Sub-clones: Considering the Part Rather than the Whole

Robert Tairas[1] **and Jeff Gray**[2]
[1]Department of Computer and Information Sciences, University of Alabama at Birmingham, Birmingham, AL
[2]Department of Computer Science, University of Alabama, Tuscaloosa, AL

Abstract - *Researchers have proposed various automated clone detection tools that can assist programmers in finding code clones (i.e., duplicated sections of code). Such tools can serve as input into the process of clone removal through refactoring activities. In this paper, we consider the scenario in which an automated clone detection tool was used to find code clones as part of the clone refactoring process. Actual refactorings associated with the detected clones were obtained from changes identified between consecutive versions of open source software artifacts. Based on two separate studies, observations reveal instances of sub-clone refactoring where only part of the clone ranges are actually refactored. We conclude that sub-clone refactoring should be considered to augment refactoring performed on the entire clone.*

Keywords: code clones; analysis; refactoring; maintenance

1 Introduction

Code clones represent sections of code that are duplicates of each other. The structural similarity between these clones is based on the degree to which they match each other (i.e., identical copies of each other, contain identifier differences, or contain a few extra or missing statements). The amount of clones in software can be as much as 15% based on several studies of cloning in software [4][9][14].

When a section of code that is related to a group of clones (i.e., clones representing the same duplication) needs to be updated, it is necessary to determine whether the code associated with the other clones in the group must also be updated. Failure to update all relevant and duplicate sections of code can introduce errors in the program. Removing the duplication represented by the clones is one way to reduce the possibility of errors. The activity of refactoring [7] (i.e., when the code is changed, but its behavior is not) can be used as a means to remove the duplication associated with the clones. Refactorings such as *Extract Method* and *Pull-up Method* can modularize the code represented by the clones, thus eliminating the duplication.

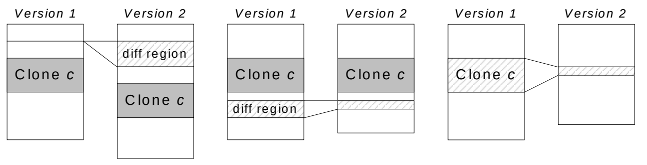
Research into the detection of clones that can automatically find clones in code have produced various techniques and tools [4][9][10][14][16][17]. However, knowledge regarding the utilization of these tools in the practice of software maintenance and specifically within the context of clone refactoring is still limited. This paper describes our investigation into how clones reported by an automated clone detection tool are refactored. We focus our interest on observing characteristics of the clones in a scenario where a programmer uses a clone detection tool to find the clones before performing clone refactoring. Our observations are based on mining the changes identified between consecutive versions of open source software artifacts. We found that in some cases the range of the clone and the range of code that was actually refactored differed in that refactoring was performed on only part of the clone (i.e., a sub-clone). By evaluating the instances of the partially refactored clones, we discovered characteristics influencing sub-clone refactoring. Such characteristics can inform a process that utilizes clone detection in an effort to remove duplicate code where sub-clones are also considered, thus augmenting the choices of refactoring given to the maintainer of the code.

The rest of the paper is organized as follows: the next section introduces the study that was performed to observe actual refactoring of duplicated code revealing sub-clone refactoring. Section 4 provides a further study focusing on one clone detection tool and additional open source projects. A discussion of general observations from our studies is included in Section 5. Related work, a conclusion and future work are given in sections 6 and 7, respectively.

2 Observing Clone Refactoring

In this paper, we focus our interest on observing the refactoring of clones that occur between consecutive versions of a software release. A programmer may perform some activity of refactoring on the original code of one version and the result of this refactoring is subsequently evident in the next version. In this case, the original code and refactored code is stored within two consecutive versions. By comparing two consecutive versions of the code, we are able to find actual refactorings performed between the two versions. More specifically, we focus on finding clone-related refactorings.

