 to 1.8. As reference bytecode classes, we generate bytecode for Java 1.8 using `javac`.

[8] https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/builder/DiffBuilder.html

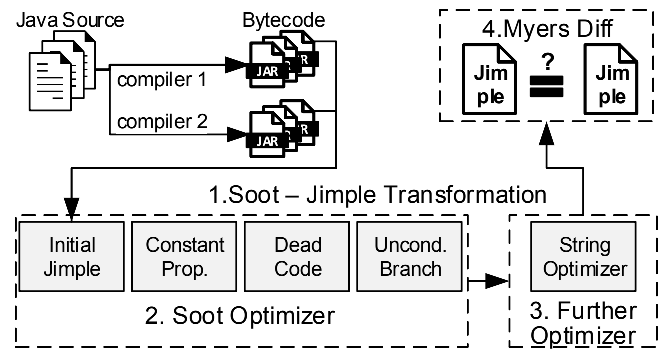

Figure 4. Overview of SootDiff's approach

In our evaluation, we compare the diff-results gained from comparing the different bytecode directly against the diff-results produced by SootDiff. To compare the "plain" bytecode of two classes, we first parse the generated bytecode and generate a textual representation using ASM's `org.objectweb.asm.util.TraceClassVisitor`. Afterwards, we compare the textual representations generated by the `TraceClassVisitor` using the established diff library `com.github.difflib.DiffUtils`. Thereby, we ignore the Java language version information contained in the bytecode. To back up our results, we (re-)validate them running the Unix tool `diff` on the binary files.

To compare the Jimple code generated by SootDiff, we apply SootDiff on the generated bytecode classes and create diff chunks using Myers' diff algorithm. We report two classes as equal if SootDiff does not report any diff chunks.

Table 1 shows the results of our comparison. The table highlights in green the test cases for which the bytecode comparison fails but SootDiff produces correct results. The table shows that for only 58 out of 144 test cases the bytecode comparison bytecode succeeds, although we ignore the Java version information in the generated `.class` files. Consequently, the generated bytecode contains more differences than the Java version, which validates our initial assumption that different compilers produce different bytecode for the same source code. In contrast, the table also shows that for 102 out of 144 test cases the comparison of the Jimple code succeeds. Even without any Jimple optimization steps, 66 test cases succeed.

The test case `DivBy0` fails because `javac` and `ecj` optimize unused local differently, as shown in Figure 5a-5c. The bytecode generated by ecj optimizes the assignment to the variable `i` in Figure 5a in Line 6, whereas the javac leaves the assignment unchanged in Line 6 in Figure 5b. Since the divide instruction may throw an `ArithmeticException` Soot does not remove the assignment statement.

The further 36 test cases fail because `ecj` and `javac` optimize control structures differently, leading to a different organization of basic blocks and branch instructions. For instance, the test case `KeyboardReader` contains a while loop

Table 1. Comparison results based on `bytecode/Jimple`

	javac			ecj				gcj	
	1.5	1.6	1.7	1.5	1.6	1.7	1.8	1.5	1.6
ArrayDemo.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
DivBy0.java	✓/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
FunctionCall.java	✓/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●
HelloData.java	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●
HelloWorld.java	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●
HelloWorldException.java	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●	✓/●
KeyboardIntegerReader.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
KeyboardReaderError.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
KeyboardReader.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
MyFileReader.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
MyFileWriter.java	✗/●	✓/●	✓/●	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍	✗/❍
Point2d.java	✗/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●
Point3d.java	✓/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●
PointerTester.java	✓/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●
PointerTester$Point2d.java	✗/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●
PointerTester$Point3d.java	✓/●	✓/●	✓/●	✗/●	✗/●	✗/●	✗/●	✗/●	✗/●

