A Comparative Study of Software Bugs in Micro-clones and Regular Code Clones

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

***Abstract*—Reusing a code fragment through copy/pasting, also known as code cloning, is a common practice during software development and maintenance. Most of the existing studies on code clones ignore micro-clones where the size of a micro-clone fragment can be 1 to 4 LOC. In this paper we compare the bug-proneness of micro-clones with that of regular code clones. From thousands of revisions of six diverse open-source subject systems written in three languages (C, C#, and Java), we identify and investigate both regular and micro-clones that are associated with reported bugs.**

Our experiment reveals that percentage of changed code fragments due to bug-fix commits is significantly higher in micro-clones than regular clones. The number of consistent changes due to bug-fix commits is significantly higher in micro-clones than regular clones. We also observe that significantly higher percentage of files get affected by bug-fix commits in micro-clones than regular clones. Finally, we found that percentage of severe bugs is significantly higher in micro-clones than regular clones. We perform Mann-Whitney-Wilcoxon (MWW) test to evaluate the statistical significance level of our experimental results. Our findings imply that micro-clones should be emphasized during clone management and software maintenance.

***Index Terms*—Code Clones, Micro-Clones, Software Bugs**

I. Introduction

Recurrent activities of copy-pasting code fragments is very common in everyday life of software development cycle. The act of copying a piece of code and then pasting it without any modification (exactly similar) or with modifications (nearly similar) is known as code cloning [1], [2]. A group of similar code fragments constructs a clone class. Code clones are mainly created because of the frequent copy/paste activities of programmers during software development and maintenance. Beside copy/paste activities there can be various other reasons behind creating clone code [3]. Whatever may be reasons behind cloning, code clones are significantly important from the perspectives of software maintenance and evolution [1].

A good number of studies [4]–[21] have been conducted on discovering the impact of cloning on software maintenance. This is a big dilemma in clone code research for the last two decades regarding whether clone code is good or bad. Reusing code can save software development time as well as costs and efforts of development. A number of studies [4], [7], [8], [10]–[13] have revealed some positive sides of code cloning. On the other hand, if a code fragment contain a bug, copy/pasting that buggy code fragment can cause severe code maintenance issue. Code cloning is often referred as 'bad smell' due to its bad impacts on software systems. In literature there is strong empirical evidence [5], [6], [9], [14]–[17], [19]–[21] of negative impacts of code clones. These negative impacts include higher instability [16], late propagation [5], and unintentional inconsistencies [6]. Existing studies [5], [22] show that code clones are related to bugs in the code-base.

During the last two decades code clone research was based on detecting, refactoring, tracking and managing code clones. Existing studies ignored code clones of small size i.e. 1 to 4 LOC stating that these clones are mostly unpromising. In a very recent study, Beller et al. [23] investigated these small code clones and denoted those as *micro-clones*. Also, Tonder et al. [24] worked on automatically detecting and removing micro-clones in large scale. In another study, Mondal et al. [25] focused on the importance of micro-clones and found that during software maintenance and evolution 80% of all consistent updates occur in micro-clones. They have shown that the number of micro-clones is very high in software systems than regular clones. However, they did not investigate bug-proneness of micro-clones. This motivates us to investigate the buggy code in both micro and regular code clones. In order to explore the effects of bugs in micro-clones and regular code clones we perform a comparative study. To the best of our knowledge, our research is the first comparative study on bug-proneness in between micro-clones and regular code clones.

We consider thousands of revisions of six diverse open source software systems written in three languages, Java, C# and C. For detecting code clones from each of the revisions of a subject system we use the NiCad clone detector [26]. We analyze the evolution history of both micro and regular code clones, and investigate whether they contain bugs and to what extent.

To investigate bug-proneness of regular and micro-clones, we observe bug-fix commits reported by the developers from thousands of commits in open source projects. The major findings of our research is as follows.

- The percentage of bug-fix changes occurring in micro-clones is significantly higher than the percentage of bug-fix changes in regular code clones.
- The number of consistent bug-fix changes is considerably higher in micro-clones than regular code clones. We found that total number of consistent bug-fix changes in micro-clones (4,118) is almost 6 times higher than regular code clones (728).
- Percentage of affected files due to bug-fix changes is significantly higher in micro-clones than regular clones.

TABLE I
RESEARCH QUESTIONS

SL	Research Question
RQ 1	Do micro-clones contain more bug-fix changes than regular code clones?
RQ 2	Are the bug-fix changes consistent in micro and regular code clones?
RQ 3	What percentage of files get affected for fixing bugs in micro and regular clones?
RQ 4	Do micro-clones contain more severe bugs compared to regular code clones?

TABLE II
SUBJECT SYSTEMS

Systems	Lang.	Domains	LLR	Revisions
Ctags	C	Code Def. Generator	33,270	774
Brlcad	C	Solid Modeling CAD	39,309	735
MonoOSC	C#	Formats and Protocols	18,991	355
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				

For micro-clones this percentage is 39.15%, whereas for regular clones the percentage is only 8.02%.

- Micro-clones can contain a significantly higher percentage of severe bugs compared to regular clones. From our inspection we found that micro-clones contain 16.98% more severe bugs than the regular code clones.

From these findings we can state that micro-clones draw our attention for clone management purpose, because they exhibit a high bug-proneness during evolution. In order to get rid of the negative impacts of micro-clones, we should track them for updating consistently. However, the existing clone trackers only consider regular code clones for tracking. Thus, it is important to investigate these clone trackers with a goal of making them capable of tracking micro-clones.

The rest of the paper is organized as follows. Section II contains the terminology, Section III discusses the experimental steps, Section IV answers our research questions by presenting and analyzing the experimental results, Section V describes a rigorous manual analysis on micro-clones. Section VI discusses the related work and compares with our study, Section VII discusses possible threats to validity, and Section VIII concludes the paper and discusses possible future work.

