A Comparative Study of Software Bugs in Clone and Non-Clone Code

Judith F. Islam Manishankar Mondal Chanchal K. Roy Kevin A. Schneider
Department of Computer Science, University of Saskatchewan, Canada
{judith.islam, mshankar.mondal, chanchal.roy, kevin.schneider}@usask.ca

***Abstract*—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 on 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. Finally, the possibility of severe bugs occurring is higher in clone code than in non-clone code. Bug-fixing changes affecting clone code should be considered more carefully. According to our findings, clone code appears to be more bug-prone than non-clone code.**

I. 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] [1]

A significant number of studies [3]–[21] 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]–[15], [17], [19]–[21] of negative impacts of code clones too. These negative impacts include higher instability [15], late propagation [4], and unintentional inconsistencies [5]. Existing studies [4], [22] 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 [23] provides evidence of bug-replication in clone code. Also, Sajnani et al. [24] showed that cloned code has less problematic bug patterns than non-cloned code. They have used bugs reported by FindBugs 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 that paper [24], 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 code, 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 [25] 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 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 non-clone code. We validate our findings using the Mann-Whitney-Wilcoxon (MWW) test for three types of clones with non-clone code.

We investigate the three research questions listed in Table I. 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 changes in Type 1 and Type 2 clone code is higher than changes in Type 3 clone code. This findings can be used for ranking of clone code. Also, the occurrence of severe bugs is more in clone code than non-clone code.

The rest of the paper is organized as follows. Section II contains the terminology, Section III discusses the experimental steps, Section IV answers the research questions by presenting and analyzing the experimental results, Section V discusses the related work, Section VI discusses possible threats to validity, and Section VII concludes the paper and discusses possible future work.

II. Terminology

We conduct our analysis considering both exact (Type 1) and near-miss clones (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

DOI reference number: 10.18293/SEKE2017-056

TABLE I
RESEARCH QUESTIONS

SL	Research Question
RQ 1	What percentage of files get affected because of clone and non-clone bug-fix commits?
RQ 2	How often do bug-fix changes occur to the clone and non-clone code?
RQ 3	Is there any difference between the severity of the bugs occurring in clone and non-clone code?

TABLE II
SUBJECT SYSTEMS

Systems	Lang.	Domains	LLR	Revisions
Ctags	C	Code Def. Generator	33,270	774
Camellia	C	Image Processing Library	89,063	170
Brlcad	C	Solid Modeling CAD	39,309	735
jEdit	Java	Text Editor	191,804	4000
Freecol	Java	Game	91,626	1950
Carol	Java	Game	25,091	1700
Jabref	Java	Reference Management	45,515	1545

LLR = LOC in the Last Revision

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.

III. 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 used by our other studies. These subject systems are listed in Table II which were downloaded from the SourceForge online SVN repository [26]. 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.

A. Preliminary Steps

We perform the following steps for detecting fixed bugs: **(1)** Extraction of all revisions (as stated in Table II) of each of the subject systems from the online SVN repository; **(2)** Detection and extraction of code clones from each revision by applying NiCad [25] clone detector; **(3)** Detection of changes between every two consecutive revisions using diff; **(4)** Locating these changes to the already detected clones of the corresponding revisions; and **(5)** Detection of bug-fix commit operations. For completing the first four steps we use the tool SPCP-Miner [27]. We will describe the detection of bug-fix commits later in this section. In Section IV we will describe how we detect bug-fix changes in clone and non-clone code.

We use NiCad [25] for detecting clones since it can detect all major types (Type 1, Type 2, and Type 3) of clones with high precision and recall [28], [29]. Using NiCad we detect block clones including both exact (Type 1) and near-miss (Type 2, 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 III. For different settings of a clone detector the 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 [30] Svajlenko and Roy 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], [31], [32]. Before using the NiCad outputs of Type 2 and Type 3 cases, we processed them in the following way.

TABLE III
NICAD SETTINGS FOR THREE TYPES OF CLONES

Clone Types	Identifier Renaming	Dissimilarity Threshold
Type 1	none	0%
Type 2	blindrename	0%
Type 3	blindrename	20%

(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 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.

B. 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 [33] 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) was 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.

IV. Experimental Results and Analysis

We mention our three research questions in Table I. In this section we present our experimental results and analyze them to find the answers to our research questions.

A. Answering the first research question (RQ 1)

RQ 1: *What percentage 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 code. 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) are 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 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 IV represents the value.

