

# Understanding Compiler Effects on Clone Detection Process

Lerina Aversano<sup>1</sup><sup>a</sup>, Mario Luca Bernardi<sup>1</sup><sup>b</sup>, Marta Cimitile<sup>2</sup><sup>c</sup>,  
Martina Iammarino<sup>1</sup><sup>d</sup> and Debora Montano<sup>3</sup><sup>e</sup>

<sup>1</sup>University of Sannio, Department of Engineering, Benevento, Italy

<sup>2</sup>Unitelma Sapienza University, Rome, Italy

<sup>3</sup>Universitas Mercatorum, Rome, Italy

**Keywords:** Software Maintainability, Software Quality Metrics, Clones Detection, Decompiled Source-Code.


**Abstract:** Copying and pasting code snippets, with or without intent, is a very common activity in software development. These have both positive and negative aspects because they save time, but cause an increase in costs for software maintenance. However, often the copied code changes due to bug fixes or refactorings, which could affect clone detection. In this regard, this study aims to investigate whether the transformations performed by the compiler on the code can determine the appearance of a set of previously undetectable clones. The proposed approach involves the extraction of software quality metrics on both decompiled and source code to bring to light any differences due to the presence of undetectable clones on the source code. Experiments were conducted on five open-source Java software systems. The results show that indeed compiler optimizations lead to the appearance of a set of previously undetected clones, which can be called logical clones. This phenomenon in Java appears to be marginal as it amounts to 5% more clones than normal, therefore a statistically negligible result in small projects, but in the future, it would be interesting to extend the study to other programming languages to evaluate any different cases.


## 1 INTRODUCTION


Software systems are constantly evolving over time, so these changes lead to the vulnerability of their architecture, which can be subject to numerous design problems that need to be managed. Among these are clones, i.e. sequences of instructions duplicated in several points of the same source code file, or in different files (Rattan et al., 2013). Several studies show that about 5-20% of a software system may contain duplicate code, so in recent years, the detection of clones in software systems has become a popular topic of study among researchers, because their presence in the source code can be considered detrimental to the quality of the system. In this regard, the idea that it is of fundamental importance to be aware of the presence of clones is increasingly widespread, in or-


der to improve the refactoring and maintenance of the source code. Over time, several techniques and tools for detecting clones have been developed, which can be classified according to the input they receive, the representation, and the algorithms they use (Nadim et al., 2022). Broadly speaking these techniques are classified as text-based, token-based, metric-based, Abstract Syntax Tree (AST), Program Dependency Graph (PDG)-based, and hybrid as shown (Saini et al., 2018). Among the most widely used tools are NICAD<sup>1</sup>, a scalable, flexible, and easy-to-use clone detection tool that can be easily incorporated into IDEs and other environments (Cordy and Roy, 2011).


The objective of this study is to understand if the transformations performed by the compiler could determine the appearance of a set of clones not previously detectable with the tools available. Some examples of typical optimizations performed by compilers can control flow reordering, inlining of constants and routines, precalculation of fixed expressions, and loop unrolling. Traditionally, as far as the Java language

<sup>a</sup> <https://orcid.org/0000-0003-2436-6835>

<sup>b</sup> <https://orcid.org/0000-0002-3223-7032>

<sup>c</sup> <https://orcid.org/0000-0003-2403-8313>

<sup>d</sup> <https://orcid.org/0000-0001-8025-733X>

<sup>e</sup> <https://orcid.org/0000-0002-5598-0822>

<sup>1</sup><https://www.txl.ca>

is concerned, most optimizations are done by the JIT compiler, at runtime, but unlike some suggest javac also has a role in optimizing the code. However, there are no tools for finding clones that work at the byte-code level nor would they have particular relevance, since the results would have to be compared with the source code. Therefore, we have used an open-source decompiler tool to convert the object files into text code in order to be able to detect clones in a manner comparable to the source code. The rest of the document is structured as follows: Section 2 reports the studies related to our study, and in Section 3 we detail the approach used. Section 4 describes the results obtained and finally, Section 5 reports the conclusions and future work.

