Java Bytecode Normalization for Code Similarity Analysis

Stefan Schott
Paderborn University, Germany

Serena Elisa Ponta
SAP Security Research, Mougins, France

Wolfram Fischer
SAP Security Research, Mougins, France

Jonas Klauke
Paderborn University, Germany

Eric Bodden
Paderborn University, Germany
Fraunhofer IEM, Paderborn, Germany

Abstract

Analyzing the similarity of two code fragments has many applications, including code clone, vulnerability and plagiarism detection. Most existing approaches for similarity analysis work on source code. However, in scenarios like plagiarism detection, copyright violation detection or Software Bill of Materials creation source code is often not available and thus similarity analysis has to be performed on binary formats. Java bytecode is a binary format executable by the Java Virtual Machine and obtained from the compilation of Java source code. Performing similarity detection on bytecode is challenging because different compilers can compile the same source code to syntactically vastly different bytecode.

In this work we assess to what extent one can nonetheless enable similarity detection by *bytecode normalization*, a procedure to transform Java bytecode into a representation that is identical for the same original source code, irrespective of the Java compiler and Java version used during compilation. Our manual study revealed 16 *classes of compilation differences* that various compilation environments may induce. Based on these findings, we implemented bytecode normalization in a tool jNorm. It uses Jimple as intermediate representation, applies common code optimizations and transforms all classes of compilation difference to a normalized form, thus achieving a representation of the bytecode that is identical despite different compilation environments.

Our evaluation, performed on more than 300 popular Java projects, shows that solely the act of incrementing a compiler version may cause differences in 46% of all resulting bytecode files. By applying bytecode normalization, one can remove more than 99% of these differences, thus acting as a crucial enabler for subsequent applications of bytecode similarity analysis.

2012 ACM Subject Classification Software and its engineering → Compilers

Keywords and phrases Bytecode, Java Compiler, Code Similarity Analysis

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.37

Supplementary Material *Software (ECOOP 2024 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.10.2.20`

Funding This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre ”On-The-Fly Computing“ (GZ: SFB 901/3) under the project number 160364472.

1 Introduction

In the past, researchers have developed many approaches for code similarity analysis on Java applications [50, 35, 52, 31, 60, 47]. These techniques target a wide variety of applications, like code clone detection, plagiarism detection, copyright infringement investigation, program comprehension, vulnerability detection and many more [49, 37]. Most developed techniques operate on source code. However, an application's source code is not always available, since applications are typically distributed in binary form. Especially in scenarios where external dependencies are included into a software product, often only the binary form is included without the corresponding source code. In case of Java, applications are distributed as JAR-archives that contain the *bytecode* of the application. Instead of compiling the source code directly to executable machine code, Java compilers generate an intermediate representation called *bytecode*, which is translated into machine code during execution time by the Just-in-time (JIT) compiler within the Java Virtual Machine (JVM). With the European Union's Cyber Resilience Act [11] coming into force soon and the US's Executive Order on Improving the Nation's Cybersecurity [6] already being effective, the creation of Software Bill of Materials (SBOM) has become mandatory. However, creating a faithful SBOM for Java applications is a difficult undertaking due to current tool's reliance on metadata [5, 13]. To reliably create such SBOMs an approach needs to be established that is able to find all used components based on the similarity of bytecode, since source code is generally not available.

There are only few approaches that have been developed for similarity analysis based on bytecode [4, 36, 58]. This may be due to the increased complexity when trying to compare bytecode instead of source code. As Dann et al. [12] and Kononenko et al. [38] have shown, the comparison of bytecode is more complex than the comparison of source code, since equal source code is compiled into different bytecode, depending on the compiler, version and configuration used. While the generated bytecode is semantically equivalent, its syntactic structure may vastly differ. To overcome this difficulty we investigate the utility of *bytecode normalization* to create a representation that is *independent* of the environment that has been used for compilation. This independent representation can subsequently be used by bytecode-based code clone, plagiarism or vulnerability detectors without the need for consideration of compilation environments, significantly simplifying their task. Our approach to achieve bytecode normalization, which builds upon Dann et al.'s approach [12], is a procedure that

1. translates the bytecode into Jimple, the primary intermediate representation of the Soot [55] bytecode optimization framework, which reduces the more than 200 available bytecode-instructions to only 15 different Jimple instructions,
2. as a baseline first applies common optimizations like constant propagation, dead code removal and unconditional branch folding to further reduce differences, and then specifically, and lastly
3. transforms *compilation differences* induced by different compilation environments.

We uncovered the set of compilation differences by systematically comparing bytecode of popular Java libraries generated by different vendors, versions and configurations of Oracle's Java Development Kit's (JDK) and OpenJDK's compiler (javac). During this initial study, we found a total of 16 classes of compilation differences.

We implemented bytecode normalization in a tool jNorm, and evaluated it on more than 300 of the most popular Java projects on GitHub by compiling the same source code within various compilation environments with different compiler vendors, versions and target levels. The evaluation shows that even a single increase of the compiler version may result in up to 46% of all generated bytecode files containing compilation differences. By applying jNorm's bytecode normalization one can reduce these differences by more than 99%. Thus, bytecode normalization can function as an important enabler for bytecode similarity analysis in all cases in which source code is not available.

To summarize, this paper makes the following original contributions:

- It investigates the usage of different Java compilers and settings in real-world projects.
- It presents a comprehensive set of 16 classes of compilation differences that are induced when using different vendors, versions or target level configurations of the JDK's and OpenJDK's Java compiler.
- It presents an approach to bytecode normalization, implemented in a tool jNorm, to virtually completely remove the differences introduced by compiling the same source code within different compilation environments.
- It evaluates jNorm on a large set of real-world Java applications collected from GitHub.

The remainder of this paper is structured as following. Section 2 introduces terms and concepts related to similarity analysis, Java compilation and normalization. Afterwards, Section 3 presents the concept of Java bytecode normalization implemented in jNorm and an overview of the uncovered compilation difference classes. Section 4 presents an evaluation of jNorm on a a set of real-world Java projects. Related work is presented in Section 5. We discuss possible threats to validity in Section 6 and conclude in Section 7.

jNorm, its source code, more detailed evaluation results and a study on the usage of Java compilers and target levels are publicly available at:

`https://doi.org/10.5281/zenodo.12625104`

2 Background

This section introduces concepts that are related to code similarity analysis and the compilation of Java applications.

2.1 Code Similarity Analysis

Code similarity analysis is a technique that seeks to determine the similarity of two or more code fragments. The calculation of the similarity of code fragments has a large number of uses, like code clone, plagiarism, licensing violation, malware or vulnerability detection [49, 37].

Depending on the desired application area different techniques are employed. Text- [50, 4] or token-based [35, 52] techniques try to find similarities within the textual information of the code fragments. Tree-based techniques [31] try to additionally leverage syntactic information of the code for the similarity analysis. Some techniques even try to find semantic similarities within code fragments [39]. These techniques are typically graph-based and offer low potential for scalability [49]. Recently machine learning based approaches [60, 51], which typically train a classifier that decides how similar code fragments are, have become popular. In terms of efficiency and scalability, text- and token-based techniques, which can solely focus on syntactic features, are much preferred.

Typically the similarity analysis is performed on a source code level. However, the source code of a compiled binary is not always available or trustworthy. In such scenarios, e.g. plagiarism detection or SBOM generation, the similarity analysis has to be performed on the binary itself. However, the resulting binary code is highly dependent on the environment it has been compiled in, i.e. different compiler versions or settings produce different code, even when compiling the same source code [12]. This characteristic makes binary similarity analysis a much more complex task.

2.2 Java compilers

In contrast to other compiled programming languages like C or Go, Java applications are not directly compiled into machine-executable code, but into Java *bytecode*. This bytecode is executed by the Java virtual machine (JVM), which comes with a just-in-time (JIT) compiler that compiles the bytecode into executable machine code at runtime. Because of this, the Java bytecode has the following characteristics:

1. **Platform independence**: The JVM architecture aims at platform independence. The generated bytecode is independent from the platform it is intended to run on [46].
2. **Unoptimized bytecode**: Optimizations are performed during runtime by the JIT compiler, therefore compiled bytecode is typically not optimized [24].

Because of these characteristics, the amount of variance across generated bytecode is not as high, when compared to machine-code, since there are no different optimization levels or differences due to the targeted platform.

There exist multiple different Java compilers like e.g. Oracle's JDK or OpenJDK's compiler [45], Eclipse's JDT core compiler [28], IBM's Jikes compiler [32], or the GNU Compiler for Java (GCJ) [19]. The JDK's and OpenJDK's compilers can be invoked programmatically or through the command-line application *javac* that comes pre-shipped with each JDK. Given the same source code as input, in many cases these compilers produce *different* Java bytecode.

In general, Java compilers support source codes that adhere to different versions of the language specification and can generate bytecode for different JVM versions lower than the compiler's version. This backwards compatibility can be used by setting the compiler's *target level*. For example, bytecode compiled with a JDK11 compiler with target level set to 8 can be executed by a JVM only supporting up to Java 8. The set of available target levels for a compiler is usually limited to a subset of earlier versions.

2.3 Jimple

Jimple is an intermediate representation (IR) of Java bytecode that was designed for providing a format that allows for simplified analysis, optimization and code transformations. Jimple maps the more than 200 Java bytecode instructions to only 15 different Jimple instructions in a *three-address* based representation. Three-address based representation means, that each instruction generally contains at most three different operands, e.g., one used for the left-hand side of an assignment and two used for binary operations on the right-hand side. This restriction greatly simplifies the processing of individual IR statements, which is why three-address IRs are nowadays commonplace. During the transformation Jimple retains all the type information present in the bytecode.

