Compiling clones: What happens?

Oleksii Kononenko, Cheng Zhang, and Michael W. Godfrey
David R. Cheriton School of Computer Science
University of Waterloo, Canada
{okononen, c16zhang, migod}@uwaterloo.ca

***Abstract*—Most clone detection techniques have focused on the analysis of source code; however, sometimes stakeholders have access only to compiled code. To address this, some approaches have been developed for finding similarities at the binary level. However, the precise relationships between source-level and binary-level similarities remains unclear: While a compiler will preserve the semantics of the source code in the transformation to an executable, the resulting binary may differ significantly in structure, including the addition and deletion of entities in the source model. Also, compilation sometimes acts as a kind of normalization, transforming syntactically different but semantically similar structures into the same binary-level representation.**

In this paper, we describe a preliminary study into the effects of the *javac* Java compiler on the results of clone detection. We use CCFinderX — which can perform clone detection on sequences of arbitrary tokens — to find clones in both the source code and the corresponding bytecode of four large Java systems. The study shows that source code and bytecode clone detection can produce significantly different results, especially for large programs. We report on a few typical examples of differences, and analyze how they are introduced by the compiler. Finally, we discuss the greater significance of this work, and sketch plans for expanded study.

I. Introduction

Software clone detection — that is, finding similar design structures that recur within a software system — can provide useful input to many software maintenance activities: identifying emergent abstractions and refactoring opportunities, finding defects in related code structures, and overall improved program comprehension. There exist a variety of techniques for clone detection, most of them focused on program source code. In order to abstract out irrelevant differences and detect genuine logic/semantic code similarity, these techniques often transform source code into other representations — such as character strings, token streams, abstract syntax trees, and program dependence graphs — that are amenable to automated analysis of similarity [1]–[3].

With the same purpose of detecting code clones, some techniques use compiled code as the basic representation of clone detection [4]–[6]. These techniques first compile source code into an intermediate representation (IR), and then use string- or token-based algorithms to detect clones. Typically, bytecode and assembler are used as the IR of Java and C/C++ source code, respectively. Other IRs can also be used, for example, the approach by Selim et al. [7] detects clones based on the Jimple IR, to complement the result of source code clone detection. Since the process of compilation generally transforms the structure of the code in a significant way while preserving the semantics, clone detection on compiled code is free of many irrelevant source level differences and is able to reveal logic similarity embedded in source code.

One of issues that has motivated clone detection research on the bytecode level is the fact the source code is not always available in the first place. Additionally, the application of such analysis is somewhat different from that of source code clone detection — here, researchers have typically focused on detecting suspected copyright violations and malware code injections. This analysis is based on the assumption that the original code and its compiled versions are similar to some degree, meaning that clone detection is likely to produce similar results as well. However, no empirical evidence has been shown to support this assumption. It remains unclear whether analyzing compiled code generally produces different results, and more importantly how the results differ from those of source level techniques and why. We have conducted a preliminary study to shed light on these key questions about clone detection on compiled code. Specifically, we seek to answer three research questions:

- RQ1 Does (token-based) clone detection produce significantly different result sets when performed on both source code and compiled code?
- RQ2 Are there clones that can be detected only at the source code level? Why?
- RQ3 Are there clones that can be detected only at the compiled code level? Why?

We have used CCFinderX — a well known and widely used clone detection tool — as the representative of token-based clone detection techniques [1], [8]. We ran CCFinderX on both the source code and the compiled bytecode of four medium- to large-sized open source Java programs that are in wide use. After cleansing the result data through some automated filtering steps, we manually analyzed the clone sets reported at both levels. We found systemic differences between the results of clone detection at the different levels; we also observed a number of factors that appear to underline these differences.

II. Background and Related Work

Clone detection has received much attention in the research community, and there are many kinds of techniques and supporting tools to perform it. While it is possible to categorize these approaches across different dimensions, here we choose to distinguish between source code-based and compiled code-based approaches.

