Comparing Software Bugs in Clone and Non-clone Code: An Empirical Study

Judith F. Islam*, Manishankar Mondal†, Chanchal K. Roy‡ and Kevin A. Schneider§

Department of Computer Science
University of Saskatchewan
Saskatoon, SK, Canada S7N 5C9

*judith.islam@usask.ca
†mshankar.mondal@usask.ca
‡chanchal.roy@usask.ca
§kevin.schneider@usask.ca

Code cloning is a recurrent operation in everyday software development. Whether it is a good or bad practice is an ongoing debate among researchers and developers for the last few decades. In this paper, we conduct a comparative study on bug-proneness in clone code and non-clone code by analyzing commit logs. According to our inspection of thousands of revisions of seven diverse subject systems, the percentage of changed files due to bug-fix commits is significantly higher in clone code compared with non-clone code. We perform a Mann–Whitney–Wilcoxon (MWW) test to show the statistical significance of our findings. In addition, the possibility of occurrence of severe bugs is higher in clone code than in non-clone code. Bug-fixing changes affecting clone code should be considered more carefully. Finally, our manual investigation shows that clone code containing `if-condition` and `if-else` blocks has a high risk of having severing bugs. Changes to such types of clone fragments should be done carefully during software maintenance. According to our findings, clone code appears to be more bug-prone than non-clone code.

Keywords: Code clones; software bugs; severe bugs.

1. Introduction

If two or more code fragments in a software system’s code-base are exactly or nearly similar to one another we call them code clones [1, 2]. A group of similar code fragments forms a clone class. Code clones are mainly created because of the frequent copy/paste activities of programmers during software development and maintenance [1].

A significant number of studies [3–20] have been conducted on discovering the impact of cloning on software maintenance. While a number of studies [3, 6, 7, 9–12]
have revealed some positive sides of code cloning, there is strong empirical evidence [4, 5, 8, 13–16, 18–20] of negative impacts of code clones too. These negative impacts include higher instability [15], late propagation [4], and unintentional inconsistencies [5]. Existing studies [4, 21] show that code clones are related to bugs in the code-base.

Several studies have showed that bugs have a great effect on code in software systems. Our previous study [22] provides evidence of bug-replication in clone code. Also, Sajnani et al. [23] showed that cloned code has less problematic bug patterns than non-cloned code. They have used the bugs reported by FindBugs [46] from just one snapshot of the last revision of the system. Whereas we consider bugs reported during the evolution of a software system through thousands of commits. In this paper [23], they worked on tool-reported bugs whereas we work on the developer-reported bugs. Moreover, they have considered only Java programming language whereas we work on two programming languages, C and Java. These issues motivate us to work on bug reports generated by developers to see the impact of bug-fix commits on both clone code and non-clone code. We consider bug-fixing commits reported by the developers from thousands of commits in open-source projects.

To explore the effects of bug-fix changes between clone and non-clone codes, we conduct a comparative study. We consider thousands of revisions of seven diverse subject systems written in two different programming languages (Java and C). We detect code clones from each of the revisions of a subject system using the NiCad [24] clone detector, analyze the evolution history of these code clones, and investigate whether and to what extent they contain bugs. To find non-clone bug-fix commits, we first identify all the commits that are related to fixing a bug. Among these bug-fix commits, we detect those which have clone code. We consider the remaining bug-fix commits as non-clone bug-fix commits. We automatically count the total number of files that contain changes in source code. Among these files, we detect those files which have changes in clone code. Omitting these files from the total files, we get the changes in non-clone code. Then we calculate the percentages of changes for both clone and non-clone code. We found that the percentage of changed files containing clone code is significantly higher than that of non-clone code. We validate our findings using the Mann–Whitney–Wilcoxon (MWW) [33] test for three types of clones with non-clone code.

We investigate the four research questions listed in Table 1. We find that the percentage of files changed due to bug-fix commits is significantly higher in clone code compared with non-clone code. Moreover, the percentage of files that have

<table>
<thead>
<tr>
<th>SL</th>
<th>Research Question</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ 1</td>
<td>What percentages of files get affected because of clone and non-clone bug-fix commits?</td>
</tr>
<tr>
<td>RQ 2</td>
<td>How often do bug-fix changes occur to the clone and non-clone codes?</td>
</tr>
<tr>
<td>RQ 3</td>
<td>Is there any difference between the severities of the bugs occurring in clone and non-clone codes?</td>
</tr>
<tr>
<td>RQ 4</td>
<td>Which types of severe and non-severe bugs can occur in code clones?</td>
</tr>
</tbody>
</table>
changes in Type 1 and Type 2 clone codes is higher than that of changes in Type 3 clone code (definitions of different types of clones will be given in Sec. 2). These findings can be used for ranking of clone code. Also, the occurrence of severe bugs is more in clone code than in non-clone code. Finally, we find that most of the severe bugs contain changes in if-condition. This is helpful for clone code management.

The rest of the paper is organized as follows. Sec. 2 contains the terminology, Sec. 3 discusses the experimental steps, Sec. 4 answers the research questions by presenting and analyzing the experimental results, Sec. 5 discusses the related works, Sec. 6 discusses possible threats to validity, and Sec. 7 concludes the paper and discusses possible future work.

2. Terminology

2.1. Types of clones

We conduct our analysis considering both exact (Type 1) and near-miss (Type 2 and Type 3) clones [1, 2]. The clone types are defined below.