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

PFCC: To find out the percentage of the number of files that have changed clone code we use following equation for all subject systems. In Table IV columns with the heading PFCC show this value. Equation 1 shows the assessment of the percentages.

$$PFCC = \frac{100 \times FCC}{FC} \tag{1}$$

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

$$OPFCC_{T_i} = \frac{100 \times \sum_{all\ systems} FCC_{T_i}}{\sum_{all\ systems} FC_{T_i}} \tag{2}$$

Here, T_i represents different types of clones where i = 1, 2 and 3 in all subject systems.

Figure 1 shows the percentage PFCC and the overall percentage OPFCC for seven subject systems individually for each clone type. Here, we can see that Ctags, Camellia and Brlcad have the higher percentage than the rest of the subject systems (jEdit, Freecol, Carol and Jabref). jEdit has the lowest percentage among all. However, in overall percentage we observe that percentage is decreasing in Type 1, Type 2 and Type 3 clone respectively. Table IV describes the FC, FCC and PFCC 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 clone Type 2.

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 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 V 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. All the columns of Table V with the heading FCNC show this value.

PFCNC: To calculate percentage of the number of files containing changed non-clone code we use following equation. All the columns with the heading PFCNC in Table V represent this value. This is shown in equation 3.

$$PFCNC = \frac{100 \times FCNC}{FNC} \tag{3}$$

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

$$OPFCNC = \frac{100 \times \sum_{all\ systems} FCNC}{\sum_{all\ systems} FNC} \tag{4}$$

PFCNCs of different clone types for each subject system are shown in Figure 1. Here, we can see that Camellia has

TABLE V
Number of Files that have Changed Non-Clone Code in Bug-fixing Commits

Subject Systems	FNC	FCNC	PFCNC
Ctags	12318	120	0.97%
Camellia	972	49	5.04%
Brlcad	6978	104	1.49%
jEdit	2801	15	0.53%
Freecol	41442	444	1.07%
Carol	13302	94	0.70%
Jabref	39389	333	0.84%

FNC = Number of Files that have Non-Clone code
FCNC = Number of Files that have Changed Non-Clone code
PFCNC = Percentage of the Number of Files that have Changed Non-Clone code

TABLE IV
NUMBER OF FILES THAT HAVE CHANGED CLONE CODE FOUND IN BUG-FIXING COMMITS

	Type 1			Type 2			Type 3		
Subject Systems	**FC**	**FCC**	**PFCC**	**FC**	**FCC**	**PFCC**	**FC**	**FCC**	**PFCC**
Ctags	12	4	33.33%	25	4	16%	117	12	10.25%
Camellia	11	2	18.18%	0	0	0%	56	6	10.71%
Brlcad	33	3	9.09%	5	1	20%	66	5	7.57%
jEdit	21338	117	0.54%	695	14	2.01%	12655	137	1.08%
Freecol	259	13	5.01%	178	11	6.17%	3425	83	2.42%
Carol	121	14	11.57%	138	14	10.14%	991	51	5.14%
Jabref	97	8	8.24%	81	7	8.64%	1143	26	2.27%

FC = Number of Files that have Clone code
FCC = Number of Files that have Changed Clone code
PFCC = Percentage of the Number of Files that have Changed Clone code