Source code clone detection — Based on the amount and kind of preprocessing that the source code goes through before clone detection is performed, source code-based techniques can

be further categorized into three families: text based, token based, and tree/graph based. Text-based approaches usually involve limited preprocessing followed by textual comparison. For example, NiCad [9] pretty-prints the parsed source code, and then performs line-by-line comparisons to find clones. MOSS [10] has a set of document type specific front-ends that "clean" the input; after that, it uses "fingerprinting" based on a rolling hashing function to detect similar fragments of code. While token-based approaches share the same initial step — converting an input into a sequence of tokens — they differ in how they analyze tokens. Baker's Dup tool first replaces the tokens that correspond to identifiers with their offsets, and then uses a suffix-tree algorithm for line-based comparison [4], [5]. CCFinder by Kamiya et al. [1] is similar to Dup but it replaces all identifiers with a single, special token and also performs language-specific normalization of the token sequence before executing a suffix tree algorithm. Once the analysis is done and clones are found, their locations are mapped back to the original source code. The last family consists of a variety of complex language-dependent approaches that build parse trees [11], abstract syntax trees [2], or program dependency graphs [3] from the source code and then apply different algorithms to find similar sub-trees or sub-graphs.

Binary code clone detection — The common motivation for analysis at this level is that source code is not always available, thus a source-based tool cannot be used. Since the source code for each programming language is compiled into a very specific form, each detection tool typically targets only one language. We note that there are fewer clone detection tools based on compiled code, compared to source code-based tools. Baker and Manber [12] adapted three tools initially designed for the source code (Dup [4], Siff [13], Diff) to find similarities in Java bytecode. Salim et al. [7] proposed to augment source code clone detection with results from bytecode analysis; to do so, they transform Java bytecode into an IR in the Jimple format, and use both CCFinder and Simian to analyze this intermediate representation as Java code. Davis and Godfrey developed two tools for clone detection in compiled code: JCD for Java bytecode and ACD for C/C++ assembler [6]; both tools are based on the same logic of applying a greedy algorithm followed by a hill climbing algorithm to find duplicates in the code. Keivanloo et al. developed SeByte [14], a tool uses semantic-enabled token matching to find clones in Java bytecode; it performs clone detection for each type of tokens, and then uses the Jaccard similarity coefficient to compute the similarity value. Chen et al. [15] proposed an approach to find clones within Android App markets; they extract control flow graphs (CFGs) from Android apps, compute a characteristic called centroid for each CFG, and then use these centroids to find similar apps.

To the best of our knowledge, there has been no systematic study examining the differences between clone detection results performed at source code and compiled code levels. However, both Salim et al. [7], and Davis and Godfrey [6] include some observations in their studies that approach this topic. In the former study, the authors used the same clone detection techniques on both source and compiled code; however, they analyze an abstraction of bytecode rather than "pure" bytecode per se. Also, the authors were focused on a quantitative comparison of clone detection on different levels and combining the results into a single, inclusive report. In the latter study, the authors used different tools for finding clones in source code and in the C++ assembler. They presented only quantitative results on the agreement between those two tools for different values of minimum length of a clone.

III. Design of Experiment

In this section, we describe how we adapted the CCFinderX clone detection tool for use in our study, we discuss the systems we selected for analysis, and we describe our approach to studying cloning at the source code and bytecode levels.

A. CCFinderX as the clone detector

We decided to use CCFinderX [8] for our study, and to focus on systems written in Java. CCFinderX is a token-based clone detection tool designed to detect Type 1 and Type 2 clones, according to the Bellon taxonomy [16]; it is an improved version of the original CCFinder tool [1]. We chose to run CCFinderX on both the source and bytecode using the default settings: 50 tokens for minimum clone length, and 12 for minimum number of token kinds. Since CCFinderX supports clone detection for Java source code out-of-the-box, we needed no additional preprocessing steps to process the source code of our example systems. However, CCFinderX cannot work directly with Java bytecode because it does not have embedded rules for parsing and tokenizing the input. To address this issue, we performed preprocessing of Java class files and extended CCFinderX with a new set of rules for lexical analysis of the preprocessed files.