Type 1 clones. If two or more code fragments in a particular code-base are exactly the same disregarding the comments and indentations, these code fragments are called exact clones or Type 1 clones of one another.

Type 2 clones. Type 2 clones are syntactically similar code fragments in a code-base. In general, Type 2 clones are created from Type 1 clones because of renaming of identifiers and/or changing of data types.

Type 3 clones. Type 3 clones are mainly created because of additions, deletions, or modifications of lines in Type 1 or Type 2 clones. Type 3 clones are also known as gapped clones.

2.2. Bug-fix commits

In the version control systems (e.g. SVN or Git) developers perform commits to keep track of the changes that they made in the code-base. Developers often identify reported bugs in the software systems and fix them. The commit that occurs to fix a reported bug is known as a bug-fix commit. To fix these bugs, changes may occur on clone code or non-clone code. We observe these changes both in clone and non-clone code to understand the contrast between them in terms of their bug-proneness.

2.3. Severe bugs

Severe bugs are the software defects which can make negative impact on the quality of software. Severe bugs are conventionally denoted as critical level in a bug report. Developers can define bug level depending on the criteria of bugs while reporting a bug in open-source projects.
3. Experimental Steps

We conduct our research on seven subject systems (three C and four Java systems). We consider these seven subject systems since these systems have variations in application domains, sizes, revisions, and also are used by our other studies. These subject systems are listed in Table 2 which were downloaded from the SourceForge online SVN repository [25]. In this table, the total number of revisions of each subject system is given along with the lines of code (LOC) in the last revision.

<table>
<thead>
<tr>
<th>System</th>
<th>Language</th>
<th>Domain</th>
<th>LLR (LOC in the Last Revision)</th>
<th>Revisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctags</td>
<td>C</td>
<td>Code Definition Generator</td>
<td>33,270</td>
<td>774</td>
</tr>
<tr>
<td>Camellia</td>
<td>C</td>
<td>Image Processing Library</td>
<td>89,063</td>
<td>170</td>
</tr>
<tr>
<td>Brлад</td>
<td>C</td>
<td>Solid Modeling CAD</td>
<td>39,309</td>
<td>735</td>
</tr>
<tr>
<td>jEdit</td>
<td>Java</td>
<td>Text Editor</td>
<td>191,804</td>
<td>4000</td>
</tr>
<tr>
<td>Freecol</td>
<td>Java</td>
<td>Game</td>
<td>91,626</td>
<td>1950</td>
</tr>
<tr>
<td>Carol</td>
<td>Java</td>
<td>Game</td>
<td>25,091</td>
<td>1700</td>
</tr>
<tr>
<td>Jabref</td>
<td>Java</td>
<td>Reference Management</td>
<td>45,515</td>
<td>1545</td>
</tr>
</tbody>
</table>

*LLR = LOC in the Last Revision.

3.1. Preliminary steps

We perform the following steps for detecting fixed bugs: (1) Extraction of all revisions (as stated in Table 2) of each of the subject systems from the online SVN repository. (2) Detection and extraction of code clones from each revision by applying NiCad [24] clone detector. (3) Detection of changes between every two consecutive revisions using UNIX `diff` command. (4) Locating these changes to the already detected clones of the corresponding revisions. (5) Detection of bug-fix commit operations. For completing the first four steps we use the tool SPCP-Miner [26]. We will describe the detection of bug-fix commits later in this section. In Sec. 4 we will describe how we detect bug-fix changes in clone and non-clone code.

We use NiCad [24] for detecting clones since it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [27, 28]. Using NiCad, we detect block clones including both exact (Type 1) and near-miss (Type 2 and Type 3) clones of a minimum size of 10 LOC with 20% dissimilarity threshold and blind renaming of identifiers. NiCad settings for detecting three clone types (Type 1, Type 2, and Type 3) are shown in Table 3. For different settings of a clone detector the

<table>
<thead>
<tr>
<th>Clone type</th>
<th>Identifier renaming</th>
<th>Dissimilarity threshold (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>none</td>
<td>0</td>
</tr>
<tr>
<td>Type 2</td>
<td>blindrename</td>
<td>0</td>
</tr>
<tr>
<td>Type 3</td>
<td>blindrename</td>
<td>20</td>
</tr>
</tbody>
</table>
clone detection results can be different and thus, the findings on bugs in code clones can also be different. Hence, selection of appropriate settings (i.e. detection parameters) is important. We used the mentioned settings in our research, because Svajlenko and Roy [29] show that these settings provide us with better clone detection results in terms of both precision and recall. Moreover, code clones with a minimum size of 10 LOC are more appropriate from maintenance perspectives [1, 30, 31]. Before using the NiCad outputs of Type 2 and Type 3 cases, we processed them in the following way.

(1) Every Type 2 clone class that exactly matched any Type 1 clone class was excluded from Type 2 outputs.

(2) Every Type 3 clone class that exactly matched any Type 1 or Type 2 clone class was excluded from Type 3 outputs.

We processed NiCad clone detection results in the mentioned ways because we wanted to investigate bug in three types of clones separately.