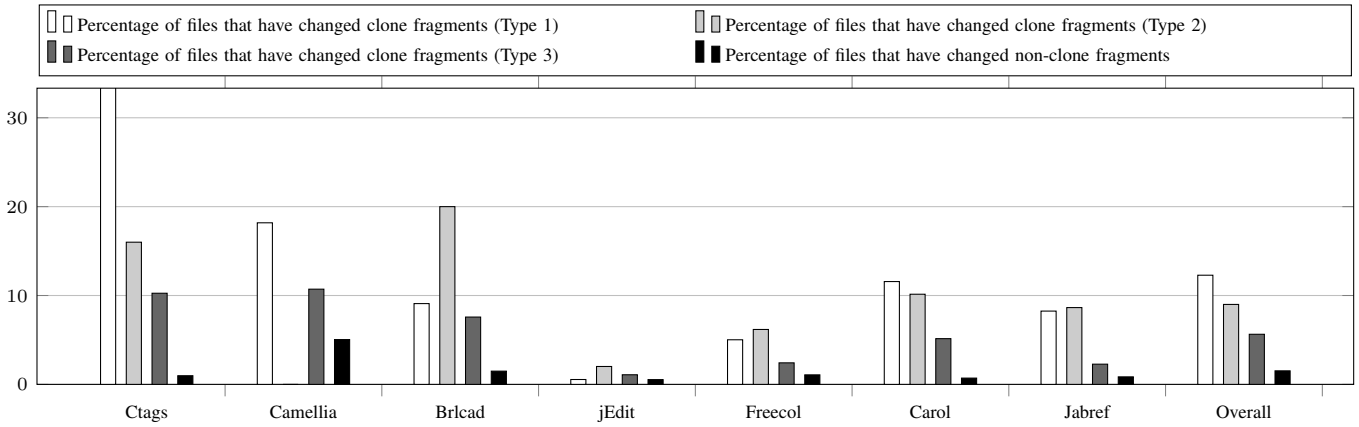

Fig. 1. Percentage of the number of files that have changed clone and non-clone fragments.

the highest percentage than rest of the subject systems. On the other hand, jEdit has the lowest percentage among all subject systems. Table V shows FNC, FCNC and PFCNC for all subject systems. We observe that percentage of changes due to bug-fix commits is higher in clone code than 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 have more changes in files than 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 is significantly higher than non-clone code. First, we perform Mann-Whitney-Wilcoxon (MWW) test [34] with percentages of Type 1 clone and non-clone code. We consider the significance level is 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 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 perform the test for percentages of Type 2 clone code and Type 3 clone code with the non-clone code. We find the p-value 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 code are significantly higher than non-clone code. Thus, we can say that percentages of all three types of clones are significantly higher than non-clone code. We list our MWW test results in Table VI.

TABLE VI
MANN-WHITNEY-WILCOXON TEST RESULT FOR RQ1

Clone Types	**p-value**	**U value**
Type 1	0.011719	8
Type 2	0.015714	9
Type 3	0.004574	5

Considering level of significance is 5%.
For 5% two-tailed level, Critical value of U is 13

Answer to RQ 1. According to our experimental results percentage of number of file changes in bug-fix commits is higher in clone code than 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 percentage than Type 3 code clone (5.63%).

We observe that percentages of files containing changes

due to bug-fix commits is high in clone code than non-clone code. However, we still do not know what percentage 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 non-clone code. To understand this we investigate our next research question.

B. 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 percentage of files 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 Section III-A, 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 Section III-B. We automatically count the number of total bug-fix commits which have changed clone code. We deduct this number from the total number of bug-fix commits and found the total number of bug-fix commits which have changed 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 VII with the heading CC show this value.

BCC: This is the total Number of Bug-fix Commits that made changes to Clone code. Columns with the heading BCC in Table VII show the values.

PBCC: Percentage of the Bug-fix Commits that made changes to Clone code. We calculate this for each subject systems and for three different types of clone i.e. Type 1, Type 2 and Type 3 clone code. All the columns with the heading PBCC in Table VII represent this value.

We use the following equation for calculating the percentage.

$$PBCC = \frac{100 \times BCC}{CC} \tag{5}$$

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

$$OPBCC_{T_i} = \frac{100 \times \sum_{all\ systems} BCC_{T_i}}{\sum_{all\ systems} CC_{T_i}} \tag{6}$$

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

TABLE VIII
NUMBER OF BUG-FIX COMMITS AFFECTING NON-CLONE CODE

Subject Systems	CNC	BCNC	PBCNC
Ctags	383	137	35.77%
Camellia	133	24	18.04%
Brlcad	589	88	14.94%
jEdit	31	9	29.03%
Freecol	672	326	48.51%
Carol	323	65	20.12%
Jabref	685	161	23.50%

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

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. This value is given in columns with the heading CNC in Table VIII.

BCNC: This is the total Number of Bug-fix Commits that made changes to Non-clone code. Columns with the heading BCNC in Table VIII represent the value.

PBCNC: Percentage of the Bug-fix Commits that made changes to Non-clone code. In Table VIII columns with the heading PBCNC show this value.