II. TERMINOLOGY

A. Types of Clones

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 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.

B. Micro Clones

Micro-clones are smaller in size than the clone size of regular code clones. According to the literature [23]–[25], micro-clones can be of 4 LOC at most. The minimum size of a micro-clone fragment can be 1 LOC. In this paper we ignore those micro-clones which are part of regular clones. Thus we consider only true or pure micro-clones.

C. 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 commits that occur to fix reported bugs are known as bug-fix commits. To fix these bugs, changes may occur in regular code clones or micro-clones. We observe these changes in regular clone and micro-clone to contrast between them in terms of their bug-proneness.

D. Severe Bugs

Severe bugs are the software defects which can make negative impact on the quality of software. This negative impact varies from critical to trivial level. Developers can define level of bug severity depending on the criteria of bugs while reporting a bug in open source projects.

III. EXPERIMENT STEPS

We conduct our research on six subject systems (two C, one C# and three Java systems). We consider these six subject systems since these systems have variations in application domains, sizes, and revisions. These subject systems are listed in Table II which were downloaded from the SourceForge online SVN repository [27]. 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. Figure 1 shows the simple flow diagram of our work procedure for this study.

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 [26] 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

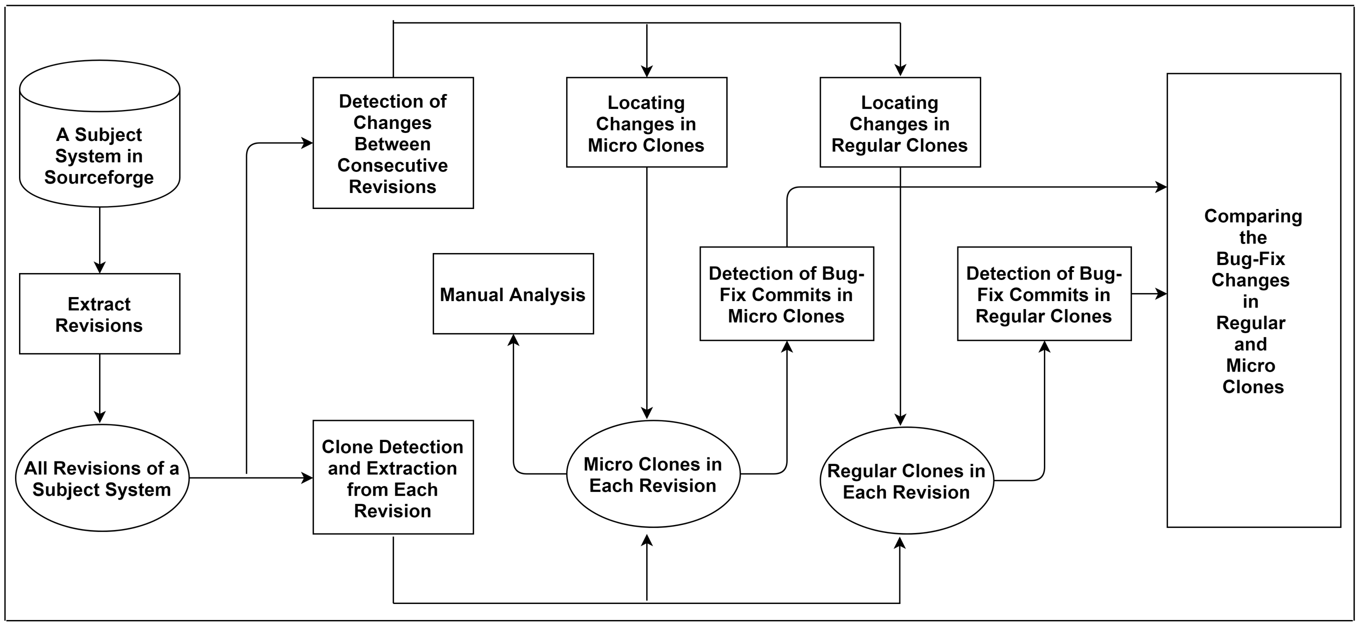

A Subject System in Sourceforge
Extract Revisions
All Revisions of a Subject System
Detection of Changes Between Consecutive Revisions
Clone Detection and Extraction from Each Revision
Locating Changes in Micro Clones
Manual Analysis
Micro Clones in Each Revision
Detection of Bug-Fix Commits in Micro Clones
Locating Changes in Regular Clones
Regular Clones in Each Revision
Detection of Bug-Fix Commits in Regular Clones
Comparing the Bug-Fix Changes in Regular and Micro Clones

Fig. 1. The execution flow diagram of the experimental steps in detecting bug-fix commits in Regular and Micro clones. The rectangles demonstrate the steps.

[28]. We perform steps 1 to 5 for both regular and micro-clones. 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 regular code clones and micro-clones.

We use NiCad [26] for detecting clones since it can detect code clones with high precision and recall [29], [30]. Using NiCad, we detect block clones (both exact and near-miss) of at least 10 lines with 20% dissimilarity threshold and blind renaming of identifiers. For micro-clones using NiCad we detect block clones of a minimum size of 1 LOC and maximum size of 4 LOC with 20% dissimilarity threshold and blind renaming of identifiers as was detected by Mondal et al. [25]. 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 for detecting regular clones, because Svajlenko and Roy [31] show that these settings provide us with better clone detection results in terms of both precision and recall.

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 [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) 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. [5]. Barbour et al. [5] 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 regular and micro-clones. Figure 1 summarizes all the steps of our experiment.

IV. Experiment Results and Analysis

Our research questions are listed in Table I. We represent answers to these research questions and analyze our experimental results in this section.