Our method of finding clone refactorings between code versions starts with the use of a clone detection tool. The tool is executed on the source code of one version. This step identifies the sections of code that are reported as clones by the tool. The next step is to compare these sections of code with the corresponding sections of code in the next version of the source code. The comparison utilizes the Unix *diff* command. For each section of code reported as a clone, the file where it resides is compared using *diff* with the next version of that same file. A batch process automates this step for all clones that were reported by the clone detection tool. We are interested in the changes reported by *diff* that consist of deletions and additions of lines in the approximate location of the reported clone range. Figure 1 illustrates three different scenarios of changes in two versions of a file. Our focus in this paper is on instances of Figure 1(c), because the changes occur within the range of a clone. For example, the deletion of lines followed by the addition of one line within the clone range can signal the possibility of an *Extract Method* refactoring activity. Figures 1(a) and 1(b) represent changes outside the clone range that are ignored in our observations.



(a) Ignored (b) Ignored (c) Further evaluated

Figure 1. File changes based on *diff* information

3 Sub-clone Refactoring

We conducted a study to observe the refactorings that were performed on clone-related code among versions of the JBoss Application Server, which is an open source implementation of J2EE [8]. Specifically, these refactorings were identified by observing changes between two consecutive versions from version 2.2.0 to 4.2.3 (44 versions). It is worth noting that we do not know if the developers of JBoss used a clone detection tool to identify clones, which quite possibly was not the case. However, the main purpose of the study is to see how a section of code that would have been reported as a clone was maintained in actual practice.

Table 1. *Extract Method*-related refactorings in JBoss

Extract Method	14
Extract Method with Pull-up Method	1
Extract Method to utility class	6
Total	21

The Simian [16] clone detection tool was initially chosen to detect the clones in the first part of our study. Simian was selected because of its relatively fast speed in detecting and reporting clones. In Simian and a majority of other clone detection tools, clones that have been detected are represented by three properties: the location of the file containing the clone and the clone's starting and ending lines within that file. Thus, a clone, c, can be represented by $c = (f, s, e)$, where f is the file containing the clone, and s and e are the line numbers signifying the range of the clone. Clones that are duplicates of each other are grouped together by Simian in its report.

The process outlined in the previous section was performed and changes reported by *diff* within the clone ranges were further evaluated. Some common refactoring activities within the range of the clones were discovered in the JBoss version sequences, such as *Extract Method*, *Pull-up Method*, and *Extract Class* [7]. In many cases, the refactorings removed the duplication of code by extracting the duplicate functionality into a new method or pulling up duplicated code to a super class. Table 1 summarizes the 21 instances of *Extract Method*-related refactorings associated with clones found across the versions of JBoss. In addition to *Extract Method*, a *Pull-up Method* refactoring also contains *Extract Method* if only a part of a method is pulled up. Also, some statements were extracted to a separate class, generating more of a utility-type method. It was observed that only in two of the refactorings in Table 1 the range that was reported by Simian as a clone was exactly the range that was refactored. In the remaining cases, refactoring was not performed on the exact range of the reported clone.

To provide a wider spectrum of observation, we evaluated additional clone detection tools. Four other clone detection tools were selected: CCFinder [10], which is a token-based clone detection tool, and CloneDR [4], Deckard [9], and SimScan [17], which are tree-based tools. The detection configuration settings for each tool are listed in Appendix A. For this evaluation, these tools were not requested to find clones on all of the source files of JBoss, but rather only on the source files containing the Simian clones associated with the identified refactorings in Table 1. The purpose of this activity was to see how other tools presented (or not) the code ranges of the same clones that were originally detected by Simian.

Table 2. Coverage of *Extract Method*-related refactorings

No.	Clone Detection Tool	Exact Coverage	Larger Coverage	Total
1.	CCFinder	4	8	12
2.	CloneDR	6	9	15
3.	Deckard	8	3	11
4.	Simian	2	0	2
5.	SimScan	6	12	18

Table 2 provides the detection results from Simian and the four additional tools for the refactorings observed in the clone ranges initially detected by Simian. The "Exact Coverage" column records the number of times each clone detection tool reported a clone group consisting of clones that represented exactly the code range that was refactored. As can be seen in the table, the number of times this is the case for each tool were all less than half of the 21 total instances. For example, Deckard exactly matched only eight of the 21 instances. The "Larger Coverage" column represents the number of times a

```
1 2   4 5   protected String getValue(String name, String value) {
1 2   4 5     if (value.startsWith("${") && value.endsWith("}")) {
1 2 3 4 5 -     try {
1 2 3 4 5 -       String propertyName = value.substring(2, value.length()-1);
1 2 3 4 5 -       ObjectName propertyServiceON = new ObjectName(...);
1 2 3 4 5 -       KernelAbstraction kernelAbstraction = KernelAbstractionFactory.getInstance();
1 2 3 4 5 -       String propertyValue = ...;
1 2 3   5 -       log.debug(...);
1 2 3   5 -       return propertyValue;
1 2 3   5 -     } catch (Exception e) {
1 2 3   5 -       log.warn(...);
1 2 3   5 -     }
          +     String replacement = StringPropertyReplacer.replaceProperties(value);
          +     if (replacement != null)
          +       value = replacement;
1 2     5     }
1 2     5     return value;
1 2     5   }
```