Bytecode preprocessing — A Java compiler transforms source code into a set of Java *class* files; these files are in a binary format and cannot be passed directly into source code-based clone detection tools. We use a tool `javap` from the Java SDK to get textual representations of the *class* files. Once we have the disassembled text files, we split them into separate files so that each file contains only one method; we do this to force CCFinderX to detect clones that do not cross method boundaries. Finally, we normalize the content of the files by applying the set of transformations proposed by Baker et al. [12]: put one opcode or argument per line, differentiate between different types of elements, and replace all non-opcode elements with their offsets. Baker et al. showed that these transformations of the input improves the performance of clone detection tools allowing them to detect longer clones.

Extending CCFinderX to bytecode — CCFinderX consists of two main parts: a set of language-specific front-ends that transform the input data into a sequence of tokens, and a language-independent clone detection engine that finds similarities in that sequence. CCFinderX also allows users to extend the tool by adding new front-ends for additional target languages. In our new front-end, each opcode has its own token type; additionally, references into the constant pool, references into the local variable table, both signed and unsigned integers, and jumps have different token types assigned to each group. Also, we mark all non-opcode tokens as identifiers, so CCFinderX can perform parameterized matching.

B. Subject programs

Bellon et al. [16], [17] compared the performance of different source code-based clone detection tools. The authors

TABLE I. SUBJECT SYSTEMS

System	Size in Java (KLOC)	# Java source Files	Size in Bytecode (KLOC)	# Java class files	# Bytecode preprocessed files
netbeans-javadoc	9	101	54	230	1380
eclipse-ant	16	178	105	267	2156
eclipse-jdtcore	98	741	586	823	8772
j2sdk1.4.0-javax-swing	103	538	639	1588	13992

used four C/C++ systems and four Java systems. They slightly changed the source code of those systems: empty lines were removed, and curly braces were moved one line up if there was no other elements in a particular line. They asked the authors of several clone detection tools to submit the results of analysis of the normalized code. After that, they manually analyzed 2% of submitted clones and constructed a reference corpus for evaluation of clone detection tools. Although we do not use the corpus data in this paper, we did use the four Java systems from their study; we plan to incorporate the corpus data in the future work.

Table I provides some information about the systems we use in our study. Since comments in the source code are automatically ignored by the clone detection tools, we report the size of the systems in terms of the number of lines of actual code. The size of the bytecode representation for each system is much bigger than the size of a corresponding source code; however, these differences are due only to the fact that we output only one element (opcode, reference, jump, etc.) per line during preprocessing. There is also a difference between the number of source code files and the number of class files, because the Java compiler outputs nested and anonymous classes into separate class files.

C. Procedure of analysis

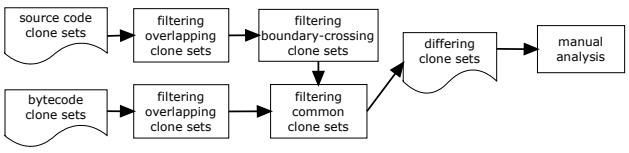

Fig. 1. Workflow of our study. The filtering steps are automatic, and the last step is done manually.

Figure 1 shows the workflow of our analysis of source level and bytecode level clone detection results. Before manually analyzing the results — which are clone sets reported by CCFinderX — we cleanse them by filtering out the *useless* and *common* ones.

The most common kind of useless results are *overlapping* clone sets: a clone set is said to be overlapping if two or more of its clone pairs overlap with each other; that is, they share some program code. Overlapping clone sets have little value for our analysis and make the manual analysis unnecessarily complicated. For example, it is almost impossible to identify boundaries between clone instances due to overlap. Therefore, we filter out overlapping clone sets from the results of both source code level and bytecode level analysis; this is done programmatically, based on the token information generated by CCFinderX. Our filtering program can accurately identify overlapping clone sets by comparing indices of beginning and ending tokens of their constituent clone instances.