## 2 RELATED WORKS

Given the prevalence of clones in the source code of software systems, their detection and management have become hotly debated topics in software engineering, as well as smells (Aversano et al., 2020a; Aversano et al., 2020b). The search for duplicate code allows a lot of information about the software system. Applications include software licensing issues (German et al., 2009), source code provenance (Davies et al., 2013; Aversano et al., 2008), and software plagiarism detection (Prechelt et al., 2002). There are several tools for source code and binary code clone detection. The Longest Common Subsequence (LCS) found in NiCad (Cordy and Roy, 2011) is one of many used techniques based on string comparison methods. Several technologies convert source code into an intermediate form, such as tokens, and perform similarity analysis on the, e.g. SourcererCC (Sajjani et al., 2016), or CCFinder (Kamiya et al., 2002). Using tokens can be thought of as a code normalization that improves the similarity measurement of two code snippets by changing their representation. Another form of normalization can be decompilation, which converts a program from a low-level language to a high-level language. To date, numerous studies have focused on detection before and after decompilation. By converting Java and C/C++ code to assembler code and using the longest common subsequence string match along with an upward search for flexible matching, (Davis and Godfrey, 2010) can identify clones. Simian and CCFinderX are improved by (Selim et al., 2010) by converting Java code into Jimple code and finding clones at that level. Their approach makes it easier for tools to find type 3 clones and deal with gap clones. (Ragkhitwetsagul and Krinke, 2017) discovered that the decompiled clones are shorter and

clearer than the original code. Their method offers the chance to examine and compare clones before and after decompilation, which offers several insightful observations.

## 3 APPROACH

Figure 1 shows the general architecture of the proposed approach, which consists of three main phases. The first phase involves the extraction from the Maven repository of the open-source software systems considered for analysis. For each of these, both the JARs of the releases and the source code have been downloaded. These are used as input for the second phase, in which respectively the release jars, are decompiled using the QUILTFLOWER tool which outputs the decompiled Java code, and the source code is manipulated for the extraction of clones and quality metrics through the NICAD and CK tools. After processing, the source and decompiled codebase clones are output in XML and CSV format and parsed to extract the results. By nature, the approach used can be generalized to any codebase in a relatively simple way, as the entire process has been automated through scripts written in C#. In the following paragraphs the selected open-source systems, the detection and matching of the clones, and finally the extraction of the quality metrics will be better detailed.

### 3.1 Open Source Software Systems

To conduct the experiments, five Java open-source software systems available on the Maven Central Repository were selected, having at least five stable versions. Table 1 shows the system name in the first column, the number of releases in the second, and the specific releases considered in the third. For each system, instead of considering the entire codebase of each project, it was decided to use only the common parts between the source and the binaries. It would not make sense to include artifacts related to, for example, test cases in the comparison as these are not present in the release JAR files.

### 3.2 Decompilation

Once obtained, the executables are decompiled using Quiltflower, a modern Java decompiler focused on improving the quality, speed, and usability of the code. Quiltflower<sup>2</sup> is an open-source project available on GitHub. It is a fork of Fernflower and Forge-

<sup>2</sup><https://github.com/QuiltMC/quiltflower>

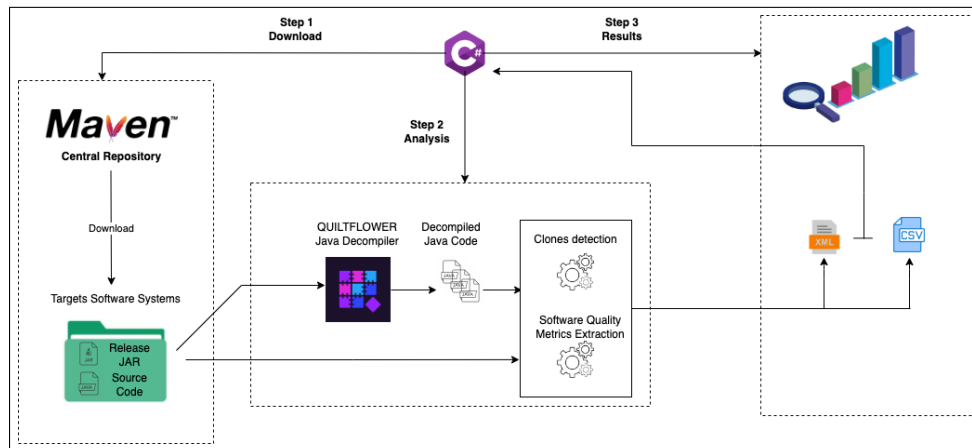


Figure 1: Architecture of the Proposed Approach.

Table 1: Open-source Software systems considered.