[1] results are in the form `bytecode/Jimple`
[2] bytecode: fail ✗/ success ✓, Jimple: fail ❍/ success ●
[3] Sample Java Classes from the University Illinois [7]
[4] The bytecode comparison ignores the different bytecode version number in the generated `.class` files.
[5] Highlighted in green where SootDiff comparison succeeds but the Bytecode differs.

```
1 void funct2(){
2 println("");
3 int i, j, k;
4 i = 10;
5 j = 0;
6 k = i/j;}
```

(a) Java DivBy0

```
1 void funct2(){
2 int i2;
3 PrintStream $r0;
4 $r0=PrintStream.out;
5 invoke $r0.<println>;
6 i2 = 10 / 0;
7 return;}
```

(b) Jimple from bytecode generated with `javac` target ver. 1.8

```
1 void funct2(){
2 PrintStream $r0;
3 $r0=PrintStream.out;
4 invoke $r0.<println>;
5 return;}
```

(c) Jimple from bytecode generated with `ecj` target ver. 1.8

Figure 5. Compiler Optimizations: `javac` vs. `ecj`

that depends on a condition of the form `while z != 0`. Javac transforms this condition to an conditional jump of the form `if z == 0 goto end of loop`, whereas the `ecj` generates a conditional jump of the form `if z!=0 goto loop`. Currently, we do not provide an optimization step in SootDiff for these differences.

5 Related Work

Finding equal or similar parts within source code is a well-known problem called *code clone detection*. The level of code similarity is categorized into three different types taken from Koschke [10].

Type 1 clones are an exact copy of the original code without modifications, except for whitespaces and comments. Type 2 clones are syntactically identical copies with only slight renaming, e.g., renaming variables or function identifiers. Type 3 clones are copies with further modifications, e.g., addition or deletion of statements.

Selim et al. [17] present an approach that is similar to SootDiff from a technical point of view; like SootDiff they use Soot’s intermediate representation Jimple to detect Type 3 source code clones. As an input, the approaches solely uses Java source code files to produce Jimple code. To detect clones, they apply existing Java Source clone detection tools on the Jimple representation. For reporting detected clones to users, they map the results from Jimple back to Java.

While the presented approach is similar to SootDiff, there exist significant differences. First, we assume that no Java sources are available, and thus we solely work on bytecode. Second, we do not run existing code clone detection tools for source code but tailored the comparison to Jimple itself. Third, we do not aim to detect Type 3 clones but aim to detect if two bytecode classes, which may have been generated by different Java compiler, originate from the same source code. Consequently, we cannot rely on existing source code clone detection techniques.

Bytecode Clone Detection Baker and Manber [1] present a clone detection approach based on disassembled bytecode using the tools Siff, Dup, and Diff. Thus, the presented approach reports the clone detection results based on untyped stack-based bytecode. In contrast, SootDiff uses Jimple to detect equal classes and methods, which is a typed three-address code, and thus closer to source code than bytecode. Thus, the results are easier to understand and interpret from

a developer's perspective. Moreover, the comparison based on Jimple, as well as Soot's optimization steps allow us to reduce dissimilarities introduces by different Java compilers, and thus enables clone detection across compilers.

Source Code Clone Detection Several approaches for different programming languages exist that aim to detect clones in source code [4, 8, 9, 13].

To this end, these approaches apply different clone detection techniques on the source code directly, e.g., string-based, token-based, abstract syntax tree (AST-)based, metric-based, etc. [10, 16]. In contrast, SootDiff uses the Jimple intermediate representation since we aim to detect if the bytecode of two bytecode artifacts is equal even if no source code is available. While the source code of classes can be recovered successfully using decompilers, which makes existing source code clone detections applicable, the source code obtained from a decompiler does not benefit from the simplifications and optimizations the intermediate representations Jimple offers, e.g., reduced instruction set.