Figure 2. File changes based on *diff* information

clone group contains clones that represent a larger coverage of code that included the statements that were refactored. In these instances, refactoring was only performed on part of the code range associated to the clones, an example of which can be seen in Figure 2 and is explained below.

Figure 2 shows an example of the coverage of the tools for a sequence of refactored statements between versions 4.0.5 and 4.2.0 of JBoss in the `EjbJarDDObjectFactory` class. Lines containing "…" were truncated for brevity. The range of the clone detected by each tool is marked with the corresponding number as assigned in Table 2 (i.e., CCFinder is 1 and SimScan is 5). The refactoring that occurs is represented by the sequence of deleted lines (i.e., identified with '-') being replaced by the sequence of added lines (i.e., identified with '+'). The refactored code in the method `getValue` is only the *Try-statement* inside the *If-statement*. Most tools (i.e., CCFinder, CloneDR, and SimScan) contain the entire method, which is more than the refactored code range and was counted as an instance of a "Larger Coverage" in Table 2. Simian's clone region contains only the method header and part of the code. Only the clone reported by Deckard is exactly the same as the range of the refactoring.

Simian looks for similarities in the textual representation of the code. This technique may not detect well-formed clones (i.e., clones representing proper syntactic blocks of code) or clones with subtle differences not related to the syntax of the code. Hence, Simian may not report the appropriate code range associated to code that was refactored. This is mainly why Simian did not have any instances designated as "Larger Coverage" in Table 2. In many cases the clone range reported by Simian did not consist of a syntactically meaningful block of code. The other tools utilize more structured representations of the code such as token or tree representations and thus can report more structured clones.

4 Deckard Sub-Clone Refactoring

In this section, we describe evaluations of observed refactorings related to clones in JBoss, and two additional open source projects: ArgoUML [2] and Apache Derby [1]. ArgoUML is an open source tool to develop UML diagrams written in Java. Apache Derby is an open source implementation of a relational database also written in Java. We used Deckard on these projects to further discover and evaluate instances of sub-clone refactoring.

4.1 Deckard characteristics

In Table 2, Deckard provided the most clones with ranges that are exactly the ranges of the observed refactorings. Clone detection tools typically combine smaller clones into larger clones that encompass the smaller clones to report maximal sized clones. Deckard reports both the maximal sized clone groups and groups representing smaller related clones. Because of this, Deckard not only reported a clone group of the exact code range that is refactored in Figure 2, but also reported a separate clone group containing a larger clone range consisting of the *If-statement* above the refactored *Try-statement*.

As a tree-based tool, Deckard can provide more syntactically meaningful clones compared to the text-based Simian tool. CCFinder, CloneDR, and SimScan also report syntactically meaningful clones, but Deckard's output consists of both the smaller clone groups and larger clone groups in which the smaller clone groups are contained. With its ability to report syntactically meaningful clones and the reporting of multiple sized clone groups, Deckard was used in a further study to observe relationships between reported clones and actual refactorings associated with these clones. JBoss was re-evaluated in addition to ArgoUML and Apache Derby. The evaluation of Deckard in the previous section differs in that it only detected source files associated with refactored clones initially detected by Simian, whereas in the study described in this section Deckard was run independently on all source files.

For ArgoUML, nine versions from 0.10.1 to 0.26 were observed. For Apache Derby, ten versions from 10.1.1.0 to 10.5.3.0 were observed. From the studies of these software artifacts, we determined properties of situations when a sub-clone, rather than the whole clone, was refactored.