Another kind of useless result is *boundary-crossing* clone sets, which contain clone pairs that span multiple methods. Due to the way in which we pre-process Java class files, bytecode level clone sets can never cross method boundaries. Because CCFinderX performs token-based clone detection, it does not really takes into account the syntactical structure of the source code. Although the tool places a restriction that a clone can start only at the beginning of a statement (which is done by specifying a list of tokens that can be at the beginning of a clone), it does not check whether a clone crosses method boundaries. In order to make the results at the two analysis levels comparable, we need to filter out the boundary-crossing clone sets from the source code results. Again, we use an automatic filter to eliminate boundary-crossing clone sets. Apart from token indices generated by CCFinderX, this filter parses source code to determine method boundaries. Then, by comparing offsets of beginning and ending tokens against offsets of method boundaries, the filter can identify clone instances extending across multiple methods.

We also filter out *common* clone sets, i.e., clone sets found in both the source and bytecode analysis. While these may consist of "true" clones, we are interested here only in the differences between the two analyses; consequently, we developed a third program to filter out the common clone sets. To avoid missing relevant clone sets, the filter is designed to be strict: a pair of clone sets (one at each level) is filtered out only if they have the same number of clone instances and there is a one-to-one mapping between the clone instances at both levels. More precisely, within a pair of common clone sets, every clone instance at the source level should correspond to a clone instance at the bytecode level at the same location of the same source file. If compiled with `-g` option (to enable debugging), Java class files include "Line Number Tables" that map some opcodes to locations in the original source code. We use this information to match the locations of clone instances at different levels. We did not run this tool on the two smallest systems, because their size meant that it was practical to perform a more accurate manual analysis of the clone detection results.

It should also be noted that in case of two largest subject systems, `jdt-core` and `swing`, we analyzed only a sample of the clone sets; this is because there were thousands of clone sets reported, and it would be impractical to analyze all of them. For both bytecode and source code clone sets, we sorted them by their *length* — defined as the maximum number of tokens of their constituent clone instances — and selected 1 out of 20 to analyze; that is, the sample rate is 5%, and the samples cover clone sets of varying size.

After the filtering and sampling, we then performed manual analysis; while this requires subjective judgement, we followed a defined protocol whenever possible to minimize bias:

1) For a clone set that is present in the source code results but not in the bytecode results, we first identify the class files that are expected to contain the clone instances, by checking the class and method names in the source code.
2) Next, we locate the bytecode opcodes corresponding to the source code snippets using line number information in the source files and the class files.

TABLE II. RESULTS OF CLONE DETECTION ON SOURCE CODE AND BYTECODE.

Subject	#Src	#Byte	#Common	#SrcOnly	#ByteOnly
netbeans-javadoc	94	94	45	49	47*
eclipse-ant	74	83	39	35	43*
eclipse-jdtcore	1207	1118	308	899	810
j2sdk1.4.0-javax-swing	470	690	96	374	594
total	1845	1985	488	1357	1494

* — more than one clone set is needed to match a clone set from the other level

3) Finally, we scrutinize the different pieces of bytecode to determine why they are not detected as clone instances. Specifically, we check the opcodes from the beginning to the end, and also consider several opcodes before and after the central opcodes; we do this because the line number information does not perfectly match that of the source code due to simplifying assumptions made by the compiler.

When analyzing the clone sets that are detected only at bytecode level, we followed an analogous protocol.

IV. RESULTS AND ANALYSIS

Table II shows statistics of clone detection on source and bytecode levels. The first column displays names of subjects. The columns "#Src" and "#Byte" show the number of clone sets detected at source level and bytecode level, respectively. The column "#Common" shows the number of clone sets that are detected at both levels. The column "#SrcOnly" shows the number of clone sets detected at source level only, while the column "#ByteOnly" shows the number of clone sets detected at bytecode level only.

As can be seen in Table II, in the two smallest subjects, there are about 50% clone sets that are specific to only one level. In the two larger subject systems the results are even more diverse at the two levels. In `eclipse-jdtcore`, about 72% of bytecode results are not detected at the source code level, and 74% of source code results are missing in bytecode level. For `swing`, the rates are 86% and 80%, respectively.

We can now answer RQ1: *Token-based clone detection performed on source code and compiled code can produce significantly different result sets.* Through manual analysis, we also found that different clone sets can be attributed to various factors, which we now discuss.