6 Conclusion

In this paper, we present SootDiff, an approach to compare the bytecode generated by different Java compilers based on Soot's intermediate representation Jimple. For the comparison of the parsed Jimple representation, we rely on the established Myers' diff algorithm. Although SootDiff currently uses only a String optimization step in combination with Soot's default optimizers, e.g., dead code eliminators, our evaluation shows that SootDiff produces promising results. However, our results also show that further optimization steps that reduce dissimilarities, e.g., the organization of basic blocks and control structures, will improve the detection of code clones further. In the future, we plan to add additional optimization and unification steps to improve SootDiff's performance. Finally, we plan to use more advanced code clone detection techniques in addition to Myers' algorithm.

References

[1] Brenda S. Baker and Udi Manber. 1998. Deducing Similarities in Java Sources from Bytecodes. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '98)*. USENIX Association, Berkeley, CA, USA, 15–15. http://dl.acm.org/citation.cfm?id=1268256.1268271

[2] V. Bauer, L. Heinemann, and F. Deissenboeck. 2012. A structured approach to assess third-party library usage. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 483–492. https://doi.org/10.1109/ICSM.2012.6405311

[3] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20, 5 (oct 2015), 1275–1317. https://doi.org/10.1007/s10664-014-9325-9

[4] Ira D. Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance (ICSM '98)*. IEEE Computer Society, Washington, DC, USA, 368–. https://doi.org/10.1109/ICSM.1998.738528

[5] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. 2017. CodeMatch: Obfuscation Won'T Conceal Your Repackaged App. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 638–648. https://doi.org/10.1145/3106237.3106305

[6] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. 2011. On the Extent and Nature of Software Reuse in Open Source Java Projects. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Klaus Schmid (Ed.). Vol. 6727 LNCS. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 207–222. https://doi.org/10.1007/978-3-642-21347-2_16

[7] Prof. Robert H. (Bob) Sloan University Illion. [n.d.]. Java Example Program. Retrieved 2019-03-16 from https://www.cs.uic.edu/~sloan/CLASSES/java/

[8] J. Howard Johnson. 1994. Substring matching for clone detection and change tracking. In *Proceedings International Conference on Software Maintenance ICSM-94*. IEEE Comput. Soc. Press, 120–126. https://doi.org/10.1109/ICSM.1994.336783

[9] T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (jul 2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

[10] Rainer Koschke. 2007. Survey of Research on Software Clones. In *Duplication, Redundancy, and Similarity in Software (Dagstuhl Seminar Proceedings)*, Rainer Koschke, Ettore Merlo, and Andrew Walenstein (Eds.). Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany. http://drops.dagstuhl.de/opus/volltexte/2007/962

[11] Raula Gaikovina Kula, Daniel M. German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do Developers Update Their Library Dependencies? *Empirical Softw. Engg.* 23, 1 (Feb. 2018), 384–417. https://doi.org/10.1007/s10664-017-9521-5

[12] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. 2011. The Soot framework for Java program analysis: a retrospective. *Cetus '11* October 2011 (2011). https://sable.github.io/soot/resources/lblh11soot.pdf

[13] Mayrand, Leblanc, and Merlo. 1996. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of International Conference on Software Maintenance ICSM-96*. IEEE, 244–253. https://doi.org/10.1109/ICSM.1996.565012

[14] Eugene W. Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (nov 1986), 251–266. https://doi.org/10.1007/BF01840446

[15] Oracle Corporation. [n.d.]. The Java programming language Compiler Group. Retrieved 2019-03-16 from http://openjdk.java.net/groups/compiler/

[16] Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. 2018. A Comparison of Code Similarity Analysers. *Empirical Softw. Engg.* 23, 4 (aug 2018), 2464–2519. https://doi.org/10.1007/s10664-017-9564-7

[17] Gehan M.K. Selim, King Chun Foo, and Ying Zou. 2010. Enhancing Source-Based Clone Detection Using Intermediate Representation. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 227–236. https://doi.org/10.1109/WCRE.2010.33

[18] Raja Vallée-Rai and Laurie Hendren. 1998. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. Technical Report. McGill University, Montreal, Canada. 1–15 pages. http://www.sable.mcgill.ca/publications/techreports/sable-tr-1998-4.ps