Name	#Release	#Considered Release
atlanmod/commons	7	1.0.2, 1.0.3, 1.0.4, 1.0.5, 1.0.6, 1.1.0, 1.1.1
commons-dbutils	7	1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7
Java-WebSocket	13	1.3.0, 1.3.4, 1.3.5, 1.3.6, 1.3.7, 1.3.8, 1.3.9, 1.4.0, 1.4.1, 1.5.0, 1.5.1, 1.5.2, 1.5.3
log4j-core	20	2.11.1, 2.11.2, 2.12.0, 2.12.1, 2.12.2, 2.12.3, 2.12.4, 2.13.0, 2.13.1, 2.13.2, 2.13.3, 2.14.0, 2.14.1, 2.15.0, 2.16.0, 2.17.0, 2.17.1, 2.17.2, 2.18.0, 2.19.0
org.jfree.fxgraphics2d	5	2.0, 2.1, 2.1.1, 2.1.2, 2.1.3

flower and changes include new language features, better control flow generation, more configurability, multithreading, and optimization.

### 3.3 Clones Detection

The purpose of this study is to find out if there are *logical clones*, a term used to refer to clones that are not immediately visible in the source, but appear when the compiler optimizes the code by simplifying its structure and potentially rearranging the statements. This type of comparison must be done on blind-type clones, as the decompiled code no longer has the names of the variables and constants, making a simple text comparison useless. As can be seen from the example shown in Figure 2, the optimized code can be significantly different. In this example from Log4j-core, the compiler has removed an internal loop simplifying the control flow.

Therefore, the NICAD tool, a TXL-based hybrid language-aware text-comparison software clone detection system, was first used to perform the clone detection. In input, it requires one or more source directories to check for clones and a configuration file that specifies the normalization and filtering to be performed and provides the output results in both XML format for easy parsing and HTML format for convenient navigation. It allows parsing of a variety of languages, including C, Java, Python, and C#, and provides a range of normalizations, filters, and abstractions. It is designed to be easily extensible using a

component-based plug-in architecture. Specifically, version 6.2 was used, with the type3-2 configuration, which compares functions in blind format, i.e. without identifiers (types, functions, and variables). The output of NICAD has foreseen: an XML file containing the single functions extracted from the codebase, identified by file and starting and ending line (i); an XML file containing the single functions in blind format, in the same way as the previous one, identified by initial and final files and lines (ii); a summary XML file containing the list of functions that have clones. These features are grouped into clone classes: all features within the same group are clones or have a high similarity coefficient. Once the analysis results are obtained, the classes of homologous clones between the source and the decompiled are compared to verify if clones have appeared that were not present in the analysis of the source.

### 3.4 Clones Matching

Given a class of similarity in the source, it is difficult to find the decompile equivalent reliably. The process of identifying the functions is not straightforward as in the blind results neither the name nor the line numbers can be relied on as they are different between the source and the decompiled. To solve this problem it is necessary to cross-reference the files generated by NICAD taking into account both the blind versions and the versions with identifiers. One must then compute a data structure that maps the filename and tuple

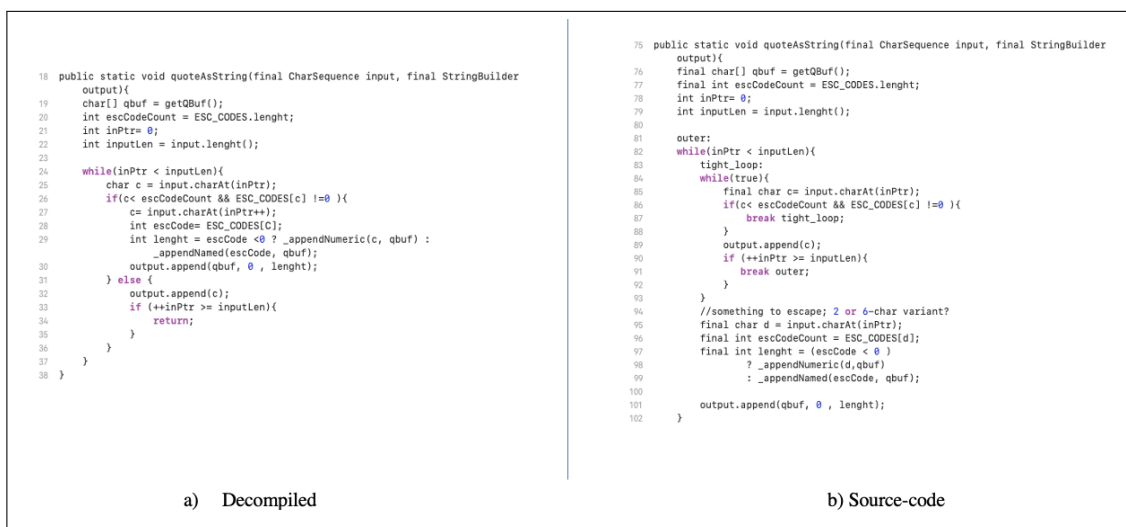


Figure 2: Comparison example of the same function extracted from Log4j-core.