A. Clone sets specific to source level detection

As shown in the column "#SrcOnly" of Table II, there are 1357 clone sets detected only at the source level. These clone sets are caused mainly by the changing length of code, creating separate files, and the generating different bytecode sequences for syntactically similar source code during compilation.

A single statement of Java source code is often translated into multiple opcodes in Java bytecode; this is because each opcode performs an elementary operation, while a source code statement usually corresponds to a sequence of elementary operations. However, long source code sequences can be compiled into short bytecode sequences. For example, long qualified names, such as `System.out` may take up several tokens at source level, but they are translated into simple opcodes, like `getstatic`. When the length of source code snippet is above the threshold and the corresponding bytecode sequence is below it, the clone set can be detected only at the source level.

During compilation of a Java program, a `.class` bytecode file is generated for each class (or interface) in the source code base. If a source code class contains inner classes, the rules of the Java language require that they must all be declared in the same source code file; however, the bytecode of any inner classes will be located in `.class` files that are separate from the `.class` file of the defining "outer" class. Thus, if part of the source code of a detected clone set belongs to an inner class, then part of the corresponding bytecode will be placed into another `.class` and cannot be connected with the bytecode of the outer class during clone detection. This often results in missing clone sets at the bytecode level.

Also, although a compiler ought, in principle, work as a normalizer of the source code, this is not always the case; sometimes, a small difference in the source code, which is ignored by the clone detector, may result in bytecode level differences that change the results of clone detection. For example, in one clone instance, the boolean literal `true` is used, while in the other clone instance of the same clone set the literal `false` is used. At the source level, CCFinderX considers both `true` and `false` as boolean literals and does not differentiate between them. However, in the bytecode, `true` and `false` yield different opcodes (i.e., `iconst_1` and `iconst_0`), making the corresponding bytecode sequences differ from each other, and no clone set is detected for them.

We also found that the compiler may produce different sets of opcode instructions for same source code when this code is placed in static and non-static methods. For the non-static methods, a compiler adds additional opcodes to load references to the current object (`this`) onto the stack; however, it does not do this for static methods. In such situations, a simple Type 1/Type 2 clone in source code becomes a Type 3 clone in bytecode. As CCFinderX does not detect Type 3 clones well, we receive a clone set that is detected only at one level.

To summarize our observations above, we can now answer RQ2: *There are clone sets that can be detected only at the level of source code, mainly because the compiler may shorten code sequences, create extra `.class` files, and generate different opcodes instead of operands.*

B. Clone sets specific to bytecode level detection

As shown in the column "#ByteOnly", there are 1494 clone sets detected only at the bytecode level. The reasons for this phenomenon relate mostly to the change of length of code, normalization of control structures, and the treatment on identifiers.

Some source code structures — such as static field definitions and array accesses — often lead very long bytecode sequences. Sometimes the source code snippet is too short to be detected, but the length of the corresponding bytecode well exceeds the threshold for clone detection. In this case, only bytecode level clone set can be detected.

At the source code level, the basic control structures can be used quite flexibly, especially for nested `if...else...` statements and loops. However, during compilation, most

control structures are translated into opcodes for conditional jumps. Even if two pieces of source code appear to have significantly different structures, their bytecode sequences can be very close to each other. In such cases, the compilation has normalized the program using limited opcodes, showing that the control flows of two seemingly different source snippets are essentially similar; that is, bytecode analysis has detected a true semantic clone that is hard to identify using syntactic approaches on the source code.

Given the above discussions, we can now answer RQ3: *There are clone sets that can be detected only at the bytecode level, mainly because the compiler may elongate the code sequences and normalize control flows.*

C. Threats to validity

We are aware of some threats to the validity of our study. Since we used CCFinderX as the only clone detector in the study, the result is unlikely to generalize to other clone detectors. Nevertheless, as a representative token-based clone detection technique, CCFinderX fits well to this preliminary qualitative study. Also, during our manual analysis, we have paid special attention to the differing clone sets that are likely introduced by the specific algorithms of CCFinderX, and excluded them from further analysis.