4.2 Evaluation results

Table 3 (Refactoring Coverage) provides a summary of clones reported by Deckard that are related to refactorings observed in the three projects. The first general observation is that the number of clones detected by Deckard in JBoss that are associated with *Extract Method*-related refactorings is more than the results from Simian (i.e., 36 with Deckard compared to 21 with Simian). This larger number is mainly because Deckard can detect more structured clones that can contain superficial differences, which Simian does not identify as clones. In addition, refactoring performed on a selection of clones in a clone group is also included. The second general observation is that although Deckard provides smaller sized clone groups representing syntactic blocks below the maximal sized clone group, the observed refactorings related to some of these clones still only account for part of the code associated to the clone ranges. In Table 3, 14 instances in JBoss, 9 instances in ArgoUML, and 15 instances in Apache Derby represent refactorings that were not performed on the entire code range of clones detected by Deckard.

Table 3. Refactoring coverage and code properties

Property		JBoss	ArgoUML	Derby
Refactoring Coverage	Exact clone coverage	19	17	12
	Sub-clone coverage	**14**	**9**	**15**
Coverage Levels	Same level	4	4	6
	1 level above	9	2	8
	> 1 level above	1	3	1
Clone Differences	Refactorable	7	4	8
	Not refactorable	7	5	7

4.3 Sub-clone refactoring properties

We further evaluated the refactorings related to the “Sub-clone coverage” instances in Table 3 to determine characteristics that may have influenced the refactoring to be performed on only part of the duplicated code.

Deckard results – In Table 2, Deckard provided the most exact matches. Its results included smaller clone groups of the maximal sized groups allowing more exact ranges to be found. Even with these reported smaller clone groups, Table 3 (Coverage Levels) shows that in some cases refactoring was still performed under the syntactic level of the reported clone. For JBoss and Apache Derby, this is mostly the case (i.e., nine and eight instances, respectively) as seen in the “1 level above” row. These instances suggest a practice that keeps some logic of the code at the original location for better program comprehension. For example, in Figure 2, the *If-Statement* is not refactored; only the statements inside the block are refactored. This eliminates most of the duplicated code, but keeps some logic in the original location.

Excluded statements – In Table 3 (Coverage Levels), JBoss and ArgoUML consisted of four instances of clones being at the same level as the refactored code. Apache Derby consisted of six instances. However, some statements in the same level were not included as part of the refactoring. An example can be seen in Figure 3, where the first and last statements in the *If block* were not refactored although they were part of the clone.

```
     if (edge instanceof MTransition) {
       MTransition tr = (MTransition) edge;
 -     FigTrans trFig = new FigTrans(tr);
 -     // set source and dest
 -     // set any arrowheads, labels, or colors
 -     MStateVertex sourceSV = tr.getSource();
 -     MStateVertex destSV = tr.getTarget();
 -     FigNode sourceFN = (FigNode) lay...
 -     FigNode destFN = (FigNode) lay...
 -     trFig.setSourcePortFig(sourceFN);
 -     trFig.setSourceFigNode(sourceFN);
 -     trFig.setDestPortFig(destFN);
 -     trFig.setDestFigNode(destFN);
 +     FigTrans trFig = new FigTrans(tr, lay);
       return trFig;
     }
```
Figure 3. Incomplete block of refactored code

Clone Differences – Differences between clones can include variable names, literal values, and object types. The extent of a group of clones’ similarity influences their possibility of refactoring. In Table 3 (Clone Differences), we consider the hypothetical situation in which the entire clone was refactored rather than just the sub-clone. In this case, a section of code is refactorable if it meets the pre-conditions for the *Extract Method* refactoring activity. Specifically for clone refactoring, differences in the clones should be able to be passed to the new method to allow for a generalized version of the duplicated code. For example, clones with variable name differences can be refactored by including a formal parameter for those variables in the new method signature. In all three software artifacts, the instances in which an entire clone could have been refactored and the times it could not be refactored did not differ much. This implies that in some cases the programmer could have refactored the entire clone, but did not.

Instances counted as “not refactorable” include situations where the entire clone ranges contained object type differences where a variable in one clone is declared as one type and the same variable in another clone is declared as a different type. Such a situation is more difficult to refactor. A possible way of refactoring situations with more complex differences is to use the control coupling [13] mechanism that includes a flag to determine which extracted code to execute (i.e., for one clone execute one sequence of statements and for another clone execute a different sequence of statements in the extracted method). It was observed that the programmer did not use such flags to refactor these instances. In most cases, the programmer refactored the more exact parts of the clones with simpler differences such as variable names and literal values and did not include differences such as variable types.