A. Answering the first research question (RQ 1)

RQ 1: *Do micro-clones contain more bug-fix changes than regular code clones?*

Motivation. Finding the number of changes due to fixing a bug in code clone is an important parameter for comparison between regular and micro-clones. Intuitively, micro-clones might have higher number of bug-fix changes than regular

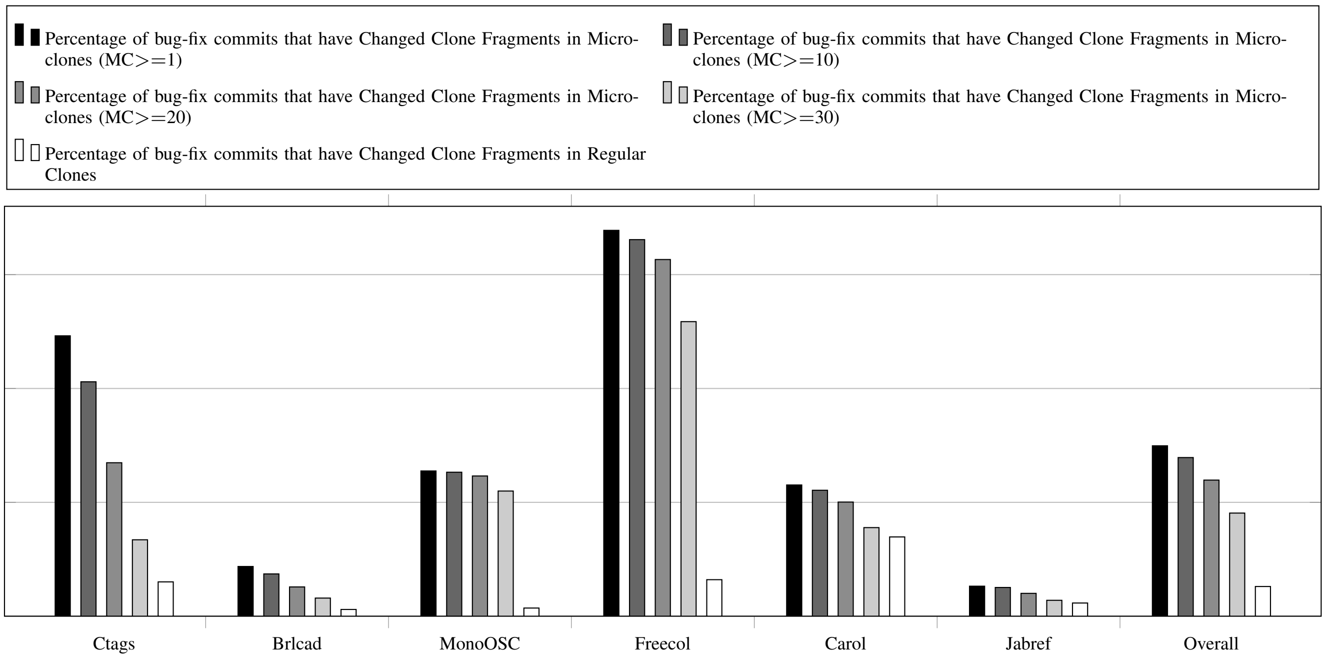


Fig. 2. Percentage of bug-fix commits that have changed clone fragments in regular and micro-clones.

clones since the number of micro-clones is higher in every system. If code clones of a particular type (micro or regular) contain more bug-fix changes then we should be more careful for that kind of clone while performing software maintenance tasks.

Methodology. To answer this research question we automatically count the number of bug-fix commits for both regular and micro-clones. We found the result as we expected i.e. number of bug-fix commits is higher in micro-clones than regular clones. Table III shows the experimental result for our first research question. While finding the bug-fix changes in micro-clones, we restrain the minimum number of characters per clone line. We refer it as Minimum number of Characters per clone line or MC in this paper. We choose four different threshold values for MC i.e. MC = 1, 10, 20, and 30. We predicted that number of bug-fix commits would increase with a decrease in MC value. We found the result as we have expected. This is obvious because when we choose MC>= 1, all the changes will be included. When we choose MC>= 30, only clone lines containing more than 29 characters will be counted. For regular code clones we choose the default value of MC>= 1. Our goal is to observe that even if we restrict the minimum number of characters per clone line for micro-clones, if there is any chance to get less number of bug-fix changes in micro-clones than regular code clones.

Figure 2 shows the percentages of bug-fix commits that have changed clone fragments in regular and micro-clones for six subject systems. We observe that for every subject system percentage of bug-fix commits is higher in micro-clones than regular clones. Also, if we increase the threshold of minimum characters per clone line (MC), the percentage decreases. This proves that even if we limit the minimum number of characters per clone line to equal or greater than 30 (MC>= 30), still the number of bug-fix changes is higher in micro-clones than regular code clones. Overall, we can see that percentages of bug-fix commits is much lower in regular clones than micro-clones.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 1. To check the statistical significance of our experiment results we perform Mann-Whitney-Wilcoxon (MWW) test [33], [34]. We want to observe whether the percentage of commits that have changes due to fixing a bug is significantly higher in micro-clones than regular clones. MWW test is a non-parametric and does not require normal distribution of data. We consider significance level of 5% for the test. We investigate six subject systems and for each of them we calculate percentage of regular clones and micro-clones. For micro-clones we calculate four percentages of bug-fix commits considering four micro-clone sizes i.e. MC>=1, 10, 20, and 30, where MC refers to the minimum number of characters per clone line. Thus, we compare the percentage from each set of micro-clones with that of regular clones. We found that percentages of bug-fix changes in micro-clones is significantly higher than regular clones. Table IV shows that for different MC values p-values are less than 0.05 i.e. the significance level that we consider for our tests. From the data we can see that the critical value of U is 7 and for each case our value of U is less than or equal to 7. This data prevails strongly that the number of bug-fix commits is significantly higher in micro-clones than in regular code clones.