Likewise equation 5 to compute the percentages we use equation 7.

$$PBCNC = \frac{100 \times BCNC}{CNC} \tag{7}$$

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

$$OPBCNC = \frac{100 \times \sum_{all\ systems} BCNC}{\sum_{all\ systems} CNC} \tag{8}$$

Here, $OPBCNC$ is the overall percentage of the bug-fix commits that contain non-clone code. The CNC, BCNC and PBCNC for all subject systems are shown in Table VIII. Here, most of the percentage ranges from 15% to 35% (only the percentage of Freecol has more than 45%). Figure 2 depicts the percentage PBCNCs and overall percentage OPBCNC of

TABLE VII
NUMBER OF BUG-FIX COMMITS AFFECTING CLONE CODE

	Type 1			Type 2			Type 3		
Subject Systems	**CC**	**BCC**	**PBCC**	**CC**	**BCC**	**PBCC**	**CC**	**BCC**	**PBCC**
Ctags	14	3	21.42%	25	4	16%	59	11	18.64%
Camellia	8	1	12.5%	5	1	20%	26	5	19.23%
Brlcad	32	2	6.25%	7	1	14.28%	47	5	10.63%
jEdit	92	37	40.21%	24	10	41.66%	99	42	42.42%
Freecol	35	7	20%	36	10	27.77%	152	46	30.26%
Carol	41	8	19.51%	44	8	18.18%	112	22	19.64%
Jabref	48	6	12.5%	46	6	13.04%	149	23	15.43%

CC = Number of Commits affecting Clone code
BCC = Number of Bug-fix Commits affecting Clone code
PBCC = Percentage of Commits that were applied for fixing Bugs in Clone code

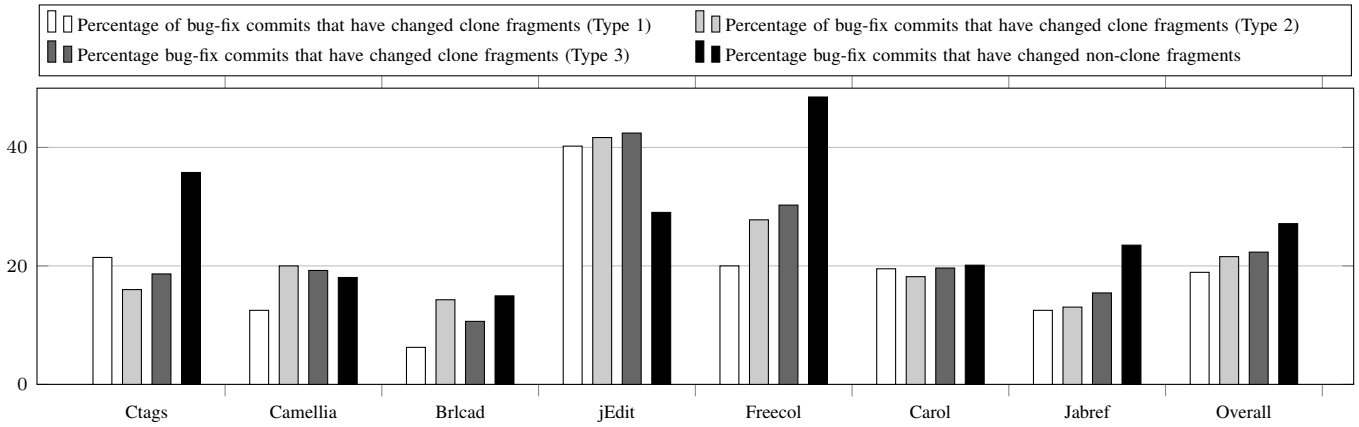

Fig. 2. Percentage of bug-fix commits that have changed clone and non-clone fragments.

seven subject systems. Here, we observe that percentage of the bug-fix commits containing changed non-clone code is highest in Freecol system and lowest in Brlcad. Overall percentage is near 30% which is higher than clone code. Though the bug-fix changes occur more in non-clone code than clone code it is not noteworthy since the difference is not that much high.

The overall percentage (OPBCC) of Type 3 (22.32%) clone is higher than Type 1 (18.91%) and Type 2 (21.56%) clone. Obviously, Type 1 clone (has exact same code) and Type2 clone (has 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 code. Also, the overall percentage (OPBCNC) for non-clone code (27.13%) is higher than clone code.

Mann-Whitney-Wilcoxon (MWW) tests for RQ 2. We perform MWW test [34] to understand whether the difference between the percentages of clone and non-clone code is significant. Percentage of each type of clone is individually tested with non-clone code. We found the p-value of 0.092892, 0.207578 and 0.344562 for Type 1, Type 2 and Type 3 clone respectively. Calculated U value for Type 1, Type 2 and Type 3 clone code is 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 clone code. Table IX describes the p-value and U value for all three types of clone code.