Jimple is the primary IR of the most popular Java bytecode optimization and analysis framework Soot [55]. Alongside various code optimization options, Soot provides an API to conveniently transform Jimple instructions. Soot can automatically convert Java bytecode to Jimple (and vice-versa).

3 Java Bytecode Normalization

As we show next, Java bytecode normalization allows for the removal of differences in Java bytecode that are solely introduced by the usage of different compilation environments. In the following we describe how we detected the compilation differences in the first place, as well as the details of our bytecode normalization approach and its implementation in jNorm.

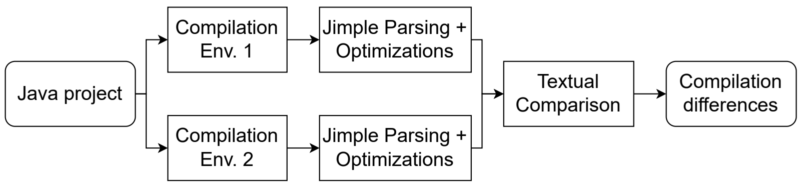


Figure 1 Setup to determine compilation differences.

3.1 Investigation of Compilation Differences

Before the development of our bytecode normalization approach jNorm, we performed a study to investigate the differences induced by different compilation environments. Figure 1 shows the setup we used to determine compilation differences. For each comparison, we supplied the source code of various versions of the popular Java libraries Apache commons-io, Apache commons-lang, Jackson-databind, SLF4J and Google Guava, to two different environments for compilation. Afterwards, we converted the resulting bytecodes to Jimple and applied code optimizations, provided by the Soot framework, to reduce dissimilarities. Finally, we performed a textual comparison on the optimized Jimple representations to determine the remaining compilation differences. Two files were considered different, and manually inspected by the authors, as soon as one character differed in the textual comparison.

Our compilation environments included the javac compilers shipped with JDKs 5–8, 11, and 17. Moreover, this version-range covers all Long-Term-Support (LTS) versions of the Java ecosystem until August 2023. A usage study of Java compilers and target levels in Java projects, which revealed these to be the by far most relevant compilers and versions, is available within an electronic appendix in our provided artifact.

We consider three types of parameters, JDK vendor, JDK version, and Java target level. We used Oracle's JDK, as well as OpenJDKs distributed by Amazon Corretto and Eclipse Adoptium.

In total, our setup revealed 16 compilation difference classes, present in the investigated projects, which are listed in Tables 1a and 1b. Table 1a shows the difference classes produced by changing the JDK version, while Table 1b shows the difference classes produced when adjusting the Java target level. We did not find any vendor-related difference classes in our initial experiments.

Furthermore, we inspected the official JDK release notes [27] related to newly released compiler versions. However, this inspection did not reveal any so far uncovered difference classes. Our evaluation performed on more than 300 of the most popular Java projects (see Sections 4.3 and 4.4) also revealed no additional difference classes.

In the following we describe jNorm's approach to bytecode normalization and how it transforms the identified compilation difference classes into a representation that is common across all investigated compilation environments.

3.2 Overview of jNorm

Figure 2 depicts an overview of jNorm. First, jNorm parses a Java bytecode file (.class file) into Jimple format, which is specifically designed for efficient optimizations and transformations. Note that jNorm also has the capability to process multiple bytecode files at once, and therefore full Java projects, but because each file is normalized independently of others we will explain bytecode normalization of single files. To reduce the initial set of differences for the following steps of the normalization process, jNorm applies different types of common

Table 1 Difference classes on JDK version and Target level change.

(a) JDK version change.

ID	JDK	Compilation Difference Class
N1	5 → 6 & 7 → 8	Synthetically generated methods
N2	5 → 6	Arithmetic
N3	6 → 7	CharSequence toString invocation
N4	7 → 8	Empty try-catch-finally block
N5	7 → 8	String constant concatenation
N6	8 → 11	Method reference operator
N7	8 → 11	Buffer method invocation
N8	8 → 11	Try-with-resources
N9	8 → 11	Duplicate checkcasts
N10	11 → 17	Enums

(b) Target level change.

ID	Target	Compilation Difference Class
N11	6 → 7	Outer class object creation
N12	8 → 11	Dynamic string concatenation
N13	8 → 11	Nest-based access control
N14	8 → 11	Invocation of private methods
N15	8 → 11	Inner class instantiation
N16	multiple[a]	Insertion or removal of typechecks

[a] This compilation difference class occurs across multiple JDK and target level changes.

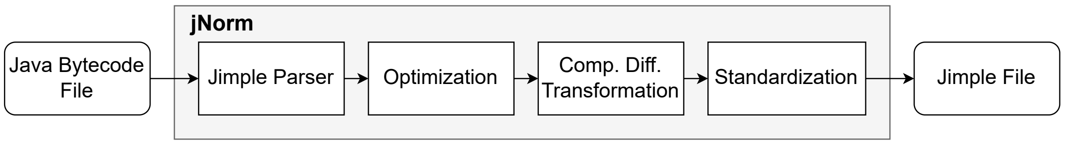


Figure 2 Overview of JNORM.

optimizations to the Jimple representation of the input bytecode file. Afterwards, in the Compilation Difference Transformation step (see Figure 2), JNORM handles the remaining set of compilation differences by performing certain transformations on the optimized Jimple representation. These transformations are targeted towards specific constructs that we found to be compiled differently based on the used compilation environment. JNORM detects these constructs within the target program and transforms them into a normalized representation. The applied transformations interfere with the naming scheme of local variables inside the target programs, which cause the introduction of new dissimilarities. JNORM handles these dissimilarities by standardizing (see Figure 2) the order and naming scheme of variables. After the normalization process is finished, JNORM outputs the normalized Jimple representation.

3.3 Jimple Parsing and Optimization

The first step of Java bytecode normalization consists of parsing the targeted bytecode file into a Jimple representation. This allows for a convenient application of common program optimizations provided by SOOT. We apply the following optimizations to each method [54]:

- **Copy Propagation**: Usages of variables in statements are replaced by their values, e.g. in a statement like `x = y + 3`, the reference to variable `y` is replaced by the value stored in `y`.
- **Constant Propagation and Folding**: Expressions that entirely consist of compile-time constants (e.g. 2 * 3) are replaced by the constant result.
- **Dead Assignment Elimination**: Assignment statements to local variables, whose value is not subsequently used, are removed.
- **Conditional Branch Folding**: The expressions inside if-conditions are statically evaluated. If the expressions evaluate to constants, the unreachable conditional branch statements are removed.

- **Unconditional Branch Folding**: Unnecessary `goto` statements are removed.
- **Unreachable Code Elimination**: Unreachable code is removed.
- **Null Check Elimination**: Null-check statements, where the checked variable is known not to be null, are removed.
- **Unused Local Elimination**: Unused local variables within a method are removed.

These optimizations already contribute to a decrease of dissimilarities introduced during compilation [12]. However, after applying the optimizations, many important compilation differences still remain, which are targeted in the next step.

3.4 Compilation Difference Transformation

Through our investigation (see Section 3.1) we identified 16 compilation difference classes summarized in Tables 1a and 1b. In the following we describe the identified classes and the transformations applied by jNorm in detail.

The transformations that jNorm performs are not arbitrarily chosen. Each transformation produces a version that is generated by at least one compiler within our dataset. Furthermore, the decision whether to transform a compilation difference class to the older or the newer version is also not arbitrary. Typically one of the two versions contains more information than the other (e.g. a more specific return type in the newer version or the amount of string concatenation calls before their combination into a single call). As we cannot simply add information that is unavailable when only having access to the bytecode, we have to transform the difference class to the version that contains less information, therefore stripping some information from the generated bytecode. However, this information cannot be used for similarity analysis, since, based on the used compilation environment, it is not guaranteed to be present in the bytecode.

Note that we do not aim at generating an executable version of the bytecode with all semantics preserved, but at preserving information that is possibly important for a similarity analysis. Similarity analysis approaches that additionally require an executable version of the analyzed application, can use the original bytecode that has not been normalized, in addition to the normalized version.

N1: Synthetically generated methods

In many cases the JDK compiler synthetically generates methods into classes. Often this is used to generate bridge-methods that enable access to private members. Such synthetically generated methods are marked by the compiler with a specific `synthetic` flag [34]. Depending on the used JDK, these methods are not always generated in certain cases, e.g. whenever a method of a class uses a `Comparator` to create a specific ordering of objects, starting from JDK6 the compiler automatically generates a corresponding `sort` method into the class. Thus, such synthetic methods introduce differences and cannot be reliably used for a code comparison.

Transformation: jNorm removes such synthetic methods from the Jimple representation, as they cannot be modified within the source code anyway.

N2: Arithmetic

In some cases, integer subtractions are replaced with additions of negative numbers inside the bytecode produced by JDK6 and higher. A statement like `i1 = i1 - 5`, generated by JDK5, is replaced by a conversion of the positive number to a negative one (`i1 = (int) -5`) and a subsequent addition with the negative number like `i2 = i2 + i1`, by JDK6 and higher.

Listing 1 `toString()` invocation (Jimple)

```
java.lang.CharSequence r1;
java.lang.String r2;

// JDK6:
r2 = virtualinvoke r1.<java.lang.Object: java.lang.String toString()>();

// JDK7:
r2 = interfaceinvoke r1.<java.lang.CharSequence: java.lang.String toString()>();
```

Transformation: Whenever JNORM identifies an addition involving negative integers, it converts it into a subtraction.

N3: CharSequence toString invocation

The JDK7 compiler changed the way the `toString` method is handled in the bytecode when invoked on an object of type `CharSequence`. As it can be observed in Listing 1 (subtle differences are highlighted within the listings), the invoke type `interfaceinvoke` replaced `virtualinvoke`, and the more specific type `java.lang.CharSequence` replaced the method return type `java.lang.Object`.