TABLE III
NUMBER OF BUG-FIX COMMITS THAT HAVE CHANGED CLONE FRAGMENTS IN MICRO-CLONES AND REGULAR CLONES

Subject Systems	**Regular Clones**	**Micro-clones**			
		1	**10**	**20**	**30**
Ctags	53	433	362	237	118
Brlcad	29	214	182	126	78
MonoOSC	26	463	459	447	399
Freecol	358	3783	3692	3497	2888
Carol	450	745	715	648	503
Jabref	112	255	244	194	135
MC = Minimum number of Characters per clone line Here, we set MC = 1, 10, 20, 30					

TABLE IV
MANN-WHITNEY-WILCOXON TEST RESULT FOR RQ1

Micro-Clones	**p-value**	**U value**
MC>= 1	0.01539	4
MC>= 10	0.01539	4
MC>= 20	0.03288	6
MC>= 30	0.04648	7
Considering level of significance is 5%. For 5% one-tailed level, Critical value of U is 7		

Answer to RQ 1. From our investigation and analysis we found that percentage of bug-fix commits containing changed clone fragments is significantly higher in micro-clones (from 4.52% to 7.48%) than regular clones (1.30%) in software systems.

From the above results of our first research question we can state that micro-clones experience changes in a significantly higher number of bug-fix commits compared to regular clones. This reveals that we should emphasize on micro-clones during code clone management and software maintenance.

B. Answering the second research question (RQ 2)

RQ 2: *Are the bug-fix changes consistent in micro and regular code clones?*

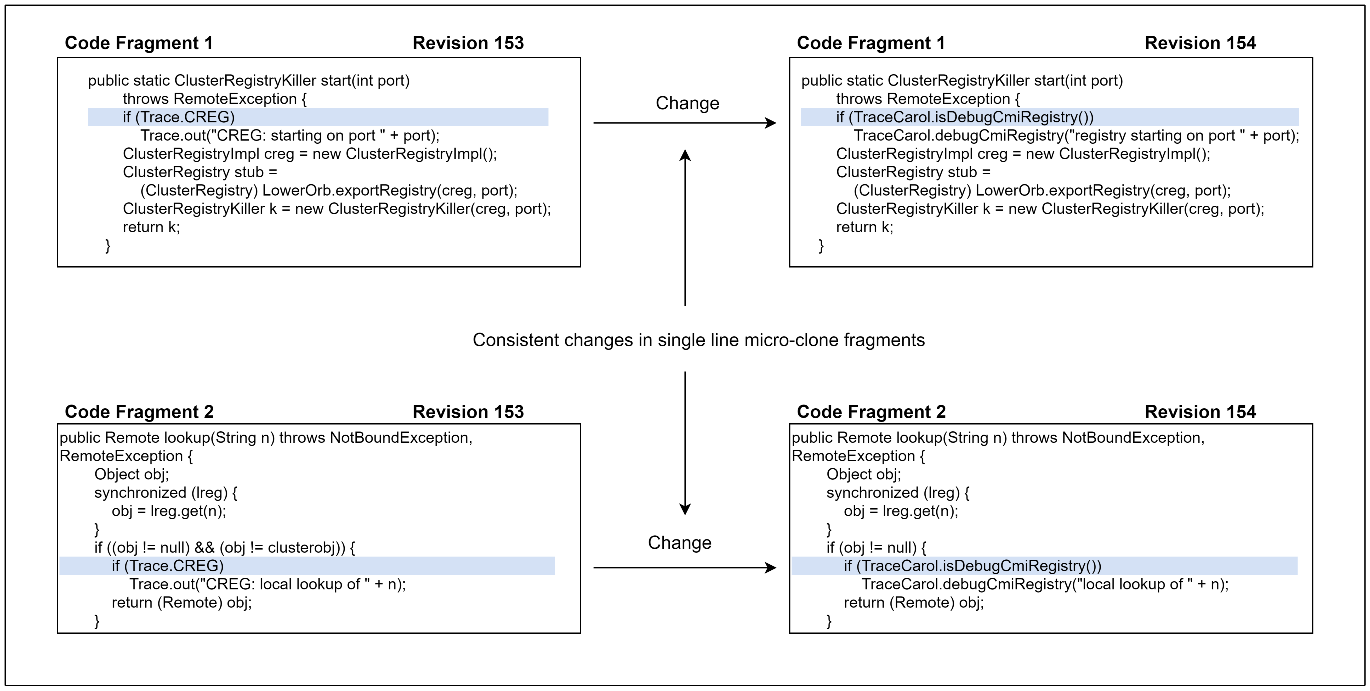
Motivation. From our first research question, we have found that the bug-fix commits occur more in micro-clones than regular clones in systems. However, we do not know whether these bug-fix commits are committing consistent changes or inconsistent changes in clone fragments. Consistent bug-fix change means more than one clone fragment from the same group experienced the same bug-fix. If the same bug-fix change need to be propagated to different clone fragments in the same group, then it indicates that the bug was replicated in different clone fragments. Replication of bug [35] is a severe negative impact of code cloning. In RQ 2, we investigate whether the intensity of consistent bug-fix changes is higher in micro-clones or in regular code clones.

Methodology. In our first research question we find all the changes that occurred due to fixing a bug i.e. number of bug-fix commits in both regular clones and micro-clones. To answer our second research question, we find out consistent bug-fix changes in regular and micro-clones. When similar changes occur in two or more clone fragments from the same clone class, we consider those changes as consistent changes. For instance, suppose CF1 and CF2 are two clone fragments in revision R. After the commit operation on revision R i.e. in revision R+1, the clone fragments CF1 and CF2 changes to CF1' and CF2' respectively. If the changes between clone fragments CF1 and CF1' are similar to the changes between clone fragments CF2 and CF2', then we can consider the changes to be consistent. Figure 3 shows an example of consistent changes in micro-clones. Here, we can see that a single line micro-clone has been changed consistently. This example has been mined from our subject system Carol written in Java. In the commit operation on revision 153, a similar change occurred in revision 154 for both code fragments.