3.2. Bug-proneness detection technique

For each subject system, we first retrieve the commit messages by applying the “SVN log” command. A commit message describes the purpose of the corresponding commit operation. We automatically infer the commit messages using the heuristic proposed by Mockus and Votta [32] in order to identify those commits that occurred for the purpose of fixing bugs. Then we identify which of these bug-fix commits make changes to clone fragments. If one or more clone fragments are modified in a particular bug-fix commit, then it is an implication that the modification of those clone fragment(s) is necessary for fixing the corresponding bug. In other words, the clone fragment(s) are related to the bug. In this way we examine the commit operations of a candidate system, analyze the commit messages to retrieve the bug-fix commits, and identify those clone fragments that are related to the bug-fix. We also determine the number of changes that occurred to such a clone fragment in a bug-fix commit using the UNIX diff command.

The procedure that we follow to detect the bug-fix commits was also previously followed by Barbour et al. [4]. Barbour et al. [4] detected bug-fix commits in order to investigate whether late propagation in clones is related to bugs. They at first identified the occurrences of late propagations and then analyzed whether the clone fragments that experienced late propagations are related to a bug-fix. In our study we detect bug-fix commits in the same way as they detected, however, our study is different in the sense that we investigate the bugs of different types of code clones. Also, Barbour et al. [4] did not investigate the most important clone type, the Type 3. Generally, the number of Type 3 clones in a system is the highest among the three clone types. We consider Type 3 clones in our bug-fix study.

4. Experimental Results and Analysis

We mention our four research questions in Table 1. In this section we present our experimental results and analyze them to find the answers to our research questions.
4.1. **Answering the first research question (RQ 1)**

**RQ 1.** What percentages of files get affected because of clone and non-clone bug-fix commits?

**Motivation.** It is important to know the percentage of affected files due to bug-fixing commits and compare between clone and non-clone codes. More affected files means more changes in the system. More attention is needed when more changes occur. Knowing the information we can emphasize on which type of code (clone or non-clone) is affecting the system more.

**Methodology.** To answer this research question we automatically count the total number of files that contain clone code in three different types of clones (Type 1, Type 2, and Type 3) and total number of files containing non-clone code. Also, we detect the total number of files that contain changed clone code in three different clone types and the total number of files containing changed non-clone code. Then we calculate their percentages for individual clone types.

FC: This is the total number of files that have clone code. Columns with the heading FC in Table 4 represent the values.

FCC: This is the total number of files that contain changed clone code. These value are given in columns with the heading FCC in Table 4.

PFCC: We also calculate the overall percentage of files that have changed clone code using the following equation for all subject systems. In Table 4 columns with the heading PFCC show the values for various systems. Equation (1) shows the assessment of the percentages:

$$\text{PFCC} = \frac{100 \times \text{FCC}}{\text{FC}}.$$  \hspace{1cm} (1)

OPFCC: We also calculate the overall percentage of files that have changed clone code using the following equation:

$$\text{OPFCC}_{T_i}(\%) = \frac{100 \times \sum_{\text{all systems}} \text{FCC}_{T_i}}{\sum_{\text{all systems}} \text{FC}_{T_i}}.$$  \hspace{1cm} (2)

<table>
<thead>
<tr>
<th>Subject system</th>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FC(^a)</td>
<td>FCC(^b)</td>
<td>PFCC(^c) (%)</td>
</tr>
<tr>
<td>Ctags</td>
<td>12</td>
<td>4</td>
<td>33.33</td>
</tr>
<tr>
<td>Camellia</td>
<td>11</td>
<td>2</td>
<td>18.18</td>
</tr>
<tr>
<td>Brlcad</td>
<td>33</td>
<td>3</td>
<td>9.09</td>
</tr>
<tr>
<td>jEdit</td>
<td>21,338</td>
<td>117</td>
<td>0.54</td>
</tr>
<tr>
<td>Freecol</td>
<td>259</td>
<td>13</td>
<td>5.01</td>
</tr>
<tr>
<td>Carol</td>
<td>121</td>
<td>14</td>
<td>11.57</td>
</tr>
<tr>
<td>Jabref</td>
<td>97</td>
<td>8</td>
<td>8.24</td>
</tr>
</tbody>
</table>

\(^a\)FC = number of files that have clone code.

\(^b\)FCC = number of files that have changed clone code.

\(^c\)PFCC = percentage of the number of files that have changed clone code.

Table 4. Numbers of files that have changed clone code found in bug-fixing commits.
Here, $T_i$ represents different types of clones where $i = 1, 2,$ and $3$ in all subject systems.

Figure 1 shows the PFCC values and the OPFCC values for seven subject systems individually for each clone type. Here, we can see that Ctags, Camellia, and Brlcad have higher percentages than the rest of the subject systems (jEdit, Freecol, Carol, and Jabref). The jEdit has the lowest percentage among all. However, with respect to OPFCC, we observe that the percentage is decreasing, in Type 1, Type 2, and Type 3 clones, respectively. Table 4 describes the FC, FCC, and PFCC values for all the subject systems individually for each clone type (1, 2, and 3). We can see from this table that only Camellia has no bug-fix commits related to Type 2 clone.

Docking the total number of files containing clone code in bug-fix commits from the total number of files in bug-fix commits we get the total number of files that have non-clone code. For answering RQ 1, we identify the total number of files that have changes in the source code. From these files we identify the total number of files that have changes in clone code. The rest of the files made changes to non-clone code due to bug-fix commits.

FNC: This is the number of total files that have non-clone code. Columns with the heading FNC in Table 5 show the values.