Transformation: Whenever JNORM identifies a call to a `toString` method with a `java.lang.CharSequence` return type, it converts the method call to its previous, more generic, version.

N4: Empty try-catch-finally block

In most cases a try-catch-finally block comes with one or more catch blocks that react to some types of thrown exceptions. However, catch blocks can be empty or even missing completely. A try-catch-finally block with empty (or even missing) catch blocks is a syntactically valid Java construct, used to execute some instructions, no matter what happens in the try block. Prior to JDK8, the JDK compiler produces a redundant exception catching block[1] in the bytecode, if a catch block is empty or missing.

Transformation: If JNORM identifies such redundant exception catching blocks, it removes them from the Jimple representation of the bytecode.

N5: String constant concatenation

When using the JDK8 or higher compiler, string concatenation optimizations are introduced. Whenever multiple string constants are concatenated, compilers prior to JDK8 would use multiple calls to the `StringBuilder.append` method. A simple concatenation like

```
String helloWorld = "Hello " + "World!";
```

would result in two calls to the `StringBuilder.append` method, one receiving “Hello” and the other receiving “World!” as argument. However, as of JDK8, the compiler concatenates these two strings at compile time and produces a single call to `StringBuilder.append`. This holds true only for subsequent string constants: whenever a substring assigned to a variable is involved in the concatenation, multiple `StringBuilder.append` calls are used.

[1] In bytecode and Jimple there exists no notion of catch blocks. We use this terminology in synonym with exception traps.

Listing 2 Method reference operator usage (Jimple)

```
org.apache.commons.io.IOFileFilter r0;

// JDK8:
virtualinvoke r0.<java.lang.Object: java.lang.Class getClass()>();

// JDK11:
staticinvoke <java.util.Objects:
  java.lang.Object requireNonNull(java.lang.Object)>(r0);
```

Listing 3 Buffer method invocation (Jimple)

```
java.nio.ByteBuffer r0;

// JDK8:
virtualinvoke r0.<java.nio.ByteBuffer: java.nio.Buffer flip()>();

// JDK11:
virtualinvoke r0.<java.nio.ByteBuffer: java.nio.ByteBuffer flip()>();
```

Transformation: When JNORM identifies subsequent calls to the `StringBuilder.append` method with string constants as arguments that are not referenced by variables, it combines them into a single call.

N6: Method reference operator

With the release of JDK8, the method reference operator (::) was introduced to the Java programming language. It allows one to refer to a method with the help of its declaring class or object name and is especially useful in combination with streams. Listing 2 shows how the operator usage is handled during compilation. Before performing the actual method call, if the operator is referring to a method of an object, a null check is performed at runtime. This is done to ensure that the object, the referred method belongs to, actually exists and is not `null`. The usual way to perform null checks in JDK8 and lower is to call the method `getClass` on the object to check. This mechanism was replaced in newer JDKs by invoking the static `requireNonNull` method.

Transformation: For normalization, JNORM transforms all occurrences back to the old null-checking mechanism.

N7: Buffer method invocation

Starting from JDK11, the return type of all subclasses of `java.nio.Buffer` was further specified. Instead of returning the type `java.nio.Buffer` (cf. Listing 3), newer JDKs further specify the return type. Listing 3 shows that methods of the class `java.nio.ByteBuffer`, compiled with JDK11, return `ByteBuffer` instead of `Buffer`. This holds true for every subclass of `java.nio.Buffer` and any method returning a `Buffer` object.

Transformation: When JNORM finds the invocation of a method of a `java.nio.Buffer` subclass with `Buffer` as return type, it transforms the return type to the more specific type.

N8: Try-with-resources

The try-with-resources statement allows to declare resources that are used within the statement, which are guaranteed to be closed at the end, no matter if an exception is thrown.

Whenever a try-with-resources statement is used in the source code, the JDK compiler produces multiple exception handlers that wrap each other in the bytecode, since the bytecode does not provide a separate instruction for such a statement. In some cases, prior to JDK11, these wrapped exception handlers are redundant, since they do not cover any application code but only automatically generated exception handling code. These redundant exception handlers are not created as of JDK11.

Transformation: Whenever jNorm identifies an exception handler that only covers automatically generated exception handling code, it removes the exception handler and its corresponding code from the declaring function.

N9: Duplicate checkcasts

Due to a bug [18] fixed in JDK11, earlier JDK compilers may insert the same `checkcast` instruction twice, one after the other.

Transformation: jNorm removes redundant typechecks for normalization, if it identifies such duplicates.

N10: Enums

Enums in Java are special types that can only take on certain predefined values. When an enum is created, the JDK compiler creates a separate class for each enum and defines the possible values inside the `clinit` function, which acts as a static initializer. In contrast to a constructor, which is called when an object of a class is initialized, the `clinit` function is called when the class itself is initialized. Prior to JDK17, the initialization of the possible enum values is performed directly inside the `clinit` method, while in JDK17 the definition is moved to its own function, which is called from `clinit`.

Transformation: If jNorm detects that the enum values are initialized within the `clinit` method, it moves the initializations into its separate method and calls this method from `clinit`.

N11: Outer class object creation

Changing the target level from Java 6 to Java 7 changes the generated bytecode, when an inner class creates an object of another sibling inner class within their shared outer class as shown in the following listing:

```
SiblingInnerClass sic = getOuterClass().new SiblingInnerClass();
```

In this case the method `getOuterClass` returns a reference to the outer class shared by both inner classes, the one that contains the above statement and the one that is created by the statement. Whenever this is the case, the compiler inserts a check to verify, that the method `getOuterClass` does not return `null`. This is done in the same way, as described for difference class N6, where the previous way of performing a null-check via the `getClass` method is replaced by a call to the `requireNonNull` method.

Transformation: jNorm transforms all occurrences back to the old null-checking mechanism, as it does for difference class N6.

Listing 4 String concatenation (Jimple)

```
 1  int i0;
 2  java.lang.StringBuilder $r0, $r1, $r2, $r3;
 3  java.lang.String[] r5;
 4
 5  // Target Level 8:
 6  $r0 = new java.lang.StringBuilder;
 7  specialinvoke $r0.<StringBuilder: void <init>()>();
 8  $r1 = virtualinvoke $r0.<StringBuilder:
 9   StringBuilder append(java.lang.String)>("Amount: ");
10  $r2 = virtualinvoke $r1.<StringBuilder:
11    StringBuilder append(int)>(i0);
12  $r3 = virtualinvoke $r2.<StringBuilder:
13    StringBuilder append(java.lang.String)>(" Pieces");
14  virtualinvoke $r3.<StringBuilder: java.lang.String toString()>();
15
16  // Target Level 11:
17  dynamicinvoke "makeConcatWithConstants" <java.lang.String (int)>(i0)
18    <java.lang.invoke.StringConcatFactory:
19      java.lang.invoke.CallSite makeConcatWithConstants(
20        java.lang.invoke.MethodHandles$Lookup,
21        java.lang.String, java.lang.invoke.MethodType,
22        java.lang.String, java.lang.Object[]
23    )>("Amount: \u0001 Pieces");
```

N12: Dynamic string concatenation

In Java 11 and higher the old string concatenation approach of repeatedly calling the `StringBuilder.append` method (see N5), is replaced by a single `invokedynamic` instruction, which defers the resolution of a method call to runtime. This change was introduced to optimize the performance of string concatenations [25]. Listing 4 showcases the differences of string concatenation compiled for target levels 8 and 11. Previously, for each part of the string concatenation, one call of the `StringBuilder.append` method was required. However, in the new version, a dynamic approach that looks similar to template-based string building is generated. A single dynamic call of the `makeConcatWithConstants` method is performed, where string constants are concatenated into a single constant, while dynamic values are expressed by placeholders (see `\u0001` in line 23 in Listing 4) which are replaced by the resolved value during runtime.

Transformation: JNORM transforms the old string concatenation procedure into a template-based concatenation using `invokedynamic`.

N13: Nest-based access control

With the release of Java 11, a new concept for accessing members of inner classes, called nest-based access control [43], was introduced to the language specification. When inner classes are defined within a class, the JDK compiler compiles each inner class into its own file. The JVM treats each class as a separate entity and therefore disallows access to private members from methods outside of the class. However, the Java language specification *does* allow such access to private members of inner classes if they are originating *from the outer class* and vice versa. Prior to the release of Java 11, such access was handled by the compiler generating public bridge methods in the inner class for each private member, that the outer class can use to circumvent calling a private method. Starting from Java version 11, this indirect access via generated bridge methods is not necessary anymore. A new property has been introduced that marks inner classes as *nestmates* of their outer class, which tells the JVM that access to private members is explicitly allowed between the marked classes. The JVM then automatically puts appropriate access-control checks into place. This change was introduced due to transparency, simplicity and security reasons.

Transformation: If jNorm finds classes that use bridge-methods to access private members of their respective inner classes, it transforms them to the nest-based access pattern created when specifying target level 11.

N14: Invocation of private methods

On top of adding nest-based access control, Java 11 comes with a new way to invoke private methods, even within the same class. Prior to Java 11, all private methods were invoked via the `invokespecial` instruction. With Java 11, to be consistent with the rules of the nest-based access control specification, certain private-method invocations were changed to use `invokevirtual` instructions [43].

Transformation: jNorm transforms private method invocations to use `invokevirtual` instead of `invokespecial`.

N15: Inner class instantiation

Going from Java 8 to Java 11, the instantiation of inner classes was changed. In some cases, in Java 8 and earlier, when an inner class is instantiated within the outer class, the JDK compiler generates an additional anonymous class that is empty. This behavior serves no apparent purpose and was removed in Java 11.

Transformation: If jNorm finds empty anonymous classes, it removes them.