5 Discussion

This section provides some points for consideration related to the studies described in this paper.

Differences of clone detection results – Table 2 shows the variations in the results of a clone detection tool that is run on the same set of source files. Even changing the configuration settings of one tool can result in a separate listing of clones. Sub-clones then can be considered relative to the tool that is used such that one tool may report a section of code as a clone, while another tool reports it as a sub-clone of a larger clone. However, if we look in terms of the use of these tools to assist in finding clones for refactoring, running multiple tools to search for clones could potentially increase the effort during the maintenance process in an unnecessary way. Selecting a single tool based on the maintainer's decision provides a more straightforward process. In this case, each tool, whichever is selected, will have its own set of clones with related instances of sub-clones that would be considered for refactoring.

Exact matching clones are easier to refactor and clone detection tools can be set to report only these types of clones. However, limiting the detection to only exact matching clones reduces the ability to observe the overall cloning of the system. If clones that contain specified differences were initially detected, the results from this detection can then be followed by evaluating whether the clone group with differences should be refactored rather than sub-clones that have more limited differences or exactly match each other. Sub-clones then can allow for more refactoring options.

Incorporating sub-clone refactoring into the clone maintenance process – Currently, if a clone reported by a clone detection tool is selected for refactoring, the process of refactoring requires many manual steps. For example, to replace several clones with a call to a new method, the method must first be extracted from one of the clone instances, which may include using a refactoring engine in an IDE. However, once the method is created each clone must be replaced with a call to that method. These are the same steps that need to be done when a sub-clone is selected for refactoring. A mechanism that can keep track of the clones and forward the necessary information to a refactoring engine upon approval of the programmer can reduce the amount of manual steps needed during clone maintenance. Based on the evaluation of sub-clones and their related refactorings in the previous section, a mechanism that can select sub-clones for refactoring should focus on allowing a programmer to select a sub-clone that is one or more syntactic levels below the main clone and the ability to include/exclude bordering statements.

6 Related Work

Related work is summarized in this section, which includes work related to the observation of clone evolution, refactoring between versions, and identifying crosscutting concerns with clones.

Clone evolution analysis – The evolution of clones in multiple release versions has been studied for various purposes, such as how the clones are maintained. Kim et al. [11] generated genealogies of clones and provided several categories related to how clones evolved (e.g., a new clone added or one was subtracted, clones were consistently or inconsistently changed). The studies described in Sections 3 and 4 differ from [11] in that they are specifically looking at the properties of the code where refactoring occurred, whereas Kim et al. focused on characteristics of clones that made refactoring unsuitable.

The work of both Aversano et al. [3] and Krinke [12] looked at how consistently clones were maintained in terms of keeping the code associated with clones of the same group consistent with each other when an update is required. They provided overall trends of how clones were consistently or inconsistently changed during a specific time frame, whereas the evolution analysis in this paper mainly considered refactoring of clones between two versions of the same source code.

Refactoring identification – Some works have looked at the instances of refactoring between versions in general without a specific focus on refactoring of clones [6][15]. However, these focused on giving an overall view of the activity of refactoring. Our paper focuses specifically on refactoring instances related to clones resulting in the observation of the relationship between the ranges of refactored code and the actual reported clone. Weißgerber and Diehl proposed a technique to detect refactorings, where a clone detection tool was used [19]. However, the detection tool was not used to determine clone-related refactorings, but rather to improve the results of the technique. Clone-related refactorings could be part of the results of the technique, but a post-processing step must be done to identify them.

Clone range analysis – A comparison of the line ranges of clones reported by clone detection tools and the lines annotated as crosscutting concerns was studied by Bruntink et al. [5]. The evaluation of multiple clone detection tools in our paper looks at instances of refactoring that had already occurred between two versions, whereas in [5] the crosscutting concerns were not already changed into aspects, but rather were determined at the beginning by a human observer.