(start line, end line) of a function in the source to the file and tuple (start line, end line) of the decompiled and vice versa. In this regard, the steps of the approach can be summarized as follows:

- The XML files containing the non-blind functions of the source and the decompiled are parsed, using XPath3 and RegEx4. Each function is identified by the file path and its complete signature (to discriminate against any overloads), creating a data structure that takes the name of *NicadFunctionsKey* and represents the key of a HashMap whose value is the tuple (initial row, final row) called *NicadFunctionsValue*.
- The *NicadFunctionsKey* in common between the two data structures loaded into memory are then found. The result is a list of *MatchedFunctions* containing the *NicadFunctionsKey* and the *NicadFunctionsValue* of the source and the decompiled.
- The blind-clones-classes XML files are parsed inside two *NicadCloneClass* lists, one for the source and one for the decompiled (cloning classes).
- The homologous IDs of the clone classes are calculated by finding which ones contain the same functions from both sides. More specifically, when new clone classes appear in the decompile that didn't exist in the source, they are discarded.
- For each pair of *NicadCloneClass* (Source, Decompiled) identified, its content is compared

### 3.5 Software Quality Metrics Extraction

To verify the internal quality of the software systems considered, the source code extracted for each of them was subjected to analysis for the extraction of quality metrics (Iammarino et al., 2019; Ardimento et al., 2021). They provide an evaluation of the long-term maintainability of the various characteristics of the system (Li et al., 2015; Alves et al., 2016). Specifically, code cohesion, complexity, size, and coupling are evaluated using the metrics previously specified by Chidaber and Kemerer (Chidamber and Kemerer, 1994). Specifically, the quality indicators were extracted with the CK tool, a static code analysis tool, available on GitHub, which allows you to calculate class and method level metrics in Java projects without the need for compiled code. Precisely, the quality indicators were extracted with the CK<sup>3</sup> tool, a static code analysis tool, available on GitHub, which allows you to calculate class and method-level metrics in Java projects without the need for compiled code.

Therefore, Table 2 shows the extracted metrics, reporting the name in the first column and a brief description in the second.

## 4 RESULTS

Following a quantitative analysis, analyzing the evolution of the source code metrics between the different releases, it emerged that the metrics that tend to grow

<sup>3</sup><https://github.com/mauricioaniche/ck>

Table 2: Software Quality Metrics.

Name	Description
Response For a Class (RFC)	Indicator of the 'volume' of interaction between classes.
Depth of Inheritance Tree (DIT)	Maximum distance of a node (a class) from the root of the tree representing the hereditary structure.
Weight Method Count per Class(WMC)	Weighted sum of the methods of a class.
Lack of Cohesion of Methods(LCOM)	The cohesion of a method expresses the property of a method to exclusively access attributes of the class.
Coupling Between Objects (CBO)	Number of collaborations of a class, that is, the number of other classes to which it is coupled
Number of unique words	Count of unique words.
Non-Commented, non-empty Lines of Code	Number of lines of code, except of blank lines.
Number of static invocation	Total number of invocations through static methods
Number of methods	Total amount of methods: static, public, abstract, private, protected, predefined, final, and synchronized.
Number of fields	Number of set of fields: static, public, private, protected, default, final, and synchronized.
Usage of each field	Calculate the usage of each field in each method.
Usage of each variable	Calculate the usage of each variable in each method.
Comparisons	Total of comparisons (e.g. == or !=).
Returns	Count of return statements.
Try/catches	Total of try and catches.
Loops	Amount of loops (for, while, do while, generalized for).
Variables	Numerical index of variables declared.
Number	Quantity of numbers (i.e. int, long, double, float).
Math Operations	Count of mathematical operations.
Parenthesized expressions	Count of expressions in parentheses.
Anonymous classes, subclasses and lambda expressions	Number of anonymous declarations of classes or subclasses.
String literals	Amount of string literals (e.g. "John Doe"). Strings that are repeated are counted as many times as they appear.
Modifiers	Number of public / abstract / private / protected / native modifiers of classes / methods.
Max nested blocks	The highest number of blocks nested together.

the most in value between the releases are LCOM, LOC, and Unique Words Quantity, except when interventions are made maintenance or refactoring. To a lesser extent in projects, as far as methods are concerned, modifiers also tend to grow, as one would expect as the size of the codebases increases. The exception is Java-WebSocket, where modifiers decrease over time, especially between releases 1.3.4 and 1.3.5. As for CBO, DIT, RFC, TCC, LCC, and NOSI, they generally tend not to change much between releases on average. However, it is possible to notice a minimal increase for Log4j, a fact that can indicate how Log4j is a fairly well-managed project, as it has a high number of releases, but these metrics have increased slightly between them.