N16: Insertion or removal of typechecks (aggressive transformation)

To check the type of an object, the bytecode instruction `checkcast` is used. Among other things, it is used when the developer performs a typecast on an object, so that the JVM can verify whether the specified type is suitable for the object. However, when changing the JDK version or target level, the compiler's behavior regarding typechecks changes. In contrast to the other compilation difference classes, this difference class cannot be isolated to a single version change, as it happens to different extents at various JDK version or target level changes. In some cases the compiler inserts `checkcast` instructions even though the developer did not write a typecast, or it does not place a `checkcast` instruction for typecasts placed by the developer. Whether the compiler places a `checkcast` instruction or not often depends on the used compilation environment.

Transformation: By default jNorm does not transform such typechecks, as we were not able to detect a pattern that indicates whether a typecheck should be removed or inserted, by just having access to the bytecode. Still, jNorm offers an *aggressive normalization mode* where it removes all `checkcast` instructions from the normalized Jimple representation of the bytecode. Such transformation removes information that can be used for similarity analysis and possibly changes the application's semantics rather than just adopting a format produced in a different compilation environment. In some cases, e.g. when the change between two bytecode fragments only consists of typecheck insertions or removals, this loss of information makes the normalized fragments indistinguishable.

```
i1 = i1 - 1;
i2 = i1 + 10;
```

(a) JDK5 (not normalized)

```
i1 = (int) -1;
i2 = i2 + i1;
i3 = i2 + 10;
```

(b) JDK6 (not normalized)

```
i1 = i1 - 1;
i3 = i1 + 10;
```

(c) JDK6 (normalized)

Figure 3 Application of standardization (Jimple).

We leave a more thorough investigation of the patterns that indicate typecheck placements in the bytecode as future work.

3.5 Standardization

Since the names of local variables are removed by default after compiling Java source code into bytecode (bytecode uses an operand stack instead of local variables), all local variables within the Jimple representation are named by concatenating their inferred type with an ascending integer number. After applying transformations that create, remove, or reorder local variables, such as the Arithmetic or Try-with-resources transformations, the ordering of local variable definitions and their naming scheme might become inconsistent. Because of this, we remove unused local variables and reorder definitions of used local variables based on their usage order, which stays consistent during all optimizations and transformations. Afterwards, we rename the local variables based on their types and usage order. This ensures a standardized naming scheme across all methods, even after applying transformations.

Figure 3 shows why standardization is necessary in some cases. Listing 3a shows subtraction generated by the JDK5 compiler, while Listing 3b shows subtraction output by the JDK6 compiler. After applying normalization to the code fragment in Listing 3b (see N2: Arithmetic), we obtain the code shown in Listing 3c. Since we removed the intermediate variable `i2`, the logical naming following the variable deletion does not match up anymore to the version that did not require any normalization. Therefore we need to apply standardization and rename every following variable usage, to achieve a representation that is equal to the code fragment that did not require normalization.

4 Evaluation

In the following we evaluate JNorm's normalization performance. To do so, we answer the following research questions.

RQ1: Does the JDK vendor influence the bytecode generation?

RQ2: How does JNorm perform on changing JDK versions?

RQ3: How does JNorm perform on changing Java target levels?

RQ4: To which degree can bytecode normalization support similarity analysis tools?

RQ5: How prevalent are the individual compilation difference transformations of JNorm?

The first three research questions focus on JNorm's normalization performance within different compilation environments. Research question 4 investigates to which extent JNorm can support similarity analysis tools. The final research question gives an overview about the most common compilation difference classes. We used similar experimental setups for each of the research questions.

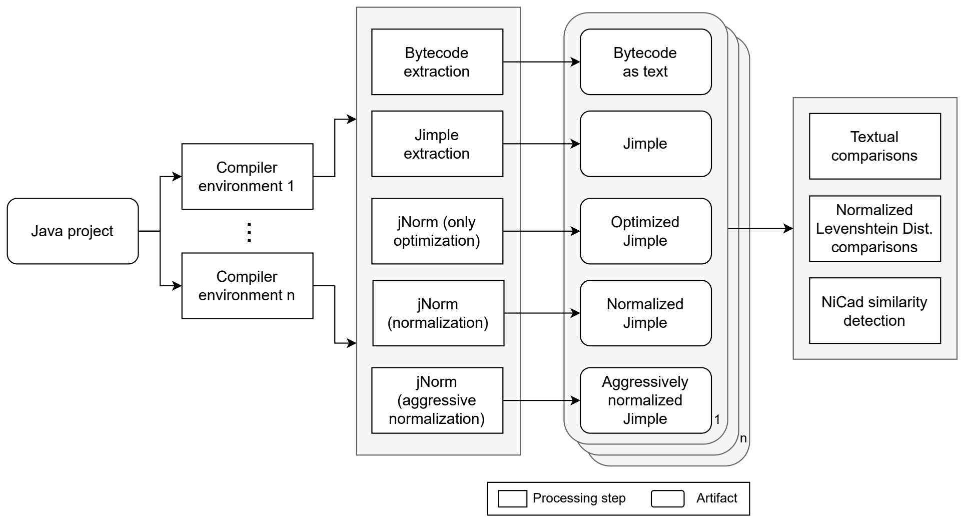


Figure 4 Overview of our experimental setup.

4.1 Experimental Setup

To evaluate jNorm's normalization performance, we use the approach depicted in Figure 4.

We selected real-world Java projects based on the following process: At first, we used the GitHub search API to obtain the 1,000 projects with the most stars that have Java listed as their main language as of August 2023. We excluded two projects that, alongside Java files, also contained other JVM-based programming languages like Groovy or Clojure, as the compiled classes would interfere with further evaluation steps. Then we filtered out every project that does not use Maven as build tool, as Maven's static configuration files in XML format, unlike Gradle, allow for an automated change of the compilation setup without knowing the project's build structure in detail. After this step we were left with 322 Maven projects. Finally, we excluded two projects that, when compiled twice within the same environment, would produce different results, because of code generation at compilation time. This is typically due to files being generated for testing purposes or due to parser code being generated from a grammar, which in some cases produces random identifiers. This left us with a set of 320 Java projects, including tutorial projects, popular libraries, frameworks and real-world applications.

We cloned each project's git repository. As automatic compilation is a known problem for Java projects [22], to increase the chances of a successful compilation in the next step, we then moved to the latest release tag (if available). As depicted in Figure 4, we compiled each of the projects within different compilation environments. We chose the compilation environments based on the setting we were interested in for the respective research questions.

To evaluate the normalization performance of jNorm, we applied different procedures to the compiled projects, as shown in Figure 4. The "Bytecode extraction" component in the figure uses ASM 9.3 [3] to extract the textual representation of the bytecode from the compiled class files (omitting all debug information). Furthermore, we additionally extracted the plain Jimple representation of the compiled classes in textual form without applying any optimizations or transformations. The plain bytecode and Jimple can be used as a baseline to establish the amount of differences induced by different compilation environments. To establish how the different normalization steps of jNorm contribute to the removal of compilation differences, we let jNorm run in different modes. "jNorm (only optimization)" only applies the Jimple parsing and optimizations described in Section 3.3. "jNorm (normalization)" applies all the steps described in Section 3 with transformations

Table 2 JDKs considered in our evaluation.

JDK Version	Oracle JDK	AC OpenJDK	EA OpenJDK
7	1.7.0_80	–	–
8	1.8.0_333	8.342.07.4	8u352-b08
11	11.0.16	11.0.16.9.1	11.0.17+8
17	17.0.4.1	17.0.5.8.1	17.0.5+8

N1–N15, but keeps all typechecks in place (default normalization mode). "jNorm (aggressive normalization)" differs from the previous as it also removes all typechecks from the resulting Jimple representation. Applying all procedures, we obtain five sets of files per compilation environment and project:

- Extracted bytecode as text
- Extracted Jimple
- Optimized Jimple
- Normalized Jimple
- Aggressively normalized Jimple

We apply different comparisons to each of the resulting file sets generated within different compilation environments, resulting in multiple comparisons per project. At first we perform a textual head-to-head comparison on the file-, as well as method-level. To do so we compare files with the same fully qualified name and methods with the same signature to each other that were produced within different compilation environments. As soon as there is a single textual difference between the compared files or methods, they are classified as being *different*. If a method is present in one file, but not the other, it is classified as *disjunct*.

In addition to textual head-to-head comparisons, which only allow for a yes/no detection of equality, we calculate the normalized Levenshtein Distance (NLD) [59] between the compared files and methods. The NLD is a measure that is used to calculate the similarity of two text sequences. The Levenshtein Distance counts the number of required character insertions, deletions or substitutions to transform one text sequence into the other. The normalized Levenshtein Distance additionally takes the length of the text sequences into account and produces a similarity value between 0% and 100%, with 100% indicating that every single character needs to be changed and 0% indicating that both text sequences are identical. Lastly, we include the similarity analysis tool NiCad [50] into our comparison process to evaluate to which degree the prior application of bytecode normalization can improve the performance of similarity analysis tools. We use NiCad for our experiment, as it is one of the most popular similarity analysis tools.

Table 2 shows the JDKs considered in our evaluation. We considered all Java Long-Term-Support versions up to August 2023. According to a 2022 survey on the state of the Java ecosystem [44], our JDK selection covers more than 97% of JDK versions used in projects. This gives us an indication for the representativeness of our version selection. Furthermore, we considered the three most popular JDK vendors according to the survey, which include Oracle's JDK, Amazon Corretto's (AC) OpenJDK and Eclipse Adoptium's (EA) OpenJDK, in our evaluation. Note that AC and EA do not distribute OpenJDK versions prior to version 8. Moreover, only a single project within our dataset can be compiled using Oracle's JDK5 and JDK6, thus we do not consider these two no longer supported JDKs in our evaluation [26].