FCNC: This is the number of total files which contain changes in non-clone code. To find out this file number we consider those files which contain changed non-clone code. We also check those files which have changed clone code in addition with non-clone code for various systems. All the columns of Table 5 with the heading FCNC show these values.
PFCNC: To calculate the percentage of the number of files containing changed non-clone code we use Eq. (3). All the columns with the heading PFCNC in Table 5 represent these values for various systems. This is shown in Eq. (3) below:

\[
PFCNC(\%) = \frac{100 \times FCNC}{FNC}.
\]  

OPFCNC: We calculate the overall percentage of files containing changed non-clone code using the following equation:

\[
OPFCNC(\%) = \frac{100 \times \sum_{all \ systems} FCNC}{\sum_{all \ systems} FNC}.
\]  

PFCNCs of different clone types for each subject system are shown in Fig. 1. Here, we can see that Camellia has the highest percentage than rest of the subject systems. On the other hand, jEdit has the lowest percentage among all subject systems. Table 5 shows the FNC, FCNC, and PFCNC values for all subject systems. We observe that percentage of changes due to bug-fix commits is higher in clone code than in non-clone code. This result was expected because total number of files containing non-clone code is much higher (almost three times) than clone code in every subject system.

The overall percentages (OPFCC) of files that have changes in Type 1 (12.28%) and Type 2 (8.99%) clones are more compared to that of files which have changes in Type 3 (5.63%) clones for bug-fix commits. Moreover, the percentage of files that have changes in clone code is higher than the percentage (OPFCNC) of files that have changes in non-clone code (1.52%).

**Mann-Whitney-Wilcoxon (MWW) tests for RQ 1.** We are interested to know whether the percentages of three clone types are significantly higher than that of non-clone code. First, we perform the MWW test [33] with percentages of Type 1 clone and non-clone code. We consider the significance level as 5% for this test. According to the data critical \(U\) is 13. If the \(p\)-value is less than 0.05 and \(U\) value is less than 13 then the result is significant. Our result shows that percentage of Type 1 clone code is significantly higher than that of non-clone code. For two-tailed test we find the \(p\)-value of 0.011719 which is much lower than 0.05. In the same way we

<table>
<thead>
<tr>
<th>Subject system</th>
<th>FNC(^a)</th>
<th>FCNC(^b)</th>
<th>PFCNC(^c) (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctags</td>
<td>12,318</td>
<td>120</td>
<td>0.97</td>
</tr>
<tr>
<td>Camellia</td>
<td>972</td>
<td>49</td>
<td>5.04</td>
</tr>
<tr>
<td>Brlcad</td>
<td>6978</td>
<td>104</td>
<td>1.49</td>
</tr>
<tr>
<td>jEdit</td>
<td>2801</td>
<td>15</td>
<td>0.53</td>
</tr>
<tr>
<td>Freecol</td>
<td>41,442</td>
<td>444</td>
<td>1.07</td>
</tr>
<tr>
<td>Carol</td>
<td>13,302</td>
<td>94</td>
<td>0.70</td>
</tr>
<tr>
<td>Jabref</td>
<td>39,389</td>
<td>333</td>
<td>0.84</td>
</tr>
</tbody>
</table>

\(^a\)FNC = number of files that have non-clone code.  
\(^b\)FCNC = number of files that have changed non-clone code.  
\(^c\)PFCNC = percentage of the number of files that have changed non-clone code.
perform the test for percentages of Type 2 clone code and Type 3 clone code with the non-clone code. We find the p-values of 0.015714 and 0.004574 for Type 2 clone and Type 3 clone, respectively. Both percentages of Type 2 clone and Type 3 clone codes are significantly higher than that of the non-clone code. Thus, we can say that percentages of all three types of clones are significantly higher than that of the non-clone code. We list our MWW test results in Table 6.

Table 6. The MWW test result for RQ 1.

<table>
<thead>
<tr>
<th>Clone type</th>
<th>p-value</th>
<th>U value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type 1</td>
<td>0.011719</td>
<td>8</td>
</tr>
<tr>
<td>Type 2</td>
<td>0.015714</td>
<td>9</td>
</tr>
<tr>
<td>Type 3</td>
<td>0.004574</td>
<td>5</td>
</tr>
</tbody>
</table>

*Considering the level of significance as 5%. For 5% two-tailed level, critical value of U is 13.

**Answer to RQ 1.** According to our experimental results, percentage of the number of file changes in bug-fix commits is higher in clone code than in non-clone code. Also, in terms of overall percentage of files Type 1 and Type 2 code clones (12.28% and 8.99%) have higher percentages than Type 3 code clone (5.63%).

We observe that percentages of files containing changes due to bug-fix commits are high in clone code than in non-clone code. However, we still do not know what percentages of clone and non-clone code get changed during bug-fix commits. Intuitively, percentages of bug-fix changes should be higher in clone code than in non-clone code. To understand this we investigate our next research question.

### 4.2. Answering the second research question (RQ 2)

**RQ 2.** How often do bug-fix changes occur to the clone and non-clone code?

**Motivation.** Though we have the answer of RQ 1 and hence we know the percentages of file changes in clone and non-clone code, but still we are not sure how much these changes are influencing the system. It is important to know the frequency of the bug-fix changes in both clone and non-clone code. From comparison between them we can understand the impact of bug-fix changes. Intuitively, more importance should be given to the more frequent one. This will help us to manage clone code.