7 Conclusion and Future Work

This paper described the analysis of code clones identified by a clone detection tool and their relationships with actual refactorings obtained from changes between consecutive versions of the source code. According to our evaluation of running different clone detection tools on open source software artifacts, the refactoring of parts of clones or sub-clones is evident in several cases. We conclude that sub-clone refactoring should be included in the clone maintenance process. Such support should allow programmers to selectively determine partial ranges in a clone for refactoring within its syntactic hierarchy in addition to the exclusion of edge statements.

For future work, we plan to include support for sub-clone refactoring in our Eclipse plug-in called CeDAR (Clone Detection, Analysis, and Refactoring) that currently can parse the output of several clone detection tools and display clone information within the Eclipse IDE [18]. The plug-in represents an effort to assist the programmer in the refactoring of clones. Future work related to this paper will include a mechanism for the programmer to select sub-clones within a clone for refactoring.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 0702764.

8 References

[1] Apache Derby, http://db.apache.org/derby.

[2] ArgoUML, http://argouml.tigris.org.

[3] L. Aversano, L. Cerulo, and M. Di Penta, "How Clones are Maintained: An Empirical Study", European Conf. on Software Maintenance and Reengineering, Amsterdam, The Netherlands, March 2007, pp. 81–90.

[4] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone Detection using Abstract Syntax Trees", Int. Conf. on Software Maintenance, Bethesda, MD, November 1998, pp. 368–377.

[5] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwe, "On the Use of Clone Detection for Identifying Crosscutting Concern Code", IEEE Trans. on Software Engineering, vol. 31, pp. 804–818, October 2005.

[6] S. Counsell, Y. Hassoun, G. Loizou, and R. Najjar, "Common Refactorings, a Dependency Graph and Some Code Smells: an Empirical Study of Java OSS", Int. Symp. on Empirical Software Engineering, Rio de Janeiro, Brazil, September 2006, pp. 288–296.

[7] M. Fowler, Refactoring: Improving the Design of Existing Code, Reading, MA: Addison-Wesley, 1999.

[8] JBoss, http://www.jboss.org.

[9] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and Accurate Tree-based Detection of Code Clones", Int. Conf. on Software Engineering, Minneapolis, MN, May 2007, pp. 96–105.

[10] 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, pp. 654–670, July 2002.

[11] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An Empirical Study of Code Clone Genealogies", European Software Engineering Conf. and the Symp. on the Foundations of Software Engineering, Lisbon, Portugal, September 2005, pp. 187–196.

[12] J. Krinke, "A Study of Consistent and Inconsistent Changes to Code Clones", Working Conf. on Reverse Engineering, Vancouver, Canada, October 2007, pp. 170–178.

[13] S. Lawrence and J. Atlee, Software Engineering: Theory and Practice, Upper Saddle River, NJ, Prentice Hall, 2006.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code," Symp. on Operating System Design and Implementation, San Francisco, CA, December 2004, pp. 289–302.

[15] C. Schofield, B. Tansey, Z. Xing, and E. Stroulia, "Digging the Development Dust for Refactorings", Int. Conf. on Program Comprehension, Athens, Greece, June 2006, pp. 23–34.

[16] Simian, http://www.redhillconsulting.com.au/products/simian.

[17] SimScan, http://blue-edge.bg/simscan.

[18] R. Tairas, and J. Gray, "Get to Know Your Clones with CeDAR", Int. Conf. on Object-Oriented Programming, Systems, Languages, and Applications, Orlando, FL, October 2009, pp. 817–818.

[19] P. Weißgerber, and S. Diehl, "Identifying Refactorings from Source-Code Changes", Int. Conf. on Automated Software Engineering, Tokyo, Japan, September 2006, pp. 231–240.

Appendix A. Clone Detection Tool Configuration Settings

This appendix summarizes the non-default configuration settings chosen for the clone detection tools used in the studies presented in this paper. In most cases the value set was the default value given by a tool. An explanation of the settings and value selected is also given.

Tool	Setting	Value	Description
CCFinder	Shaper level	Hard	Generate clones enclosed in blocks as much as possible
CloneDR	All default settings		
Deckard	Similarity	0.95	Allow for small differences in clones
	Stride	0	Limit the size of the clones
Simian	All default settings		
SimScan	Volume	Small	Include smaller matches
	Similarity	Loosely similar	Allow for small differences in clones
	Speed /quality	Exhaustive search	Provide more detailed results