We executed the compilations and normalizations on a Debian 10 system, configured to use four cores of an Intel Xeon E5-2695 v3 (2.30 GHz) CPU and 32GB of main memory. We used Maven 3.8.6 for the invocation of builds.

4.2 RQ1: Does the JDK vendor influence the bytecode generation?

Before assessing the differences introduced when using different JDK or Java versions, we evaluate if different JDK *vendors* induce differences in the bytecode. Even though most vendors build upon the same OpenJDK source code, there are still some adjustments in regards to e.g. security fixes or performance improvements [1]. This research question aims at determining whether these changes may affect the generated bytecode. To do so, we compiled our full dataset of Java projects using the compilers of the JDK's listed in Table 2. Subsequently we compared all bytecode files generated by the compilers of the investigated vendors, using the same Java and JDK version, against each other.

None of the generated bytecode files contain any difference related to the vendor of the JDK used for compilation. These results indicate that changing the JDK vendor does not influence the bytecode generation of the JDK's compiler.

Based on this result, we consider a single JDK vendor (Oracle) in the remaining research questions.

4.3 RQ2: How does jNorm perform on changing JDK versions?

To investigate jNorm's normalization performance on different JDK versions, we kept all compilation settings at the project's configured default values and only varied the used JDK version within our experimental setup (see Section 4.1). For this experiment we considered versions 7, 8, 11, and 17 of Oracle's JDK.

Table 3 shows the results of our comparisons. The first column shows the pair of JDK versions used to generate the different artifacts we consider (see Section 4.1), e.g., in the first row (sets of) artifacts generated with JDK7 have been compared to (sets of) artifacts obtained from JDK8. The number inside the parentheses indicates the amount of projects we were able to compile with the respective JDKs. As we were not able to compile every project with all JDKs in our experimental setup, the number of compared projects varies based on the successful builds for each JDK. Notice that only few projects could be compiled using JDK7. This is due to features introduced in Java 8 being very popular in modern projects, e.g., default interface functions, streams and lambda expressions. To isolate differences introduced by incremental version increases, we decided to compare a JDK version with the next higher version in our experimental setup. To confirm that we do not miss differences by only comparing incremental version increases, we initially performed a comparison of projects compiled with JDK7 and JDK17 and compared the resulting set to the union of all incremental comparisons. In total we were able to compile ten projects using each version of Oracle's JDK in our experimental setup, comprising 4,621 bytecode files. This analysis showed that the set of differences obtained when comparing JDK7 to JDK17 is equal to the union of the sets of differences obtained when comparing each incremental version increase, i.e., $D_{7\to8} \cup D_{8\to11} \cup D_{11\to17} = D_{7\to17}$ with $D_{i\to j}$ representing the set of files containing compilation differences when comparing bytecode files yielded by the JDK i and JDK j compilers. This comparison holds true for all five processed sets of artifacts (bytecode, plain, optimized, normalized and aggressively normalized Jimple), showing that a comparison of incremental version increases does not miss compilation differences.

Columns two and three show the accumulated results of the textual comparison on a file-level granularity. The remaining columns show the accumulated comparison results on a method-level granularity. Columns "Files" and "Methods" show the total amount of files and methods we managed to compile with the respective JDKs. The "Diffs" columns

Table 3 Normalization results for different JDK versions[2]. Percentage in brackets indicates the share of files/methods with compilation differences.

	Files	Diffs	Methods	Diffs	NLD	Disj.
JDK7 – JDK8 (29)						
Bytecode	8,060	1,058 (13.13%)	48,052	113 (0.24%)	4.02%	4
Jimple	8,069	93 (1.15%)	48,068	113 (0.24%)	5.78%	4
Optimized	8,069	93 (1.15%)	48,068	113 (0.24%)	5.90%	4
Normalized	8,069	24 (0.30%)	45,654	32 (0.07%)	8.13%	0
Aggressive	8,069	24 (0.30%)	45,654	32 (0.07%)	7.46%	0
JDK8 – JDK11 (98)						
Bytecode	45,625	8,068 (17.68%)	417,906	10,253 (2.45%)	9.00%	2,834
Jimple	60,265	2,959 (4.90%)	461,594	5,594 (1.21%)	9.31%	2,832
Optimized	60,265	2,852 (4.89%)	461,594	5,459 (1.18%)	8.28%	2,832
Normalized	60,265	995 (1.65%)	408,250	1,309 (0.32%)	4.80%	8
Aggressive	60,265	392 (0.65%)	408,250	426 (0.10%)	3.16%	8
JDK11 – JDK17 (91)						
Bytecode	58,623	13,936 (23.77%)	469,536	3,146 (0.67%)	18.94%	2,510
Jimple	80,584	2,566 (3.18%)	535,184	3,033 (0.57%)	17.30%	2,501
Optimized	80,584	2,553 (3.17%)	535,184	3,016 (0.56%)	17.06%	2,501
Normalized	80,584	120 (0.15%)	487,280	141 (0.03%)	4.24%	0
Aggressive	80,584	82 (0.10%)	487,280	95 (0.02%)	3.51%	0

show the total amount of files or methods that contained differences within the textual head-to-head comparison and their respective shares. The "NLD" column shows the average NLD of methods that contain differences, indicating the degree of dissimilarity induced by the compilation into individual methods. Note that only methods that are considered as not equal by the textual comparison are considered for the calculation of the NLD. The "Disj." (disjunct) column represents the amount of methods that are present within the file generated by one JDK, but not within the file generated by the other JDK, e.g. synthetically generated bridge-methods. Therefore a direct comparison of such methods is not possible. Note that the transformation of bytecode to Jimple in some cases splits classes into multiple files, thus the number of bytecode files may differ from the number of Jimple files. In cases when dynamically invoked features like e.g. lambda functions are used, they are split into a separate file. We considered the by Soot additionally generated files in our comparison.

One thing that is immediately noticeable is the high amount of differences in bytecode files when considering the file-level granularity, whereas the share of differences is considerably lower at method-level granularity. A detailed investigation into the differing bytecode files revealed this to be due to the presence of nested class information, which in bytecode is contained inside the class, but outside of methods. Depending on the used JDK, different modifiers are used or the order of these definitions varies. One can also observe that by simply converting the bytecode to Jimple, the amount of differences at file-level granularity considerably decreases. Nested class information, in contrast to the bytecode representation, is stored implicitly in the Jimple representation, which causes the disappearance of many dissimilarities. At method-level granularity, the amount of differences between bytecode and Jimple stays fairly similar. This indicates that besides removing some information

[2] Individual project results are available on `https://doi.org/10.5281/zenodo.12625104`

at class level, the plain conversion to Jimple itself does not significantly contribute to the normalization of method-level bytecode. Additionally it can be seen that the optimization step does not significantly contribute to the normalization by itself either. On the contrary, when looking at the results for the normalized file set, the amount of dissimilarities and disjunct methods heavily decreases. The remaining dissimilarities decrease even further when applying an aggressive normalization. Depending on the considered JDKs, the amount of compilation differences at file-level granularity decreases by up to 99% from the textual comparison of plain bytecode to the aggressively normalized Jimple. At method-level granularity the dissimilarity amount decreases by up to 97%.

We investigated the remaining differences in more detail to determine whether we missed other compilation difference classes. We found that the remaining differences are mostly due to more complex cases of compilation difference classes N4 and N8, which target try-catch blocks. Sometimes when such try-catch blocks are nested in specific ways, jNorm fails to apply the corresponding transformation correctly. Other differences are due to incorrect optimizations applied by the Soot framework or an incorrect renaming of local variables.

Since we considered a small set of Java projects to establish the compilation difference classes in the first place and the evaluation across a large dataset of real-world Java projects only revealed a small set of edge cases of the already known difference classes not yet handled by jNorm, we believe that our normalization addresses the most common difference classes appearing within projects compiled with the investigated JDK versions. We will address the transformations incorrectly applied by jNorm in future work.

jNorm can remove up to 99% of the file-level differences and up to 97% of the method-level differences, which are induced when compiling the same source code with different versions of the JDK compiler.

4.4 RQ3: How does jNorm perform on changing Java target levels?

The target level that has been used to compile a specific Java class is typically included in the compiled class in form of a *major version* identifier [33]. While in some cases this information can be used to compile the source code to the version specified within the bytecode files, this is not possible when one wants to directly compare two already compiled bytecode files. Thus it is also important to assess jNorm's performance on differing target levels.

To evaluate jNorm's normalization performance on different Java target levels, we fixed the used JDK version and adjusted the target level in each project's build configuration, within our experimental setup (see Section 4.1). All other build settings have been kept at each project's provided configuration. We consistently used Oracle's JDK11 in our experiment, as it is the most used JDK version in 2022 [44], which also offers backwards compatibility down to target level 6. To adjust the project's target level we scanned each project of our dataset for build files (pom.xml). Inside each of the detected build files, we adjusted the *target*, *release*, or *java.version* properties, which are used to declare the desired Java target level [9, 8], to compile the project to our desired target levels. To validate that all projects were compiled with the intended target level, we verified the target level indicator [33] within the resulting bytecode files and removed it subsequently to not interfere with the comparison.

Table 4 shows the results of our experiment. The table has the same structure as Table 3, besides the first column now representing Java target levels instead of JDK versions. As in the previous experiment, we compare one target level to the next higher target level within our experimental setup, to best isolate compilation differences. To confirm that we do not miss differences by only comparing incremental version increases, we perform a comparison

Table 4 Normalization results for different target levels of the JDK11 compiler[3]. Percentage in brackets indicates the share of files/methods with compilation differences.