**Methodology.** We know the total number of commits of each subject system. As discussed in Sec. 3.1, we report the total number of commits that have changes in clone fragments. To answer the RQ 2 we automatically count the total number of bug-fix commits that contain changes in clone code. First, we find out the total number of bug-fix commits as described in Sec. 3.2. We automatically count the
number of total bug-fix commits which have changed the clone code. We deduct this number from the total number of bug-fix commits and then find the total number of bug-fix commits which have changed the non-clone code. In the following way we calculate the occurrences of bug-fix changes in clone and non-clone code.

CC: This is the total number of commits that made changes to clone code. All the columns of Table 7 with the heading CC show these values for various systems.

BCC: This is the total number of bug-fix commits that made changes to clone code. Columns with the heading BCC in Table 7 show the values.

PBCC: Percentage of the bug-fix commits that made changes to clone code. We calculate this for each subject system and for three different types of clone, i.e. Type 1, Type 2, and Type 3 clone codes. All the columns with the heading PBCC in Table 7 represent these values for various systems.

We use the following equation for calculating the percentage:

\[
PBCC(\%) = \frac{100 \times BCC}{CC}.
\]  

We calculate the overall percentage of the bug-fix commits containing clone code using the following equation:

\[
OPBCC_T(\%) = \frac{100 \times \sum_{all \ systems} BCC_{T_i}}{\sum_{all \ systems} CC_{T_i}}.
\]  

Here, OPBCC$_T$ is the overall percentage of clone code found in the bug-fix commits with respect to $T_i$ type of clones ($i = 1, 2, 3$). Table 7 shows the values of CC, BCC and PBCC for seven subject systems and each type of clone individually. Figure 2 describes the PBCCs of all subject systems along with the OPBCCs. We can see that jEdit has the highest percentage (over 40%) and Brlcad has the lowest percentage (less than 15%). However, with respect to OPBCC Type 3 code clone has the highest percentage than Type 1 and Type 2 code clones.

<table>
<thead>
<tr>
<th>Subject system</th>
<th>Type 1</th>
<th>Type 2</th>
<th>Type 3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CC</td>
<td>BCC</td>
<td>PBCC (%)</td>
</tr>
<tr>
<td>Ctags</td>
<td>14</td>
<td>3</td>
<td>21.42</td>
</tr>
<tr>
<td>Camellia</td>
<td>8</td>
<td>1</td>
<td>12.5</td>
</tr>
<tr>
<td>Brlcad</td>
<td>32</td>
<td>2</td>
<td>6.25</td>
</tr>
<tr>
<td>jEdit</td>
<td>92</td>
<td>37</td>
<td>40.21</td>
</tr>
<tr>
<td>Freecol</td>
<td>35</td>
<td>7</td>
<td>20</td>
</tr>
<tr>
<td>Carol</td>
<td>41</td>
<td>8</td>
<td>19.51</td>
</tr>
<tr>
<td>Jabref</td>
<td>48</td>
<td>6</td>
<td>12.5</td>
</tr>
</tbody>
</table>

*aCC = number of commits affecting clone code.

*bBCC = number of bug-fix commits affecting clone code.

**PBCC = percentage of commits that were applied for fixing bugs in clone code.
We first identify the list of commits that made changes to the source code. From these commits we detect which commits made changes to clone code. The remaining commits in the list made changes to non-clone code. In the same way we deduct the total number of bug-fix commits containing changes in clone code from the total number of bug-fix commits to find the number of changes in non-clone code bug-fix commits. Applying these findings we answer RQ 2.

CNC: This is the total number of commits that made changes to non-clone code. The values for different systems are given in columns with the heading CNC in Table 8.

![Graph](image)

**Fig. 2.** Percentage of bug-fix commits that have changed clone and non-clone fragments.

We first identify the list of commits that made changes to the source code. From these commits we detect which commits made changes to clone code. The remaining commits in the list made changes to non-clone code. In the same way we deduct the total number of bug-fix commits containing changes in clone code from the total number of bug-fix commits to find the number of changes in non-clone code bug-fix commits. Applying these findings we answer RQ 2.

CNC: This is the total number of commits that made changes to non-clone code. The values for different systems are given in columns with the heading CNC in Table 8.

<table>
<thead>
<tr>
<th>Subject system</th>
<th>CNC*</th>
<th>BCNCb</th>
<th>PBCNCc (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctags</td>
<td>383</td>
<td>137</td>
<td>35.77</td>
</tr>
<tr>
<td>Camellia</td>
<td>133</td>
<td>24</td>
<td>18.04</td>
</tr>
<tr>
<td>Brlcad</td>
<td>589</td>
<td>88</td>
<td>14.94</td>
</tr>
<tr>
<td>jEdit</td>
<td>31</td>
<td>9</td>
<td>29.03</td>
</tr>
<tr>
<td>Freecol</td>
<td>672</td>
<td>326</td>
<td>48.51</td>
</tr>
<tr>
<td>Carol</td>
<td>323</td>
<td>65</td>
<td>20.12</td>
</tr>
<tr>
<td>Jabref</td>
<td>685</td>
<td>161</td>
<td>23.50</td>
</tr>
</tbody>
</table>

*^CNC = number of commits affecting non-clone code.
^BCNC = number of bug-fix commits affecting non-clone code.
^PBCNC = percentage of commits that were applied for fixing bugs in non-clone code.
BCNC: This is the total number of bug-fix commits that made changes to non-clone code. Columns with the heading BCNC in Table 8 represent these values.