TABLE IX
MANN-WHITNEY-WILCOXON TEST RESULT FOR RQ2

Clone Types	**p-value**	**U value**
Type 1	0.092892	16
Type 2	0.207578	20
Type 3	0.344562	23

Considering level of significance is 5%.
For 5% two-tailed level, Critical value of U is 13

Answer to RQ 2. Comparing the overall percentage 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 clone code (20.93%). Also, Type 3 clone code (22.32%) has the higher percentage of changes than Type 1 and 2 clone code (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.

C. Answering the third research question (RQ 3)

RQ 3: *Is there any difference between the severity 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 sever 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 code we automatically process the bug-fix commit massages followed by a manually inspection. It is important to give priority to more severe bugs while fixing them.

Methodology. We automatically perform a heuristic search proposed by Lamkanfi et al. [35] in the bug-fix commit massages to identify severe bugs. Then we manually investigate the results for validation. We consider five subject systems (Ctags, Camellia, Brlcad, jEdit, Freecol) for this experiment. For the non-clone bug-fix commits we choose some random bug-fix commits which does 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. [35] suggested most significant terms for different components indicating severe and non-severe bugs. For example, 'fault', 'hang', 'freez', '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 the simplicity. That is whether the bug is severe or not i. e. 'TRUE' or 'FALSE'. Here, 'TRUE' means the bug is sever (i.e. bug-fix commit messages containing the terms which represent severity) and 'FALSE' means the bug is non-sever (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 code. 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 highest percentage (85.71% for clone 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 are 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 and non-clone code bug-fix commits is 17.46% which is highly significant. This 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 messages of the bug-fix commits for both clone and non-clone code we found that clone code bug-fix commits has 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 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.

V. Related Work

Shajnani et al. [24] performed a comparative study between clone and non-clone code 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 [23] 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 [19] 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. [36] developed a tool called CP-Miner which is capable of detecting bugs related to inconsistencies in copy-paste activities. Steidl and Göde [20] investigated finding instances of incompletely fixed bugs in near-miss code clones by investigating 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.

None of the studies discussed above investigated bug-fix commits in code clones and non-clone code simultaneously. Mondal et al. [22] 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 code. Mondal et al. [22] did not investigate the bug-proneness of non-clone code in their study. 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.

VI. Threats to Validity

We used the NiCad clone detector [25] for detecting clones. While all clone detection tools suffer from the *confounding configuration choice problem* [37] and might give different results for different settings of the tools, the setting that we used for NiCad for this experiment are considered standard [38] and with these settings NiCad can detect clones with high precision and recall [28]–[30]. 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 Mocus and Votta [33] and also used by Barbour et al. [39]. The technique proposed by Mocus and Votta [33] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [39] showed that this probability is very low. According to their investigation, the technique has an accuracy of 87% in detecting bug-fix commits.

VII. Conclusion

In this paper we conduct an in-depth comparative study of software bugs in both clone and non-clone code. 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 code. From our examination, changes to files due to bug-fix commits is higher for clone code than for non-clone code. Additionally, changes to files due to bug-fix commits happens 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 non-clone code bug-fix commits. We believe that our findings on bug-fix commits are valuable for better understanding of clone management such as ranking of clone codes and software maintenance.

References

[1] C. K. Roy, M. F. Zibran, and R. Koschke, "The Vision of Software Clone Management: Past, Present, and Future (Keynote paper)," in *Proc. CSMR-WCRE*, 2014, pp. 18–33.

[2] C. K. Roy, "Detection and analysis of near-miss software clones," in *Proc. ICSM*, 2009, pp. 447–450.

[3] L. Aversano, L. Cerulo, and M. D. Penta, "How clones are maintained: An empirical study," in *Proc. CSMR*, 2007, pp. 81–90.

[4] L. Barbour, F. Khomh, and Y. Zou, "Late Propagation in Software Clones," in *Proc. ICSM*, 2011, pp. 273–282.

[5] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proc. ICSE*, 2011, pp. 311–320.

[6] N. Göde and J. Harder, "Clone Stability," in *Proc. CSMR*, 2011, pp. 65–74.

[7] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, "Is Duplicate Code More Frequently Modified than Non-duplicate Code in Software Evolution?: An Empirical Study on Open Source Software," in *Proc. EVOL/IWPSE*, 2010, pp. 73–82.

[8] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do Code Clones Matter?" in *Proc. ICSE*, 2009, pp. 485–495.

[9] C. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software," in *Proc. Empirical Software Engineering*, 2008, pp. 13(6): 645–692.

[10] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *Proc. WCRE*, 2007, pp. 170–178.

[11] J. Krinke, "Is cloned code more stable than non-cloned code?" in *Proc. SCAM*, 2008, pp. 57–66.