	Files	Diffs	Methods	Diffs	NLD	Disj.
T6 – T7 (25)						
Bytecode	13,774	29 (0.21%)	76,102	55 (0.07%)	2.38%	0
Jimple	14,411	29 (0.20%)	77,833	55 (0.07%)	2.36%	0
Optimized	14,411	29 (0.20%)	77,833	55 (0.07%)	2.32%	0
Normalized	14,411	8 (0.06%)	73,102	20 (0.03%)	0.56%	0
Aggressive	14,411	8 (0.06%)	73,102	20 (0.03%)	0.95%	0
T7 – T8 (31)						
Bytecode	4,100	70 (1.71%)	33,109	89 (0.27%)	1.90%	2
Jimple	4,127	33 (0.80%)	33,190	37 (0.11%)	5.05%	2
Optimized	4,127	33 (0.80%)	33,190	37 (0.11%)	4.67%	2
Normalized	4,127	32 (0.78%)	31,338	37 (0.12%)	4.67%	0
Aggressive	4,127	4 (0.10%)	31,338	4 (0.01%)	4.10%	0
T8 – T11 (80)						
Bytecode	28,709	13,260 (46.19%)	293,804	25,301 (8.61%)	18.39%	3,677
Jimple	42,690	9,654 (22.61%)	335,700	25,294 (7.53%)	17.74%	3,677
Optimized	42,690	9,654 (22.61%)	335,700	25,294 (7.53%)	17.58%	3,677
Normalized	42,690	110 (0.26%)	297,541	140 (0.47%)	2.76%	0
Aggressive	42,690	88 (0.21%)	297,541	115 (0.39%)	2.40%	0

of the maximum possible target level distance. Again, the following equation holds for the twelve projects (1,245 bytecode files) that can be compiled to each target level within our experimental setup, when using JDK11 $D_{11.6\to 11.7} \cup D_{11.7\to 11.8} \cup D_{11.8\to 11.11} = D_{11.6\to 11.11}$ with $D_{11.i\to 11.j}$ representing the set of files containing differences when comparing bytecode files yielded by JDK11 set to target levels i and j.

The number within the parentheses inside the first column indicates the amount of projects we were able to compile using JDK11 configured with the respective Java target levels. The total amount of successful builds is lower than the one we obtained in our previous experiment. This is due to the change in the provided build configuration that this experiment requires, which often leads to projects not being able to compile anymore.

One thing that is immediately noticeable in Table 4 is the low amount of compilation differences throughout target levels 6, 7 and 8. Many of these are removed by the (aggressive) normalization. We investigated the remaining differing files and discovered that all remaining differences are due to wrong type inference and incorrect renaming of local variables performed by the Soot framework. The picture changes when considering a target level change from Java 8 to Java 11, as visible in the third row of Table 4. Almost half of the compared files show differences in a textual head-to-head comparison. This value decreases to around 10% when considering method-level granularity. Furthermore, the NLD is very high, indicating that the compiled methods are significantly dissimilar. This large amount of differences is due to many highly used features being affected by the target level increase. From Java target level 8 to 11 the way that string concatenation, private method calls, and inner classes are handled has been changed. These are features that are frequently used within Java

[3] Individual project results are available on `https://doi.org/10.5281/zenodo.12625104`

projects. The conversion to Jimple removes around half of the differences at file-level, but removes only few differences on method-level. The subsequent optimization does not remove any differences in the textual comparison. The normalization, instead, heavily decreases the dissimilarities. After aggressive normalization the amount of dissimilarities decreases by more than 99.3% from the textual comparison of plain bytecode to aggressively normalized Jimple. On a method-level granularity the dissimilarity amount even decreases by 99.6%.

Again we performed a manual inspection of the remaining differing files and methods to uncover possibly missed compilation difference classes. The inspection showed that the remaining differences can mostly be attributed to incorrect optimizations by the SOOT framework and incorrect transformations of dynamic string concatenation and nest-based access control applied by JNORM in specific scenarios (e.g. boolean variables being handled as integer values in the string concatenation). Again, we leave addressing of incorrectly applied transformations by JNORM for future work.

As we already stated in the previous research questions, the absence of previously not uncovered compilation difference classes leads us to believe that we uncovered the most frequent compilation difference classes for projects compiled to the investigated Java target levels. Furthermore, we additionally investigated compilation environments in which we compared changes to the JDK version *and* target level, however, again we did not uncover any further difference classes.

JNORM can remove up to 99.3% of the file-level differences and up to 99.6% of the method-level differences, which are induced when compiling the same source code with different configured Java target levels.

4.5 RQ4: To which degree can bytecode normalization support similarity analysis tools?

In this research question we investigate whether existing similarity analysis tools are already capable of handling compilation differences on their own, without the need of a previous normalization by JNORM. To do so, we used our dataset from the previous research questions, but instead of performing a textual head-to-head comparison, we used the code clone detector NiCad 6.2 [50] to determine its performance without and with normalization applied. NiCad is a flexible and extensible code clone detector that is frequently used in similarity analysis studies. While there are other Java similarity analysis tools available, e.g. CCFinder [35], SourcererCC [52] or JPLAG [47], we decided on NiCad due to it being the most popular similarity analysis tool and its simple extensibility. It supports detection on block- or function-level granularities and different languages (e.g. Java, Rust, C) and applies its own pretty-printing and code normalizations before the similarity analysis. NiCad offers a wide array of possible applications like code clone or plagiarism detection and therefore provides many configurations. We employed NiCad set to function-level granularity with three different configurations (Plagiarism-1, Plagiarism-2 and Default) in our evaluation setup (described in Section 4.1). NiCad does not support Java bytecode nor Jimple out-of-the-box. To perform our experiment, we extended NiCad with the capability to handle Jimple inputs by adjusting and extending the provided Java grammar and normalization/transformation rules. Due to Jimple's syntactic similarity to Java source code, the only adjustments we performed on NiCad consisted of adding Jimple-exclusive statements (identity, invoke, goto, monitor), adjusting try-catch definitions and adjusting if-statements to their Jimple structure.

We decided to perform a plagiarism detection experiment since this is a typical scenario where only bytecode originating from an unknown compilation environment is available, but the corresponding source code is not. To perform the experiment we selected a method pair

Table 5 NiCad performance on method pairs with compilation differences. Percentage indicates the share of method pairs correctly identified as clones.

(a) JDK version comparison.

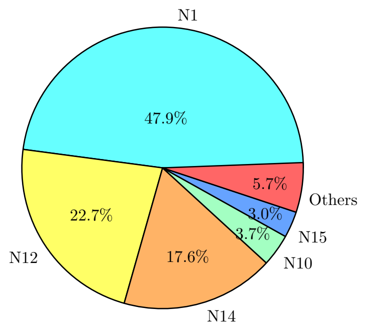
	Plag-1	Plag-2	Default
J7 – J8 (104)			
Jimple	12.5%	75.0%	99.0%
Optimized	14.4%	66.3%	99.0%
Normalized	78.9%	85.6%	99.0%
Agg. Normalized	81.7%	87.5%	100%
J8 – J11 (4,967)			
Jimple	35.2%	57.1%	90.2%
Optimized	38.6%	59.3%	92.1%
Normalized	82.9%	94.0%	99.9%
Agg. Normalized	97.4%	98.6%	99.9%
J11 – J17 (2,858)			
Jimple	6.4%	9.8%	26.1%
Optimized	6.9%	10.0%	25.9%
Normalized	96.7%	99.3%	99.7%
Agg. Normalized	98.0%	99.4%	99.9%

(b) Target level comparison.

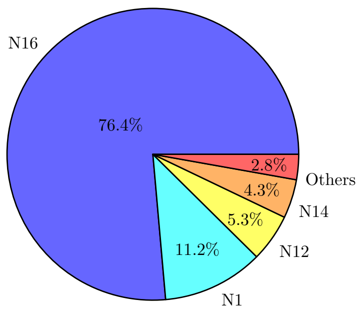
	Plag-1	Plag-2	Default
T6 – T7 (53)			
Jimple	88.7%	94.3%	100%
Optimized	88.7%	94.3%	100%
Normalized	92.5%	98.1%	100%
Agg. Normalized	92.5%	98.1%	100%
T7 – T8 (37)			
Jimple	10.8%	86.5%	100%
Optimized	13.5%	83.8%	100%
Normalized	16.2%	83.8%	100%
Agg. Normalized	91.9%	100%	100%
T8 – T11 (24,095)			
Jimple	24.7%	28.3%	70.8%
Optimized	25.7%	28.0%	70.4%
Normalized	99.7%	99.9%	100%
Agg. Normalized	99.8%	99.9%	100%

that originated from the same source code, but was compiled within different compilation environments, and gave it to NiCad running with different configurations and checked whether it reported the pair as matching or not. Since each method pair originates from the exact same source code, NiCad *should* report it as plagiarism. We considered all methods that were not equal in the textual comparison within our Jimple dataset (see RQ2 4.3 & RQ3 4.4). We executed NiCad on the same set of methods across all other representations (optimized, normalized and aggressively normalized Jimple).

Tables 5a and 5b report the results of our experiment with NiCad. The first column of each table shows the compilation environments that the compared files originated from (J stands for JDK version and T stands for target level) and indicate which normalization steps have been applied before being forwarded to NiCad. The number in parentheses indicates the number of *method pairs* compared by NiCad. Note that, since NiCad was not able to analyze some pairs, the number is slightly lower than the differing methods reported in RQ2 and RQ3. The remaining columns show the detection recall of NiCad using different configurations. The Plagiarism-1 and Plagiarism-2 configurations are explicitly targeting plagiarism detection. The allowed degree of dissimilarity after applying pretty-printing and its own normalizations is specified at 10%. The Plagiarism-2 configuration additionally applies the *blind* renaming scheme, which removes identifier names. The provided Default configuration of NiCad does not target a plagiarism detection scenario but an aggressive code clone detection that allows a dissimilarity of up to 30%. It also applies the additional blind-renaming. Note that we test NiCad in a best-case scenario: First, we provide it with Jimple code, which in contrast to bytecode already contains fewer compilation differences. Second, we evaluate NiCad's performance on small configuration changes only (e.g. comparing JDK7 to JDK8) and do not combine multiple configuration changes, which would result in even more compilation differences. Finally, as most similarity analysis studies [52] we do not take potential false positives into account. Especially with its aggressive Default configuration, which only requires 70% similarity for a match, NiCad would incorrectly classify method pairs as plagiarism cases, which actually do not originate from the same source code.