PBCNC: Percentage of the bug-fix commits that made changes to non-clone code. In Table 8 columns with the heading PBCNC show these values.

Similar to Eq. (5) to compute the percentages, we use Eq. (7):

\[
PBCNC(\%) = \frac{100 \times BCNC}{CNC}.
\]  

To see the overall percentage of the number of bug-fix commits that are related to non-clone code we use the following equation which is similar to Eq. (6):

\[
OPBCNC(\%) = \frac{100 \times \sum_{all\ systems} BCNC}{\sum_{all\ systems} CNC}.
\]  

Here, OPBCNC is the overall percentage of the bug-fix commits that contain non-clone code. The CNC, BCNC and PBCNC values for all subject systems are shown in Table 8. Here, most of the percentage values range from 15% to 35% (only the percentage of Freecol is more than 45%). Figure 2 depicts the PBCNCs and the OPBCNC of seven subject systems. Here, we observe that the percentage of bug-fix commits containing changed non-clone code is the highest for Freecol system and it is lowest for Brlcd. Overall percentage is near 30% which is higher than the clone code. Though the bug-fix changes occur more in non-clone code than in clone code it is not noteworthy since the difference is not that much high.

The overall percentage (OPBCC) of Type 3 clone (22.32%) is higher than those of the clones of Type 1 (18.91%) and Type 2 (21.56%). Obviously, Type 1 clone (has the exact same code) and Type 2 clone (due to renaming of identifiers or changing data type of identifiers) have less changes than Type 3 clone code (has addition, deletion, or modification of code). We believe for this reason Type 3 clone code is affected more than Type 1 and Type 2 clone codes. Also, the overall percentage (OPBCNC) for non-clone code (27.13%) is higher than that of clone code.

The MWW tests for RQ 2. We perform MWW test [33] to understand whether the difference between the percentages of clone and non-clone codes is significant. Percentage of each type of clone is individually tested with that of non-clone code. We found the \(p\)-values of 0.092892, 0.207578, and 0.344562 for Type 1, Type 2, and Type 3 clones, respectively. Calculated \(U\) values for Type 1, Type 2, and Type 3 clone codes are 16, 20, and 23, respectively. Here, every \(p\)-value is greater than 0.05 (significance level is 5%) and every \(U\) value is greater than 13 (critical \(U\) value is 13). This indicates the result is insignificant. Insignificant result denotes percentage of bug-fix commits having changed non-clone code is slightly higher than that of clone code. Table 9 describes the \(p\)-values and \(U\) values for all three types of clone code.
Answer to RQ 2. Comparing the overall percentages of bug-fix commits containing clone and non-clone code, we found that frequency of bug-fix commits is slightly higher in non-clone code (27.13%) than in clone code (20.93%). Also, Type 3 clone code (22.32%) has the higher percentage of changes than Type 1 and Type 2 clone codes (18.91% and 21.56%) due to bug-fix commits.

We observe that bug-fix commits occur in non-clone code more often than clone code by 6.2%. From MWW test we find that the difference between percentages of clone and non-clone code is insignificant.

4.3. Answering the third research question (RQ 3)

RQ 3. Is there any difference between the severities of the bugs occurring in clone and non-clone code?

Motivation. In every single commit there is a message or comment written by the programmer which describes about the changes that they made from the previous commit. In case of bug-fix commits these messages describe about the bug that occur in the code-base of the system. By reading a bug-fix commit message, we can understand whether a bug is severe or not. This message is helpful for debugging and understanding the scenario of the situation. To understand the severity of bugs and compare between clone and non-clone codes, we automatically process the bug-fix commit messages followed by a manual inspection. It is important to give priority to more severe bugs while fixing them.

Why use commit messages. In each bug report there is an attribute named “priority” which indicates different levels of importance of that bug report. This attribute helps developers to decide which bug should be fixed first. We do not consider this attribute as the measurement unit of bug severity. This is possibly the easiest way to get an idea of the severity of a bug. However, we manually check that in most of the cases bug reporters do not assign this value rather they leave it as the default value which usually define “normal” level of severity. Thus, an incorrect evaluation will take place if we consider “priority” as the scale of the bug severity.
Saha et al. [45] proved in their study that we cannot assume the level of “priority” is always accurate. This paper [45] supports our decision about not to consider the “priority” of the bug report as the measurement of the severity. Considering this situation we decide to investigate commit messages instead of bug “priority” of the bug report.

**Methodology.** We perform NLP-based preprocessing on bug-fix commit messages to reduce the noise. There are numerous types of data preprocessing. Among them we use two types of NLP-based data preprocessing:

1. Tokenization (removing punctuation, special characters, and numbers).
2. Stop words removing.

On the preprocessed commit messages, we automatically perform a heuristic search proposed by Lamkanfi et al. [34] in the bug-fix commit messages to identify severe bugs. Then we manually investigate the results for validation. We consider five subject systems (Ctags, Camellia, Brlcad, jEdit, and Freecol) for this experiment.

For the non-clone bug-fix commits we choose some random bug-fix commits which do not contain clone code. It is not feasible to check all the non-clone bug-fix commits by manual inspection. Hence, we keep the total number of non-clone bug-fix commits equal to the total number of clone bug-fix commits to maintain the data impartiality.