Table V shows the data of consistent changes of clones found in regular and micro-clones for six subject systems. Here, we can see that the total number of consistent bug-fix changes is higher in micro-clones than regular code clones for every subject system.

Figure 4 shows the percentage of consistently changed bug-fix commits in regular and micro-clones for six subject systems along with overall values. Here, we observe that for three systems, Ctags, MonoOSC, Freecol, percentage of consistent changes is higher in micro-clones than regular code clones. On the other hand, for rest of the systems, i.e. Brlcad, Carol, Jabref, percentage of consistent changes is higher in regular code clones than micro-clones. Overall, the percentage of consistent changes is approximately equal in both regular and mirco-clones. Though the result shows that there is no significant difference in the percentage of consistent changes, we believe that the actual number of consistent bug-fix changes is more important. All the bugs need to be fixed for ensuring consistency of the software system. We found that the total number of consistent bug-fix changes in regular clones is 728. On the other hand, the total number of consistent changes due to bug-fix commits in micro-clones is 4,118. For the purpose of clone refactoring or clone tracking, we have to deal with the actual number of clones. Thus, we need to emphasize on micro-clones which have higher number of clones as well as consistent changes of bug-fix commits. To evaluate the significance of the difference between regular and micro-clones, we do the MWW statistical test.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 2. To investigate statistical significance we perform Mann-Whitney-Wilcoxon test [33], [34] on the results. We found that for two-tailed, 5% significant level and critical value of U is 5, the difference between two percentages of consistent bug-fix changes in regular and micro-clones is not statistically significant. The MWW test shows that the p-value is 0.93624 which is greater than 0.05 and the value of U found 17 which is greater than the critical value of U i.e. 5. Though the result



```
public static ClusterRegistryKiller start(int port)
        throws RemoteException {
        if (Trace.CREG)
            Trace.out("CREG: starting on port " + port);
        ClusterRegistryImpl creg = new ClusterRegistryImpl();
        ClusterRegistry stub =
            (ClusterRegistry) LowerOrb.exportRegistry(creg, port);
        ClusterRegistryKiller k = new ClusterRegistryKiller(creg, port);
        return k;
    }
```

```
public static ClusterRegistryKiller start(int port)
        throws RemoteException {
        if (TraceCarol.isDebugCmiRegistry())
            TraceCarol.debugCmiRegistry("registry starting on port " + port);
        ClusterRegistryImpl creg = new ClusterRegistryImpl();
        ClusterRegistry stub =
            (ClusterRegistry) LowerOrb.exportRegistry(creg, port);
        ClusterRegistryKiller k = new ClusterRegistryKiller(creg, port);
        return k;
    }
```

```
public Remote lookup(String n) throws NotBoundException,
RemoteException {
        Object obj;
        synchronized (lreg) {
            obj = lreg.get(n);
        }
        if ((obj != null) && (obj != clusterobj)) {
            if (Trace.CREG)
                Trace.out("CREG: local lookup of " + n);
            return (Remote) obj;
        }
```

```
public Remote lookup(String n) throws NotBoundException,
RemoteException {
        Object obj;
        synchronized (lreg) {
            obj = lreg.get(n);
        }
        if (obj != null) {
            if (TraceCarol.isDebugCmiRegistry())
                TraceCarol.debugCmiRegistry("local lookup of " + n);
            return (Remote) obj;
        }
```

Fig. 3. This is an example of consistent changes in micro-clones. Code Fragment 1 and Code Fragment 2 contain a pair of single line micro-clone from our subject system Carol. Consistent changes in single line micro-clone are shown in between revision 153 and revision 154. We highlight the single line micro-clone in both code fragments and for both revisions. Here, similar changes occur at the commit operation on revision 153. We also observe that the surrounding code of one micro-clone fragment is not similar to that of the other micro-clone fragment.

TABLE V
NUMBER OF CONSISTENT CHANGES IN REGULAR AND MICRO-CLONES

Subject Systems	Regular Clones	Consistent Regular Clones	Micro Clones	Consistent Micro Clones
Ctags	53	21	433	184
Brlcad	29	16	214	114
MonoOSC	26	15	463	357
Freecol	358	252	3783	2835
Carol	450	355	745	532
Jabref	112	69	255	96

shows that there is no significant difference in the percentage of consistent changes, we should emphasize on the number of consistent bug-fix changes as we stated above. In terms of consistency, we have to observe the number of clones rather than percentages. We found that the average number of consistently changed bug-fix commits is 121.33 for regular clones and for micro-clones it is 686.33. Hence, the number of consistent changes in micro-clones is higher than that of regular code clones.

Answer to RQ 2. After investigating the bug-fix changes in clones we found that number of consistent bug-fix changes is significantly higher in micro-clones than in regular clones. The difference between two average numbers is 565.

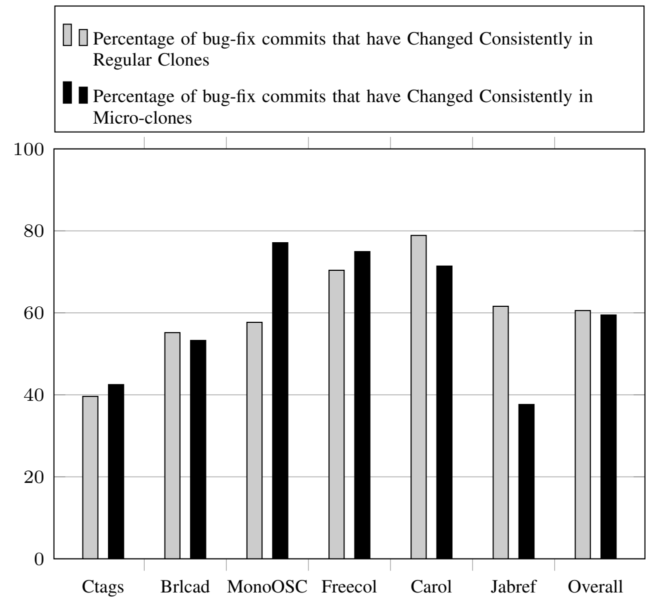


Fig. 4. Percentage of bug-fix commits that have consistent changes of clones in regular and micro-clones.