(a) Normalization.



(b) Aggressive Normalization.

Figure 5 Average prevalence of the individual compilation difference classes.

One can observe that before applying normalization the Plagiarism-1 configuration performs poorly for most JDKs. However, after normalization is applied the performance drastically increases. A similar effect, although not as drastic, can be observed for Plagiarism-2. The aggressive Default configuration of NiCad performs well for compilation environments that do not contain a high degree of dissimilarity. However, for environment changes that actually induce significant differences in methods (J11 - J17 and T8 - T11), indicated by a high NLD between method pairs (see Tables 3 and 4), NiCad continues to perform poorly before normalization, but offers a significantly increased performance when normalizations are applied first via jNorm. Recall, that the Default configuration allows for up to 30% dissimilarity and is still not able to reliably classify method pairs as matching. Note that the provided code did not contain any intentional modifications, e.g. obfuscations, as the removal of such intentional modifications in not part of jNorm's scope. Even without intentional obfuscations, NiCad was not able to detect many of the clones.

NiCad, one of the most popular code clone detectors, is not capable of handling all differences induced by different compilation environments on its own. However, when applying normalization via jNorm first, NiCad's performance increases significantly.

4.6 RQ5: How prevalent are the individual compilation difference transformations of jNorm?

To investigate the prevalence of the compilation difference classes and their individual contribution to the normalization, we tracked each applied transformation within the normalization process of our dataset used throughout RQ2 – RQ4.

Figure 5 shows the average amount of transformations across each JDK and Java version setting. On average jNorm applies 888 transformations during normalization per project. One can see that a few of the established compilation difference classes make up the biggest share of the transformations. For plain normalization, transformation N1, which handles synthetically generated methods, makes up almost half of all transformations. This is due to the large amount of bridge-methods generated for each private method within nested classes. This transformation is followed by transformations N12, which normalizes string concatenations, and N14, which normalizes the invocation of private methods. Both of these are frequently used features within Java applications. The remaining transformations are only sparsely required. For aggressive normalization, transformation N16, which removes all typechecks, makes up by far the biggest share of all transformations. This is due to the compiler frequently placing typechecks into the bytecode. This is further enhanced by our aggressive approach of removing *all* occurrences of typechecks. Since jNorm applies transformation N16 after all other transformations, the prevalence of the other compilation difference classes is the same as for plain normalization. During aggressive normalization 4,134 transformations are applied on average per project.

> Transformations N16 (Insertion or removal of typechecks), N1 (Synthetically generated methods), N12 (Dynamic String concatenation) and N14 (Invocation of private methods) make up 97.2% of all applied transformations and are thus the most prevalent during normalization.

5 Related Work

Many similarity analysis approaches have been proposed, targeting source code, bytecode, or binary code, which typically come with their own set of normalizations.

Bytecode level. Only few approaches have been developed for bytecode similarity analysis. However, there are various scenarios in which Java source code is not available. Whenever this is the case, the comparison has to be performed on the bytecode. SeByte [36] is a similarity detector targeting Java bytecode. It divides the bytecode into tokens and separates them based on their types to employ the Jaccard similarity measure for matching. Baker and Manber [4] leverage a combination of the similarity comparison tools Diff, Siff and Dup to determine the degree of similarity of Java bytecode files. Yu et al. [58] use the Smith-Waterman algorithm to determine the similarity of two bytecode snippets. They extract instruction and method-call sequences from the bytecode and apply the Smith-Waterman algorithm to align the extracted sequences. Ji et al. [30] propose an approach to perform a plagiarism detection on bytecode. They divide the bytecode into sequences and utilize the adaptive local alignment to find potential plagiarisms. Davis and Godfrey [17] propose an approach to find clones that works on Assembler and bytecode. Their approach implements a greedy matching of instruction types and arguments by using an internal weight measure. Chen et al. [7] present an approach that aims at detecting application clones on Android markets. They utilize control flow graphs, to compare apps to each other and find clones in the Dalvik bytecode.

The above mentioned approaches do not explicitly mention how the differences in bytecode resulting from different compilation environments.

Source code level. For source code level similarity analysis many approaches have been developed. NiCad [50, 10] is a textual based code clone detector that targets a variety of programming languages. It uses different means of normalization and is designed to be easily extensible. CCFinder [35] transforms the input source code into a set of tokens and performs the comparisons on this set of tokens. SourcererCC [52] uses a similar token-based detection approach. However, SourcererCC specifically aims at high scalability and is optimized towards a usage on large software repositories. JPlag [47] divides the source code into token strings and applies a greedy string tiling algorithm to find plagiarisms within sets of applications. DECKARD [31] leverages the Abstract Syntax Tree representation of an application's source code to perform the similarity analysis. StoneDetector [2] uses a more specialized code-representation called dominator trees, a concept often used in compilers, to detect structural clones, which use different syntactical constructs to implement the same control flow. DeepSim [60] uses a deep learning model to find semantic similarities within code snippets that are syntactically different. Oreo [51] is another code clone detection tool that leverages deep learning. It uses a pre-trained model that utilizes several code metrics to decide whether two code snippets are clones of each other, even if their syntactical similarity is below 70%.

Source code based similarity analysis approaches have become much more permissive to syntactic differences over the years. This allows some tools to perform a similarity analysis across intermediate representations that are syntactically similar to the targeted source code. Selim et al. [53] investigated how the additional supplementation of the Jimple intermediate representation, alongside the source code, of a Java application can help in code clone detection tasks. To do so, they applied the clone detection tools CCFinder and Simian to Jimple code, which is syntactically similar to Java source code.

Ragkhitwetsagul et al. [49] evaluate and compare 30 different code similarity detection techniques, including code clone detectors, plagiarism detectors and compression tools, within different similarity analysis scenarios.

Binary level. As machine code is usually at the hardware level and there is a lot of variety in compilation environments and optimization levels, binary similarity analysis is a complex problem. David et al. [14, 15, 16] propose multiple approaches that decompose the assembly code of the binary into strands, which encode specific semantic behaviors in small units. Before comparing the units, in a similar way as jNorm and SootDiff do to achieve a more normalized representation, they transform the units to LLVM-IR and apply some optimizations and transformations to them, which are specific to LLVM-IR. Luo et al. [40] model the semantics of binaries with a set of symbolic formulas that represent input-output relations and use a theorem solver to determine their similarity. Hemel et al. [23] created the Binary Analysis Tool which uses different comparison strategies, like string matching, compression and a binary delta check to find software license violations within binaries. Many approaches like SAFE [42] and Xu et al.'s approach [57] use machine learning to determine the similarity of binaries. Marcelli et al. [41] investigate and compare multiple machine learning based approaches that try to classify the similarity of binaries. Haq and Caballerto [21] present a survey of binary code similarity in which they analyze and systemically categorize 70 different binary similarity analysis approaches developed since 1999.

While most binary similarity analysis techniques cannot be directly applied towards bytecode similarity analysis, they can theoretically at least be adapted to it.

Compiler influence There are few works that investigate the relation of compilers to similarity analysis. Kononenko et al. [38] investigate a compilation's degree of influence on code clone detection. To do so, they compare the detected code clones within Java source code and bytecode compiled from the same source code which results in different sets of detected clones. Ragkhitwetsagul and Krinke [48] investigate how compilation and decompilation can influence the clone detection performance within Java code bases. They suggest that decompilation can aid as a complementary measure to source code based clone detection, but is not sufficient on its own. Dann et al. [12] investigate the impact that different compilation environments have on the resulting bytecode. They propose the bytecode comparison tool SootDiff, which employs an approach similar to jNorm, however only support one of the transformations we defined (string constant concatenation) and only considers Java versions 5 to 8. Xiong et al [56] investigate sources of non-determinism in the Java build process that hinder builds from being reproducible. They uncover 14 patterns that may introduce non-equivalences in the build and present corresponding mitigation strategies. In the context of jNorm many of these sources of non-determinism are addressed by the conversion of bytecode to Jimple.

6 Threats to Validity

For our evaluation of jNorm we exclusively relied on projects that use Maven as build tool. While other Java build tools such as Gradle [20] exist, Maven is the most popular [29]. Furthermore, we limited our experiments to Java versions 5–8, 11 and 17. Although this version range covers all LTS-versions (up to August 2023) and are also the by far most frequently used Java versions in projects (see electronic appendix), other versions may yield different evaluation results and cause unidentified compilation differences. While we included a large number of Java projects in our evaluation, there may be some rare instances of differences induced by different compilation environments, which we did not uncover across our data set. Moreover, we tried to isolate the detected compilation difference classes to the specific configuration change they are caused by. There may also be other configuration changes that cause the same differences. However, as jNorm does not know the used configuration anyway, the differences will still be normalized.

Even though jNorm, in its default mode, only transforms constructs from the version produced within one compilation environment to the version produced in another environment, there still may be semantic changes created by the applied transformation. Furthermore, in few cases there are incorrect analyses, transformations and optimizations applied by the Soot framework before the application of jNorm's transformations.

7 Conclusion

In this paper we presented the concept of bytecode normalization for code similarity analysis. Bytecode normalization addresses the problem of comparing the bytecode of Java applications that are compiled in different compilation environments. This is especially necessary when the source code is not available, like in plagiarism, copyright or vulnerability detection and SBOM creation scenarios. By converting bytecode into the intermediate representation Jimple, applying common optimizations, transforming remaining compilation differences and applying naming standardization, we create a representation that is always the same no matter the JDK and Java (LTS) version used to compile the source code. To do so, we identified and presented 16 compilation difference classes, which are induced in the bytecode when using different JDK and Java versions.