Lamkanfi et al. [34] suggested most significant terms for different components indicating severe and non-severe bugs. For example, “fault,” “hang,” “freeze,” “deadlock,” etc. represent severe problems of the system. The words “favicon,” “deprec,” “mnemon,” “outbox,” etc. represent non-severe problems of the system. We take these terms as keywords to decide the severity of the bug. Though there are different levels of bug severity we consider only two categories for simplicity. That is whether the bug is severe or not, i.e. “TRUE” or “FALSE”. Here, “TRUE” means the bug is severe (i.e. bug-fix commit messages containing the terms which represent severity) and “FALSE” means the bug is non-severe (i.e. bug-fix commit messages containing the terms which represent non-severity). This is important for bug triaging process since severe bugs need more care and prompt fixing.

We calculate the percentage of severe bugs in bug-fix commits for both clone and non-clone codes. We observe that the existence of severe bugs in Camellia (both clone and non-clone), Brlcad (clone), and jEdit (non-clone) systems is zero (0%). We also observe that Freecol has the highest percentage (85.71% for clone code and 62.5% for non-clone code) of severe bugs in both clone and non-clone bug-fix commits compared to other subject systems. Percentages of severe bugs in rest of the subject systems range from 50% to 67%. Ctags (non-clone) and Brlcad (non-clone) have the lowest percentage (50%) of severe bugs. Overall, clone code has higher tendency of having severe bugs than non-clone code. The difference between the overall percentages of severe bugs of clone code and non-clone code bug-fix commits is 17.46% which is highly significant. These findings imply that more importance should be given on clone code while fixing bugs for better software maintenance.
Answer to RQ 3. After careful inspection of each commit message of the bug-fix commits for both clone and non-clone code, we found that clone code bug-fix commits have higher percentage of severe bugs (overall percentage 71.63%) than the non-clone code (overall percentage 54.17%). This proves that occurrence of severe bugs is higher in clone code compared with non-clone code.

We observe that severity of bugs is higher in clone code than in non-clone code, though we find that some of the bug-fix commit messages are very short and it is not enough to describe the severity of the bug. Considering this constraint of the messages in bug-fixing commits, the result may vary in different cases. However, severe bugs should have the highest priority in software maintenance.

4.4. Answering the fourth research question (RQ 4)

RQ 4. Which types of severe and non-severe bugs can occur in code clones?

Motivation. After answering RQ 3 we know that the occurrence of severe bugs is higher in clone code than in non-clone code. We wanted to collect more details on this to get an insight of those severe bugs which occurred in clone code. We investigate which type of severe bugs have high frequency of occurrence. This information will help to find out which type of severe bugs are causing more inconvenience to the systems so that the developers can be more careful about handling those type of severe bugs while fixing them.

Methodology. In Sec. 4.3 we described the procedures of identifying severe and non-severe bugs. To answer our fourth research question first we categorize all the bugs occurring in clone code. We consider five subject systems (Ctags, Camellia, Brlcad, jEdit, and Freecol) for this experiment. We investigate the source code of each bug-fixing commit manually, totaling 337 bug-fix commits. We categorize the types of changes of source code due to these bug-fixing commits. Later we map severe and non-severe bugs with these categories of code changes. In our previous study [22] we showed this categorization. However, we did not investigate the severity of bugs in that paper.

Table 10 shows the results of our manual investigation of severe and non-severe bugs in clone code. Here, we observe that most of the severe bugs occurred due to addition and modification of if-condition. We suggest developers to handle more carefully while copying or modifying an if-condition. On the other hand, the highest frequency was found for non-severe bugs in modification of if-condition. The second highest frequency for non-severe bugs was found for “Modification of parameters in the called method.” The other frequent change types shown in Table 10 are addition and deletion of method call, deletion of if-else blocks, and replacement of old method.
call by new method call. There are also some infrequent change types such as: replacement of C preprocessor by method call, addition, deletion, or modification of loops.

We also observe that Type 3 clone code contains the highest number of severe bugs and Type 2 clone code contains a few severe bugs. Type 1 clone code does not contain any severe bugs. On the other hand, all the three types of clone code contain non-severe bugs. Among them Type 1 and Type 2 clone codes contain the highest number of non-severe bugs than Type 3 clone code. This summarizes that we should emphasize on Type 3 clone code while performing clone management operations.

Answer to RQ 4. According to our manual investigation, code clones containing if-condition and if-else blocks have a higher risk of severe bugs. These types of changes in clone code should be made carefully during software maintenance.

We observe the types of changes for fixing both severe and non-severe bugs in code clones. It is important to consider these types of changes during clone management. This will reduce software maintenance cost and effort in future.

5. Related Works

Sajnani et al. [23] performed a comparative study between clone and non-clone codes for different bug patterns. They used bug reports generated by a tool, i.e. FindBugs, whereas we worked on real bug reports that were reported by developers. In our previous study [22], it has been shown that cloning is responsible for replicating bugs. However, it does not show any comparison with non-clone code in that study.

Bug-proneness of code clones has been investigated by a number of existing studies. Li and Ernst [18] performed an empirical study on the bug-proneness of clones by investigating four software systems and developed a tool called CBCD on the basis of their findings. CBCD can detect clones of a given piece of buggy code. Li et al. [35] developed a tool called CP-Miner which is capable of detecting bugs related to inconsistencies in copy-paste activities. Steidl and Göde [19] investigated

Table 10. Most frequent change types of severe and non-severe bugs in code clones.