Furthermore, comparing the results obtained on the source code and those obtained on the decompiled it emerged that some metrics are always different between the various projects. Specifically, the max Nested Blocks Quantity metric indicates that there are no nested blocks in the decompiled source code because the control flow was changed by the compiler. The Numbers Quantity metric indicates that the symbolic constants are absent in the decompiled data because they are replaced by the literals by the compiler. Again, the Return Quantity and LOC metrics indicate a decrease in return statements and lines of code as a result of the compiler sorting and optimizing the statements. Finally, other syntax-dependent factors such as Parenthesized Expression Quantity are not taken into consideration because the compiler may remove parentheses inserted by programmers to aid readability. Optimizations of a compiler are limited to changes not visible from the outside. These can be the elision of local variables, restructuring of control flow, or in extreme cases elimination of private methods. The compiler cannot rearrange the class hierarchy or the order and number of method calls, so,

as would be expected, these metrics are equal within a margin of error.

Figure 3 shows the comparison of the presence of the clones in the source code, shown with the blue bars, and the decompiled one represented by the orange bars. For each system considered in the analysis, the respective graph is shown, which shows the number of clones on the ordinate axis and the various releases considered on the abscissa axis. The results show that in general there was an increase in the absolute number of clones between the source and the decompiled. This effect is particularly noticeable in larger projects. Furthermore, by analyzing the results produced on a sample basis, it is observed that most of the new clones mainly consist of changes to the control flow or the member access syntax, for example through the use of the keyword *this*. It is therefore observed that NiCAD does not correctly handle the aliasing of names, not detecting potential clones where the only difference is how a given variable, method, or class name.

A specific example is shown in Figure 4, where the source code is shown on the left, and the same code is subjected to decompilation on the right. Regarding the variations of the control flow instead, we observe the reordering of some blocks and the introduction of keywords not previously present, this does not change the meaning of the code but raises a possible discussion: the Java compiler is not particularly aggressive and these differences could even just depend on the interpretation of the bytecode and therefore be an artifact of the source reconstruction process. Thus, the process of compiling and subsequently decompiling could be viewed conceptually as a preprocessing step to normalize the code by removing this variability, as a result, most of these clones could be detectable using a better preprocessing algorithm in NiCAD.