From the above observation we can state that number of consistent bug-fix changes are more in micro-clones than regular clones. This finding is important for refactoring and tracking of code clones.

TABLE VI
NUMBER OF FILES THAT CHANGES IN REGULAR AND MICRO-CLONES

Subject Systems	**Total Number of Files**	**Regular Clones**	**Micro Clones**
Ctags	116	9	41
Brlcad	153	3	36
MonoOSC	132	5	32
Freecol	310	52	269
Carol	367	49	186
Jabref	377	17	54

C. Answering the third research question (RQ 3)

RQ 3: *What percentage of files get affected for fixing bugs in micro and regular clones?*

Motivation. It is important to answer this research question because the impact of bug-fix commits on micro and regular clones is revealed through this answer. Intuitively, a higher percentage of affected files indicates a higher number of changes in the system. More attention is needed when more changes occur. Answer to this research question implies which type of code clones (regular or micro clone) are affecting the system more. Knowing the information we can emphasize on that particular type of code clone while maintaining and managing code clones.

Methodology. To answer our third research question first we find out total number of files those have been changed after a bug-fix commit during software evolution. Then we find the total number of files which get affected due to fixing a bug in regular clones and micro-clones respectively. Thus, we calculate the percentage of files that get affected due to bug-fix commits. Table VI shows the result of our investigation on RQ3. Here, we can see that number of affected files due to bug-fix commits is higher in micro-clones than regular code clones. Figure 5 illustrates the percentage of files that get affected due to bug-fix commits for six subject systems. In this figure we can see that for each subject system, percentage of affected files is higher in micro-clones than regular clones. The difference in percentage between micro and regular clones is the highest in Freecol subject system. Overall, percentage of affected files due to bug-fix commit in regular clone is 8.02% and for micro-clone the percentage is 39.15%. To understand the statistical significance of the difference between regular and micro-clones, we perform following MWW test for this research question.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 3. We perform MWW test [33], [34] to observe the statistical significance level of our experimental result. For a two tailed and significance level 0.05, we get the critical value of U is 5. MWW test results show that our value of U is 1 which is less than 5. Also, we found the p-value is 0.0083 which is much less than significance level 0.05. Thus, we find that percentage of files that get affected by bug-fix commits is significantly higher in micro-clones than regular code clones.

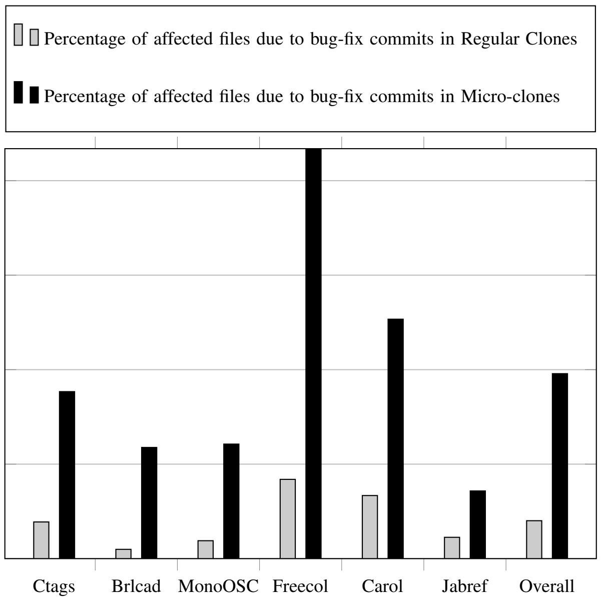


Fig. 5. Percentage of files that get affected by bug-fix commits in regular and micro-clones.

TABLE VII
MANN-WHITNEY-WILCOXON TEST RESULT FOR RQ2, RQ3 AND RQ4

Research Question No.	**p-value**	**U value**	**Critical Value of U**
RQ2	0.93624	17	5
RQ3	0.0083	1	5
RQ4	0.04846	11	11
Considering level of significance is 5%. For RQ3 and RQ4, U value <= Critical value of U			

Answer to RQ 3. According to our investigation we found that percentage of files that get affected due to fixing a bug in the software systems is significantly higher in micro-clones than regular code clones.

During software maintenance it is important to manage clones intelligently. For this purpose finding which type of code clones are important is a crucial issue. From our above experiment we can state that micro-clones are more important than regular clones while managing code clones in software systems.

D. Answering the fourth research question (RQ 4)

RQ 4: *Do micro-clones contain more severe bugs compared to regular code clones?*

Motivation. From our previous research questions we tried to find out percentage of changes, consistent changes and affected files due to fixing a bug during software evolution. Still we need to understand which type of clone is a severe threat in our systems while software maintenance. Severe bugs can be more harmful than non-severe bugs. Fixing a severe bug

TABLE VIII
NUMBER OF SEVERE BUGS IN REGULAR AND MICRO-CLONES

Subject Systems	**Regular Clones**	**Severe Bugs in Regular Clones**	**Micro Clones**	**Severe Bugs in Micro Clones**
Ctags	53	12	433	113
Brlcad	29	0	214	9
MonoOSC	26	0	463	394
Freecol	358	65	3783	607
Carol	450	46	745	91
Jabref	112	9	255	44

is an emergency task compared to fixing a non-severe bug. For this purpose we observe and compare the severity of bugs in both regular and micro-clones.