Our use of the *javac* Java compiler confines the validity of results to the specific compilation technique of Java platform. Our results might not be applicable to other Java compilers, other languages that use some intermediate representation of the source code such as C#, and likely are not applicable to languages that are compiled into pure machine code, such C/C++.

Only four Java programs were used in the study, imposing a threat to generalizing the results to other programs. In addition to being used by previous studies, these four programs are diverse in terms of size and application domain. It is worth noting that our analysis of results is mainly qualitative, rather than quantitative.

Another threat stems from the filtering and sampling steps of our study. As explained in Section III, the filtering of overlapping and boundary-crossing clone sets can filter out clone sets that are actually useless to the study. A major concern is that the rules of filtering equivalent clone sets can be too strict, so that some clone sets, which should have been viewed as equivalent by human developers, may be considered to be different. During our manual analysis, we encountered few such cases, but the differences between clone detection results (shown in Table II) might be inflated. Clearly, it is possible that some interesting clone sets may be overlooked due to the sampling step. However, considering the sheer number of clone sets in the two larger subjects, sampling is necessary to make the manual analysis feasible.

V. Conclusion and Future Work

Clone detection results for the source code and bytecode levels are similar yet different. After studying four Java systems of different sizes we found that each of these levels has its own unique features that translates into a difference in the results. The manual analysis is extremely time consuming, and in the vast majority of cases it results in finding clone sets that are similar but have not been removed by filtering (usually because of 1-2 line shift). A deeper study would be aided by a tool that can intelligently identify and discard similar clone sets, so a human need focus only on analyzing the "true" differences.

We plan to extend this study to examine the other four systems, written in C/C++, from the Bellon benchmark. C/C++ compilers usually allows a programmer to select different settings for code optimization which will likely result in new kinds of differences between two levels. At the same time, some observations, such the effects of changing the length of code and normalization of control structure, may be valid in a wide range of compilation techniques. In addition, we want to use the data published by Bellon as an oracle to calculate statistical information for each type of differences.

References

[1] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. on Software Engineering*, vol. 28, no. 7, July 2002.

[2] R. Koschke, R. Falke, and P. Frenzel, "Clone detection using abstract syntax suffix trees," in *Proc. of the Working Conf. on Reverse Engineering*, 2006.

[3] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proc. of the ACM/IEEE Intl. Conf. on Software Engineering*, 2008.

[4] B. S. Baker, "A program for identifying duplicated code," in *Proc. of Computing Science and Statistics: 24th Symposium on the Interface*, 1992.

[5] ——, "On finding duplication and near-duplication in large software systems," in *Proc. of the Working Conf. on Reverse Engineering*, 1995.

[6] I. J. Davis and M. W. Godfrey, "From whence it came: Detecting source code clones by analyzing assembler." in *Proc. of the Working Conf. on Reverse Engineering*, 2010.

[7] G. M. K. Selim, K. C. Foo, and Y. Zou, "Enhancing source-based clone detection using intermediate representation," in *Proc. of the Working Conf. on Reverse Engineering*, 2010.

[8] "CCFinderX," http://www.ccfinder.net/ccfinderxos.html.

[9] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. of the IEEE Intl. Conf. on Program Comprehension*, 2008.

[10] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proc. of the ACM Intl. Conf. on Management of Data*, 2003.

[11] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. of the IEEE Intl. Conf. on Software Maintenance*, 1998.

[12] B. S. Baker and U. Manber, "Deducing similarities in Java sources from bytecodes," in *Proc. of the USENIX Annual Technical Conf.*, 1998.

[13] U. Manber, "Finding similar files in a large file system," in *Proc. of the USENIX Annual Technical Conf.*, 1994.

[14] I. Keivanloo, C. K. Roy, and J. Rilling, "Sebyte: A semantic clone detection tool for intermediate languages," in *Proc. of the IEEE Intl. Conf. on Program Comprehension*, 2012.

[15] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. of the ACM/IEEE Intl. Conf. on Software Engineering*, 2014.

[16] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Trans. on Software Engineering*, vol. 33, no. 9, 2007.

[17] "The Bellon benchmark," http://www.bauhaus-stuttgart.de/clones/.