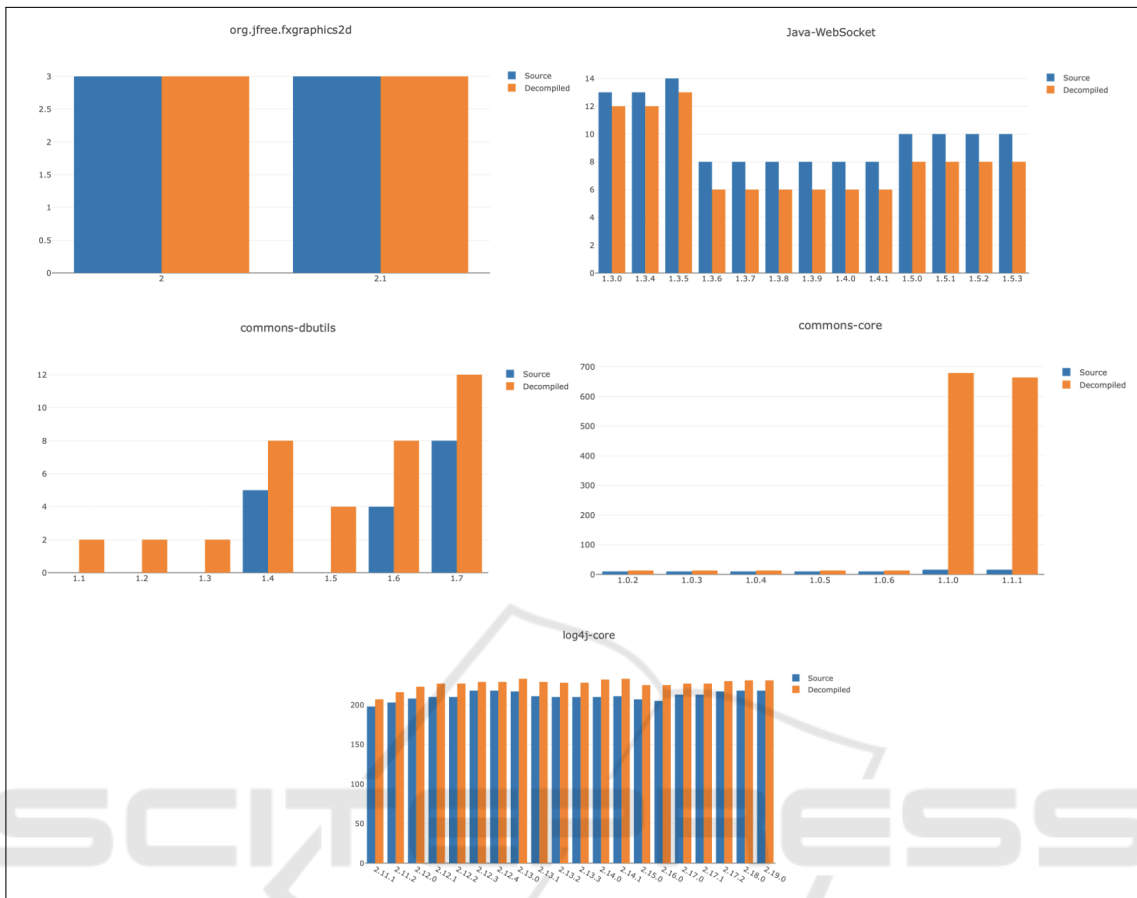


Figure 3: The presence of the clones compared on the source code and the decompiled.

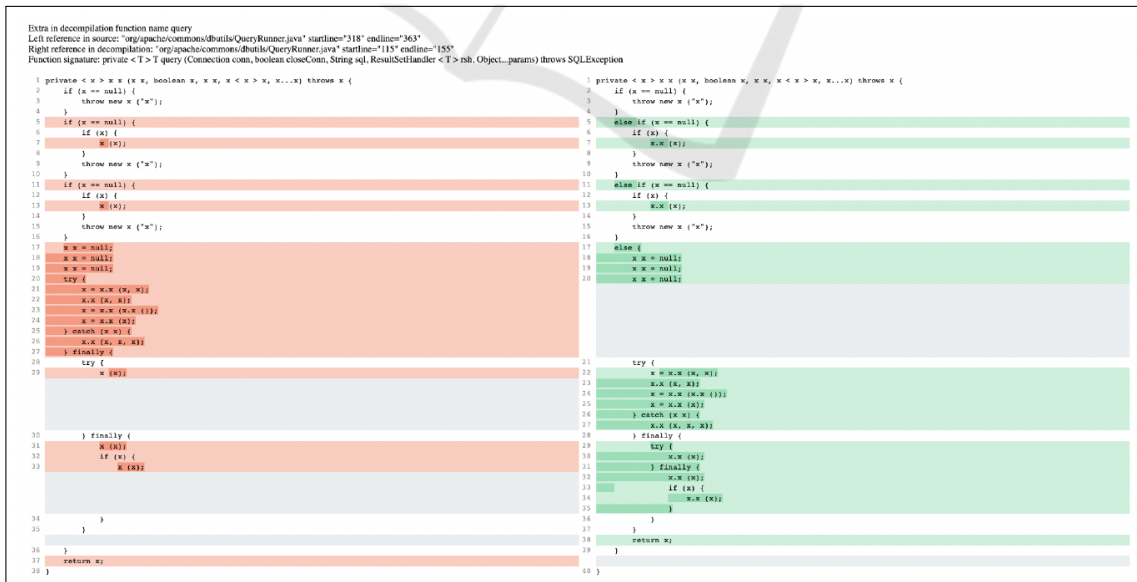


Figure 4: Difference between source (left) and decompiled (right) related to commons-dbutils release 1.4.