Methodology. To distinguish between severe and non-severe bugs in regular and micro-clones we automatically perform a heuristic search in bug-fix commit messages of regular and micro clone code. Lamkanfi et al. [36] proposed a list of keywords which identify severe and non-severe bugs from textual information. We search these keywords in bug fixing commit messages to mine the severe bug-fixing revisions. There are different levels or categories of severity in a bug report. However, we consider only two levels of severity i.e. true or false for the simplicity of our experiment. Table VIII shows the result of our heuristic search. Here, we can see that number of severe bugs is low in regular clones compared with micro-clones. In fact, two subject systems, Brlcad and MonoOSC contain no severe bugs in their system for regular code clones. Highest number of severe bug-fixing commit found in Freecol where the number of micro-clones is also very high. Figure 6 depicts the percentage of severe bugs in six subject systems for both regular and micro-clones. For each of the subject systems percentage of severe bugs is high in micro-clones than regular clones except for Freecol. Overall, micro-clones contain 26.82% of severe bugs and regular clones contain 9.84% of severe bugs.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 4. To understand whether the difference between two percentages is significant or not we perform MWW test [33], [34]. For significance level of 5% and one-tailed MWW test, we find the critical value of U is 11. After performing MWW test on our data set of severe bugs for regular and micro-clones, it shows that U value is 11 which is equal to the critical value of U i.e. 11. Also, the p-value of our data set is 0.04846 which is less than 0.05 i.e. significance level. This proves that the percentage of severe bug-fix commits is significantly higher in micro-clones than regular clones. Table VII shows the MWW test results for our second, third and fourth research questions.

Answer to RQ 4. After observing the severity of the bug-fix commits in both regular and micro-clones, we found that percentage of severe bugs is significantly higher in micro-clones than regular clones.

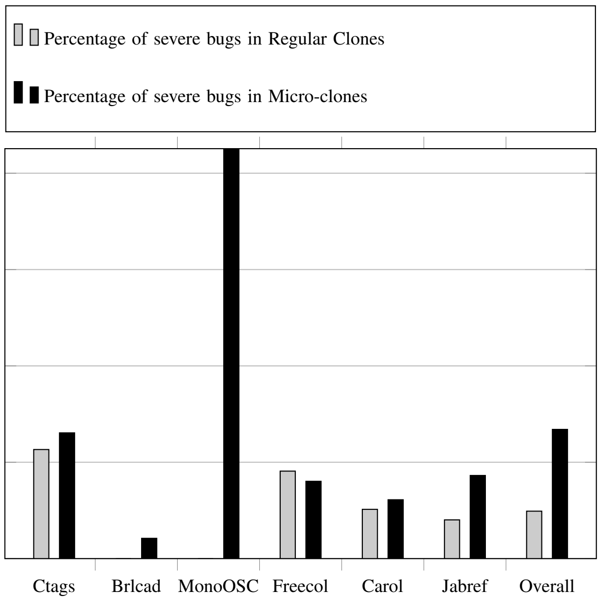


Fig. 6. Percentage of severe bugs in bug-fix commits in regular and micro-clones.

Since severe bugs need to be fixed immediately, it is a vital issue for fixing a severe bug during software maintenance. From the above investigation, we can state that micro-clones contain more severe bugs than regular clones for which we have to deal micro-clones carefully than regular clones while software maintenance.

V. MANUAL ANALYSIS

We manually investigate the types of bug-fix changes that occurred in micro-clones. Knowing this information is helpful for programmers so that he or she can be more careful while copying the code fragments containing those statements. The result of our manual investigation will be helpful for preventing bug-proneness in code bases in advance. Since, manually investigating all the bug-fix changes is not feasible, we inspect first 50 distinct changes from each of our subject systems. Thus we observe 50 X 6 = 300 distinct changes in total. For each change observation the average time is 10 minutes. Hence, for six subject systems and first fifty distinct changes from each of them, we spent approximately (50 X 6 X 10) = 3000 minutes = 50 hours. Moreover, there were some cases which implies one bug-fix change falls into more than one change types. In this case, we choose most appropriate one. We categorize the bug-fix changes and then count them for each category. In total we define 37 categories of bugs. To visualize the changes in bug-fixing micro-clones instantly, we use online difference checker in code [37].

Table IX shows the list of most frequent change types in bug-fix changes in micro-clones. Here, we can see that "Deletion of Statement" is the highest in number i.e. 70 in total and for subject system MonoOSC has the highest number of this type of change i.e. 23. Second and third frequent changes are "Addition of Statement" and "Modification of Statement" i.e.

TABLE IX
MOST FREQUENT CHANGE-TYPES DURING FIXING BUGS IN MICRO CLONES

	Change Types	**Ctags**	**Brlcad**	**MonoOSC**	**Freecol**	**Carol**	**Jabref**	**Total**
1	Deletion of Statement	15	7	23	6	10	9	70
2	Addition of Statement	3	1	21	7	5	1	38
3	Modification of Statement	5	6	1	7	5	10	34
4	Modification of Function Parameter	8	8	2	3	1	2	24
5	Modification of if-condition	4	6	1	1	3	2	17
6	Deletion of Function Call	3	0	0	4	2	6	15
7	Modification of Function Call	3	5	0	1	3	0	12

38 and 34 respectively. Again, MonoOSC contains the highest number of "Addition of Statement" types i.e. 21. The other frequent changes are "Modification of Function Parameter", "Modification of if-condition", "Deletion of Function Call" and "Modification of Function Call". We found that most of the change types in MonoOSC is "Addition of Statement" or "Deletion of Statement".

Finding the type of bugs or code construction for micro-clones is important because more frequent type of bug should be considered more carefully during software development. Since, from our manual analysis we found that most change type is "Deletion of Statement", so when a developer deletes any statement she should be more cautious for not creating any bugs. The other frequent change types during bug-fixes in micro-clones are "Addition of Statement", "Modification of Statement", and "Modification of Function Parameter".