[12] J. Krinke, "Is Cloned Code older than Non-Cloned Code?" in *Proc. IWSC*, 2011, pp. 28–33.

[13] A. Lozano and M. Wermelinger, "Tracking clones' imprint," in *Proc. IWSC*, 2010, pp. 65–72.

[14] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Proc. ICSM*, 2008, pp. 227–236.

[15] M. Mondal, C. K. Roy, M. S. Rahman, R. K. Saha, J. Krinke, and K. A. Schneider, "Comparative Stability of Cloned and Non-cloned Code: An Empirical Study," in *Proc. SAC*, 2012, pp. 1227–1234.

[16] M. Mondal, M.S. Rahman, R.K. Saha, C.K. Roy, J. Krinke and K.A. Schneider, "An Empirical Study of the Impacts of Clones in Software Maintenance," in *Proc. ICPC*, 2011, pp. 242-245.

[17] M. Mondal, C. K. Roy, and K. A. Schneider, "An Empirical Study on Clone Stability," in *ACM SIGAPP Applied Computing Review*, 2012, pp. 12(3): 20–36.

[18] S. Thummalapenta, L. Cerulo, L. Aversano, and M. D. Penta, "An empirical study on the maintenance of source code clones," in *Empirical Software Engineering*, 2009, pp. 15(1): 1–34.

[19] J. Li and M. D. Ernst, "CBCD: Cloned Buggy Code Detector," in *Proc. ICSE*, 2012, pp. 310–320.

[20] D. Steidl and N. Göde, "Feature-Based Detection of Bugs in Clones," in *Proc. IWSC*, 2013, pp. 76–82.

[21] L. Jiang, Z. Su, and E. Chiu, "Context-Based Detection of Clone-Related Bugs," in *Proc. ESEC-FSE*, 2007, pp. 55–64.

[22] M. Mondal, C. K. Roy, and K. A. Schneider, "A Comparative Study on the Bug-Proneness of Different Types of Code Clones," in *Proc. ICSME*, 2015, pp. 91–100.

[23] J. F. Islam, M. Mondal, and C. K. Roy, "Bug Replication in Code Clones: An Empirical Study," in *Proc. SANER*, 2016.

[24] H. Sajnani, V. Saini, and C. V. Lopes, "A Comparative Study of Bug Patterns in Java Cloned and Non-cloned Code," in *Proc. SCAM*, 2014, pp. 21–30.

[25] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *Proc. ICPC Tool Demo*, 2011, pp. 219–220.

[26] SVN repository. [Online]. Available: http://sourceforge.net/

[27] M. Mondal, C. K. Roy, and K. A. Schneider, "Spcp-miner: A Tool for Mining Code Clones that are Important for Refactoring or Tracking," in *Proc. SANER*, 2015, pp. 484–488.

[28] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach," in *Science of Computer Programming*, 2009, pp. 74 (2009): 470–495.

[29] C. K. Roy and J. R. Cordy, "A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools," in *Proc. Mutation*, 2009, pp. 157–166.

[30] J. Svajlenko and C. K. Roy, "Evaluating Modern Clone Detection Tools," in *Proc. ICSME*, 2014, pp. 321–330.

[31] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," in *Proc. Technical Report 2007-541*, School of Computing, Queen's University, 2007, 115 pp.

[32] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy and M. M. Mia, "Towards a Big Data Curated Benchmark of Inter-Project Code Clones", in *Proc. ICSME*, 2014, pp. 476-480.

[33] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes using Historic Databases," in *Proc. ICSM*, 2000, pp. 120–130.

[34] Mann-Whitney U Test. [Online]. Available: goo.gl/JIFmo3

[35] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the Severity of a Reported Bug," in *Proc. MSR*, 2010, pp. 1–10.

[36] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code," in *Proc. OSDI*, 2004, pp. 20–20.

[37] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for Better Configurations: A Rigorous Approach to Clone Evaluation," in *Proc. ESEC/SIGSOFT FSE*, 2013, pp. 455–465.

[38] C. K. Roy and J. R. Cordy, "NICAD: Accurate Detection of Near-miss Intentional Clones Using Flexible Pretty-printing and Code Normalization," in *Proc. ICPC*, 2008, pp. 172–181.

[39] L. Barbour, F. Khomh, and Y. Zou, "An empirical study of faults in late propagation clone genealogies," in *Proc. Journal of Software: Evolution and Process*, 2013, pp. 25(11):1139–1165.