<table>
<thead>
<tr>
<th>Change type</th>
<th>Severe</th>
<th>Non-severe</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Addition of if-else blocks</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>2 Modification of if-condition</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>3 Deletion of if-else blocks</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4 Modification of parameters in the called method</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>5 Addition of method call</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6 Replacement of old method call by new method call</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7 Deletion of method call</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
finding instances of incompletely fixed bugs in near-miss code clones by inquiring a broad range of features of such clones involving machine learning. Göde and Koschke [5] investigated the occurrences of unintentional inconsistencies to the code clones of three mature software systems and found that around 14.8% of all changes that occurred to the code clones were unintentionally inconsistent. Chatterji et al. [39] performed a user study to investigate how clone information can help programmers localize bugs in software systems. Jiang et al. [20] performed a study on the context-based inconsistencies related to clones. They developed an algorithm to mine such inconsistencies for the purpose of locating bugs. Using their algorithm they could detect previously unknown bugs from two open-source subject systems. Inoue et al. [40] developed a tool called “CloneInspector” in order to identify bugs related to inconsistent changes to the identifiers in the clone fragments. They applied their tool on a mobile software system and found a number of instances of such bugs. Xie et al. [41] investigated fault-proneness of Type 3 clones in three open-source software systems. They investigated two evolutionary phenomena on clones: (1) mutation of the type of a clone fragment during evolution and (2) migration of clone fragments across repositories and found that mutation of clone fragments to Type 2 or Type 3 clones is risky.

None of the studies discussed above investigated bug-fix commits in code clones and non-clone code simultaneously. Mondal et al. [21] investigated bug-proneness of code clones. While the primary target of that study was to compare the bug-proneness of three clone types, our target is to compare the bug-proneness of clone and non-clone codes. Mondal et al. [21] did not investigate the bug-proneness of non-clone code in their study.

Rahman et al. [42] found that bug-proneness of cloned code is less than that of non-cloned code on the basis of their investigation on the evolution history of four subject systems using DECKARD [43] clone detector. However, they considered monthly snapshots (i.e. revisions) of their systems and thus, they have the possibility of missing buggy commits. They have calculated and found that on an average 3.3% of bugs have late propagation fixing with different staging snapshots. In our study, we consider all the snapshots/revisions (i.e. without discarding any revisions) of a subject system from the beginning one. Thus, we believe that we are not missing any bug-fix commits. Moreover, our goal in this study is different. We investigate and compare the impacts of bug-fix commits of different types of code clones whereas they only focused on the bug-proneness of clones. Selim et al. [44] used Cox hazard models in order to assess the impacts of cloned code on software defects. They found that defect-proneness of code clones is system-dependent. However, they considered only method clones in their study. We consider block clones in our study. While they investigated only two subject systems, we consider seven diverse subject systems in our investigation. Also, we investigate the bug-fix possibilities of different types of clones. Selim et al. [44] did not perform a type-centric analysis in their study.

A number of studies have also been done on the late propagation in clones and its relationships with bugs. Aversano et al. [3] investigated clone evolution in two
subject systems and reported that late propagation in clones is directly related to bugs. Barbour et al. [4] investigated eight different patterns of late propagation considering Type 1 and Type 2 clones of three subject systems and identified those patterns that are likely to introduce bugs and inconsistencies to the code-base.

Focusing on this, we perform an in-depth investigation on bug’s impacts in code clones and non-clones in this research. Our experimental results are promising and provide useful implications for better understanding of the bug-proneness of clone and non-clone code.

6. Threats to Validity

We used the NiCad clone detector [24] for detecting clones. While all clone detection tools suffer from the confounding configuration choice problem [36] and might give different results for different settings of the tools, the settings that we used for NiCad for this experiment are considered standard [37] and with these settings NiCad can detect clones with high precision and recall [27–29]. Thus, we believe that our findings on the bug-proneness of code clones are of significant importance.

Our research involves the detection of bug-fix commits. The way we detect such commits is similar to the technique proposed by Mockus and Votta [32] and also used by Barbour et al. [38]. The technique proposed by Mockus and Votta [32] can sometimes select a nonbug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [38] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

The number of total subject systems is not enough in our research to be able to generalize our findings regarding the comparative bug-fix changes of different types of clones. However, our candidate systems were of diverse variety in terms of application domains, sizes, and revisions. Thus, we believe that our findings are important from the perspectives of clone and non-clone codes.

7. Conclusion

In this paper we conduct an in-depth comparative study of software bugs in both clone and non-clone codes. For clone code we also consider three major types of clones, Type 1, Type 2, and Type 3. We investigated thousands of revisions of seven diverse subject systems. We also investigated bug-fix commit messages to measure the frequency of severe bugs in clone and non-clone codes. From our examination, changes to files due to bug-fix commits are higher for clone code than for non-clone code. Additionally, changes to files due to bug-fix commits happen more in Type 1 and Type 2 code clones than in Type 3 code clones. In addition, percentage of severe bugs is higher in clone code than in non-clone code bug-fix commits. Finally, our manual investigation shows that certain types of code construct in code clones have a higher risk of containing severe bugs. These are if-condition and if-else blocks. Changes to these types of code constructs in clone code should be prioritized during
software evolution and maintenance. We believe that our findings on bug-fix commits are valuable for better understanding of clone management such as ranking of clone code and software maintenance.

References