Finally, the graphs of the measure of the differences found between the clones in the source code

and the decompiled are shown in Figure 5. Specifically, the relative graph is shown for each system,

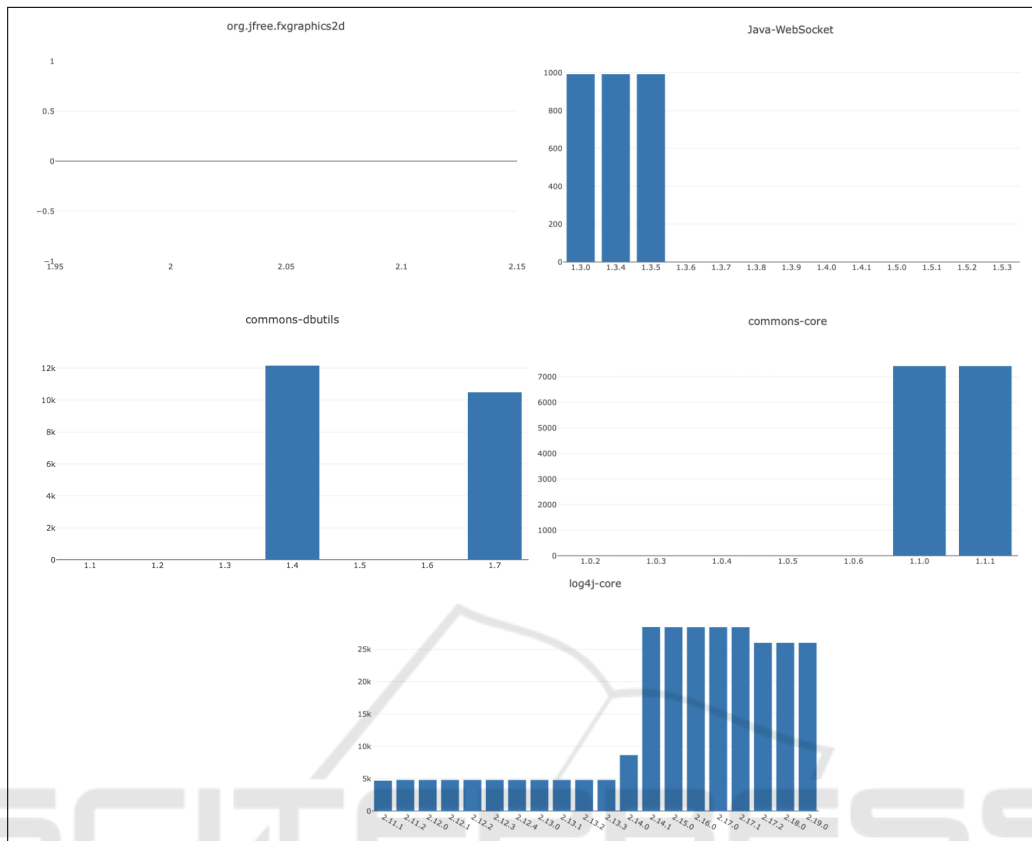


Figure 5: Trend of the differences in the presence of clones in the considered systems.

reporting the number of clones on the y-axis and the releases considered on the x-axis. The graphs show certain points in time where the value of the difference changes significantly. The clones present on both sides have been ignored, consequently, a change to this extent implies that, with the same clones, there are more variations compared to the previous version. Since the number of clones remains more or less unchanged at some point in the history of the project, something was changed causing this effect. The case of org.jfree.fxgraphics2d is special because it shows that the difference is always 0. In JavaWebSocket in the first three releases the same difference is detected and then cancels out in the following ones, on the contrary for Commons-core the difference at first zero, grows in the penultimate release to remain stable. In log4j this difference is mostly stable, then increases.

The particular case is that of commons-dbutils which shows that at one point there is a high variation in the 1.4 release, then disappearing in subsequent releases and finally returning in the current release, 1.7. This behavior is believed to be due to the version and/or options of the JDK. To validate this hypothesis, the GIT history of the projects where the phenomenon occurs was analyzed. There was a

correlation between changes to the POM file and the observed results. In different commits between different versions, the properties relating to the target and source javac version, JDK version, Maven version, and Maven Java compiler plugin parameters are changed, so it was not possible to identify a single determining factor. To give a satisfactory explanation to these observations, further data or experiments are needed by modifying only the properties in question.

## 5 CONCLUSIONS

Finding clones, exact or identical pieces of code, within or between software systems is the goal of clone detection. This study proposes an approach to compare the presence of clones in source code and decompiled code. Decompilation can be considered a code normalization technique because it includes some syntactic changes made to the Java source code.