Based on the concept of bytecode normalization we implemented jNorm. Our evaluation on a large set of popular real-world Java projects showed that compiling equal Java source code within different compilation environments leads up to 46% of all generated bytecode files containing differences for single incremental version increases. By using jNorm we can reduce the number of compilation differences by more than 99%, for most of the investigated compilation environments. Furthermore, our evaluation of the similarity analyzer NiCad showed that the tool was not able to handle all compilation differences on its own, yet an application of bytecode normalization via jNorm prior to the similarity analysis significantly improved the tool's performance, showcasing the effectiveness of bytecode normalization.

jNorm creates a code representation that is independent of the environment the Java source code has been compiled in and thus lowers the required complexity for subsequent similarity analysis. This could potentially pave the way for further research in the field of bytecode similarity analysis, bringing it closer to the wide range of tools and techniques currently available for analyzing source code similarity.

References

1 Amazon Corretto 8. Accessed 2023-03-31. URL: `https://docs.aws.amazon.com/corretto/latest/corretto-8-ug/what-is-corretto-8.html`.

2 Wolfram Amme, Thomas S. Heinze, and André Schäfer. You look so different: Finding structural clones and subclones in java source code. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2021, Luxembourg, September 27 - October 1, 2021*, pages 70–80. IEEE, 2021.

3 ASM: Java bytecode manipulation and analysis framework. Accessed 2022-10-24. URL: `https://asm.ow2.io/`.

4 Brenda S. Baker and Udi Manber. Deducing similarities in java sources from bytecodes. In *1998 USENIX Annual Technical Conference, New Orleans, Louisiana, USA, June 15-19, 1998*. USENIX Association, 1998.

5 Musard Balliu, Benoit Baudry, Sofia Bobadilla, Mathias Ekstedt, Martin Monperrus, Javier Ron, Aman Sharma, Gabriel Skoglund, César Soto-Valero, and Martin Wittlinger. Challenges of producing software bill of materials for java. *IEEE Security & Privacy*, pages 2–13, 2023.

6 Executive Order on Improving the Nation's Cybersecurity. Accessed 2023-09-12. URL: `https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity`.

7 Kai Chen, Peng Liu, and Yingjun Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, pages 175–186. ACM, 2014.

8 Apache Maven Compiler Plugin - Setting the –release of the Java Compiler. Accessed 2023-04-03. URL: `https://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-release.html`.

9 Apache Maven Compiler Plugin - Setting the -source and -target of the Java Compiler. Accessed 2023-04-03. URL: `https://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-source-and-target.html`.

10 James R. Cordy and Chanchal K. Roy. The nicad clone detector. In *The 19th IEEE International Conference on Program Comprehension, ICPC 2011, Kingston, ON, Canada, June 22-24, 2011*, pages 219–220. IEEE Computer Society, 2011.

11 Cyber Resilience Act. Accessed 2023-09-12. URL: `https://digital-strategy.ec.europa.eu/en/library/cyber-resilience-act`.

12 Andreas Dann, Ben Hermann, and Eric Bodden. Sootdiff: bytecode comparison across different java compilers. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis, SOAP@PLDI 2019, Phoenix, AZ, USA, June 22, 2019*, pages 14–19. ACM, 2019.

13 Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. Identifying challenges for oss vulnerability scanners-a study & test suite. *IEEE Transactions on Software Engineering*, 48(9):3613–3625, 2021.

14 Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. *Acm Sigplan Notices*, 51(6):266–280, 2016.

15 Yaniv David, Nimrod Partush, and Eran Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation*, pages 79–94, 2017.

16 Yaniv David, Nimrod Partush, and Eran Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices*, 53(2):392–404, 2018.

17 Ian J. Davis and Michael W. Godfrey. From whence it came: Detecting source code clones by analyzing assembler. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 242–246. IEEE Computer Society, 2010.

18 JDK-6246854 : Unnecessary checkcast in generated code. Accessed 2022-10-28. URL: `https://bugs.java.com/bugdatabase/view_bug.do?bug_id=6246854`.

19 GNU Compiler for Java (GCJ). Accessed 2022-10-17. URL: https://gcc.gnu.org/wiki/GCJ.

20 Gradle Build Tool. Accessed 2022-11-07. URL: https://gradle.org/.

21 Irfan Ul Haq and Juan Caballero. A survey of binary code similarity. *ACM Comput. Surv.*, 54(3):51:1–51:38, 2022.

22 Foyzul Hassan, Shaikh Mostafa, Edmund S. L. Lam, and Xiaoyin Wang. Automatic building of java projects in software repositories: A study on feasibility and challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, pages 38–47. IEEE Computer Society, 2017.

23 Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 63–72, 2011.

24 The Java HotSpot Performance Engine Architecture. Accessed 2022-10-14. URL: https://www.oracle.com/java/technologies/whitepaper.html.

25 JEP 280: Indify String Concatenation. Accessed 2022-10-27. URL: https://openjdk.org/jeps/280.

26 Oracle Java SE 6 and JRockit End of Support. Accessed 2022-12-12. URL: https://support.oracle.com/knowledge/Middleware/2244851_1.html.

27 JDK Release Notes. Accessed 2023-03-30. URL: https://www.oracle.com/java/technologies/javase/jdk-relnotes-index.html.

28 Eclipse Java development tools (JDT). Accessed 2022-10-17. URL: https://www.eclipse.org/jdt/core/.

29 The State of Developer Ecosystem 2023. Accessed 2023-12-15. URL: https://www.jetbrains.com/lp/devecosystem-2023/java/.

30 Jeong-Hoon Ji, Gyun Woo, and Hwan-Gue Cho. A plagiarism detection technique for java program using bytecode analysis. In *2008 third international conference on convergence and hybrid information technology*, volume 1, pages 1092–1098. IEEE, 2008.

31 Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 96–105. IEEE Computer Society, 2007.

32 IBM Jikes Compiler for the Java Language. Accessed 2022-10-17. URL: https://sourceforge.net/projects/jikes/.

33 The ClassFile Structure. Accessed 2023-12-12. URL: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.1.

34 Oracle JVM Specification - Chapter 4. The class File Format. Accessed 2023-04-03. URL: https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.7.8.

35 Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.

36 Iman Keivanloo, Chanchal Kumar Roy, and Juergen Rilling. Sebyte: Scalable clone and similarity search for bytecode. *Sci. Comput. Program.*, 95:426–444, 2014.

37 Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 595–614. IEEE Computer Society, 2017.

38 Oleksii Kononenko, Cheng Zhang, and Michael W. Godfrey. Compiling clones: What happens? In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 481–485. IEEE Computer Society, 2014.

39 Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eighth Working Conference on Reverse Engineering, WCRE'01, Stuttgart, Germany, October 2-5, 2001*, pages 301–309. IEEE Computer Society, 2001.

40 Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400, 2014.

41 Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2099–2116, 2022.

42 Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329. Springer, 2019.

43 JEP 181: Nest-Based Access Control. Accessed 2022-10-28. URL: `https://openjdk.org/jeps/181`.

44 2022 State of the Java Ecosystem Report. Accessed 2022-10-24. URL: `https://newrelic.com/resources/report/2022-state-of-java-ecosystem`.

45 The Java programming language Compiler Group. Accessed 2022-10-17. URL: `https://openjdk.org/groups/compiler/`.

46 The Java Language Environment - Chapter 4: Architecture Neutral, Portable, and Robust. Accessed 2022-10-17. URL: `https://www.oracle.com/java/technologies/architecture-neutral-portable-robust.html`.

47 Lutz Prechelt, Guido Malpohl, Michael Philippsen, et al. Finding plagiarisms among a set of programs with jplag. *J. Univers. Comput. Sci.*, 8(11):1016, 2002.

48 Chaiyong Ragkhitwetsagul and Jens Krinke. Using compilation/decompilation to enhance clone detection. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, pages 1–7. IEEE, 2017.

49 Chaiyong Ragkhitwetsagul, Jens Krinke, and David Clark. A comparison of code similarity analysers. *Empir. Softw. Eng.*, 23(4):2464–2519, 2018.

50 Chanchal Kumar Roy and James R. Cordy. NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *The 16th IEEE International Conference on Program Comprehension, ICPC 2008, Amsterdam, The Netherlands, June 10-13, 2008*, pages 172–181. IEEE Computer Society, 2008.

51 Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: detection of clones in the twilight zone. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 354–365. ACM, 2018.

52 Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. Sourcerercc: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1157–1168. ACM, 2016.

53 Gehan M. K. Selim, King Chun Foo, and Ying Zou. Enhancing source-based clone detection using intermediate representation. In *17th Working Conference on Reverse Engineering, WCRE 2010, 13-16 October 2010, Beverly, MA, USA*, pages 227–236. IEEE Computer Society, 2010.

54 Soot Options and Phases. Accessed 2022-10-17. URL: `https://soot-oss.github.io/soot/docs/4.3.0/options/soot_options.html`.

55 Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, and Vijay Sundaresan. Soot – A java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research, November 8-11, 1999, Mississauga, Ontario, Canada*, page 13. IBM, 1999.

56 Jiawen Xiong, Yong Shi, Boyuan Chen, Filipe R Cogo, and Zhen Ming Jiang. Towards build verifiability for java-based systems. In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 297–306, 2022.

57 Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 363–376, 2017.

58 Dongjin Yu, Jiazha Yang, Xin Chen, and Jie Chen. Detecting java code clones based on bytecode sequence alignment. *IEEE Access*, 7:22421–22433, 2019.

59 Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.

60 Gang Zhao and Jeff Huang. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, pages 141–151. ACM, 2018.