In five open-source software systems, we investigate decompilation as a preprocessing step for clone identification. The results show that indeed compiler optimizations lead to the appearance of

many previously undetected clones, which have been termed "logical clones". This phenomenon in Java is marginal as it amounts to 5% more clones than in regular Java, statistically negligible in small projects. Furthermore, the trend of software quality metrics in the presence of clones was also studied and it emerged that some metrics differ between the various projects.

Among the problems encountered, it should be considered that for the detection of clones, the textual output of the decompiler is compared, and this poses a problem as the results will be different based on the chosen decompiler and the configuration and version of the decompiler itself. An example of this effect is the presence of the dot operator for name resolution: the decompiler always imports the class using the keyword *import* while some projects prefer to use the dot operator. This created ambiguity and made it impossible to match some methods, but at the same time, it allowed to detection of clones that NiCAD does not consider as such because it does not differentiate between the method/property access point operator and the access point operator a class in a package. A deeper analysis should consider different decompilers or work directly at the bytecode level of the JVM to detect repeating instruction patterns. It should be noted that at the moment there are no mature tools capable of working on Java bytecode at this level. A possible development could be the extension of the project to other languages. At present, NiCAD supports C, C#, and Python in addition to Java. However, the CK tool only supports Java.

## REFERENCES

- Alves, N. S., Mendes, T. S., de Mendonça, M. G., Spínola, R. O., Shull, F., and Seaman, C. (2016). Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121.
- Ardimento, P., Aversano, L., Bernardi, M. L., Cimitile, M., and Iammarino, M. (2021). Temporal convolutional networks for just-in-time design smells prediction using fine-grained software metrics. *Neurocomputing*, 463:454–471.
- Aversano, L., Bernardi, M. L., Cimitile, M., Iammarino, M., and Romanyuk, K. (2020a). Investigating on the relationships between design smells removals and refactorings. In *International Conference on Software and Data Technologies*.
- Aversano, L., Carpenito, U., and Iammarino, M. (2020b). An empirical study on the evolution of design smells. *Inf.*, 11:348.
- Aversano, L., Cerulo, L., and Palumbo, C. (2008). Mining candidate web services from legacy code. page 37 – 40.
- Chidamber, S. R. and Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493.
- Cordy, J. R. and Roy, C. K. (2011). The nicad clone detector. In *2011 IEEE 19th International Conference on Program Comprehension*, pages 219–220. IEEE.
- Davies, J., German, D. M., Godfrey, M. W., and Hindle, A. (2013). Software bertillonage: Determining the provenance of software development artifacts. *Empirical Software Engineering*, 18:1195–1237.
- Davis, I. J. and Godfrey, M. W. (2010). From whence it came: Detecting source code clones by analyzing assembler. In *2010 17th Working Conference on Reverse Engineering*, pages 242–246. IEEE.
- German, D. M., Di Penta, M., Gueheneuc, Y.-G., and Antoniol, G. (2009). Code siblings: Technical and legal implications of copying code between applications. In *2009 6th IEEE International Working Conference on Mining Software Repositories*, pages 81–90. IEEE.
- Iammarino, M., Zampetti, F., Aversano, L., and Di Penta, M. (2019). Self-admitted technical debt removal and refactoring actions: Co-occurrence or more? page 186 – 190.
- Kamiya, T., Kusumoto, S., and Inoue, K. (2002). Cfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE transactions on software engineering*, 28(7):654–670.
- Li, Z., Avgeriou, P., and Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220.
- Nadim, M., Mondal, M., Roy, C. K., and Schneider, K. A. (2022). Evaluating the performance of clone detection tools in detecting cloned co-change candidates. *Journal of Systems and Software*, 187:111229.
- Prechelt, L., Malpohl, G., Philippsen, M., et al. (2002). Finding plagiarisms among a set of programs with jplag. *J. Univers. Comput. Sci.*, 8(11):1016.
- Ragkhitwetsagul, C. and Krinke, J. (2017). Using compilation/decompilation to enhance clone detection. In *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, pages 1–7. IEEE.
- Rattan, D., Bhatia, R., and Singh, M. (2013). Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199.
- Saini, N., Singh, S., and Suman (2018). Code clones: Detection and management. *Procedia Computer Science*, 132:718–727. International Conference on Computational Intelligence and Data Science.
- Sajnani, H., Saini, V., Svajlenko, J., Roy, C. K., and Lopes, C. V. (2016). Sourcererc: Scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1157–1168.
- Selim, G. M., Foo, K. C., and Zou, Y. (2010). Enhancing source-based clone detection using intermediate representation. In *2010 17th working conference on reverse engineering*, pages 227–236. IEEE.