VI. RELATED WORK

Micro-clone is a recent concern of code clone research area. The term *micro-clones* was first introduced by Beller et al. [23]. They have identified and statistically proved that majority of software bugs in micro-clones occur in the last line or statement of micro-clones. Tonder et al. [24] agreed with them and proposed detection and removal of micro-clones in large scale. Both Beller's and Tonder's paper used PVS-Studio [38] static analysis tool to detect faulty micro-clones. Also, Tonder et al. [24] used Boa [39] software mining infrastructure which contains parsed ASTs for all Java files in 380,125 Java repositories on GitHub and a domain-specific language (DSL). They [24] found that 95% of their pull requests from active GitHub repositories merged quickly and 76% of their accepted patches are removing (REM category) micro-clones. In contrast with these two papers [23], [24], Mondal et al. [25] investigated the importance of micro-clones during software evolution. They showed that micro-clones have a very high tendency of getting updated consistently.

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. [40] 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 [6] 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. [41] performed a user study to investigate how clone information can help programmers localize bugs in software systems. Jiang et al. [21] 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. [42] 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. [43] 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 bugs in micro-clones and regular code clones 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 (Type 1, Type 2, and Type 3), our target is to compare the bug-proneness of micro-clones and regular code clones. Mondal et al. [22] did not investigate the bug-proneness of micro-clone code in their study. Higo et al. [44] proposed a method to distinguish problematic code clones from non-problematic code clones. They have stated that not all clones are problematic for the systems. Thus, it is important to find and fix the problematic code clones. As a result of their study, they have found 22 problematic code clones.

Rahman et al. [45] 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 [46] clone detector. Authors choose DECKARD [46] as clone detection tool over CCFinder [47] and CP-Miner [40] since they found the performance of DECKARD is better than CCFinder and CP-Miner in their experiment. However, they considered monthly snap-shots (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 on regular and micro-clones whereas they only focused on the bug-proneness of regular clones.

Selim et al. [48] 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 six diverse subject systems in our investigation. Also, we investigate the bug-fix possibilities of regular and micro-clones. Selim et al. [48] did not perform a type (regular and micro) centric analysis in their study.

A new aspect has been discussed in a recent study [49] showing that bug-proneness is related with how recently the clone has been changed on a subject system. The more recent the changes happened the more possibility of occurring bugs. In another study, Mondal et al. [50] investigated bug propagation in code cloning and found that 33% of bug-fixing code clones contain propagated bugs. They have suggested to prioritize these clones for refactoring and tracking. It is noticeable in their research that near-miss code clones contain more propagated bugs than identical code clones. However, Mondal et al. [49], [50] did not investigate bug-proneness of micro-clones.

On the other hand, from a different perspective Rahman and Roy [51] show the relation between stability and bug-proneness of code clones. They have investigated five open source diverse subject systems written in Java. They have found statistically significant relation between stability and bug-proneness of code clones. Also, buggy clones have the tendency of changes more often than non-buggy clones. Moreover, Type 2 and Type 3 clones have stronger relation likelihood with their stability compared to Type 1 clones. They have also investigated in the fine-grained change types perspective and found low to medium significance influence by the changes on the relation between stability and bug-proneness of code clones. However, they did not investigate bug-proneness of micro-clones in their study. We investigate bug-proneness of micro-clones in our study.

Rakibul and Zibran [52] perform a comparative investigation in between buggy and non-buggy clone code. They have studied on three open source software systems written in Java containing 2,077 revisions in total. Using SourceMeter [53] they have observed 29 source code quality metrics to characterize the buggy clone code. Their approach to finding bugs in code clones is similar to other studies like [45], [49]. They have found that buggy clones have significantly higher complexity and lower maintainability than non-buggy clone code. Also, size of the method of buggy clone is higher than non-buggy clone method. While Rakibul and Zibran investigated regular code clones, we investigate the bug-proneness of micro-clones in our study.

We see that a number of studies have been conducted on the bug-proneness of regular code clones. However, bug-proneness of micro-clones have been ignored. Focusing on this we perform an in-depth investigation on bug's impacts in micro code clones and regular code clones in our research. Our experimental results are promising and provide useful implications for better understanding of the bug-proneness of micro-clone and regular clone code.

VII. Threats to Validity

We used the NiCad clone detector [26] for detecting both micro and regular clones. While all clone detection tools suffer from the *confounding configuration choice problem* [54] and might give different results for different settings of the tools, the setting that we used for NiCad for this experiment are considered standard [55] and with these settings NiCad can detect clones with high precision and recall [29]–[31]. Thus, we believe that our findings on the bug-proneness of micro code clones and regular 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 [32] and also used by Barbour et al. [56]. The technique proposed by Mocus and Votta [32] can sometimes select a non-bug-fix commit as a bug-fix commit mistakenly. However, Barbour et al. [56] 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-proneness of micro and regular 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 managing code clones.

VIII. Conclusion

In this paper, we investigate and compare the bug-proneness and their characteristics in between regular and micro-clones. From our investigation on six diverse subject systems, we found that micro-clones need to be handled more carefully than regular clones during software maintenance. We have found that changes to clone fragments due to fixing a bug occur more in micro-clones than regular clones. Moreover, total number of consistent changes due to bug-fix commits is higher in micro-clones than regular clones. Also, percentage of files those get affected due to bug-fix changes is higher in micro-clones than regular clones. Additionally, we found that percentage of severe bugs is higher in micro-clones than regular code clones. We believe that these findings are important for clone management specifically for managing micro-clones. Considering the findings of this study, in our future research we would like to investigate replicated bugs in micro-clones.

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] 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, p. 115.

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

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

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

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

[8] 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.

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

[10] 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.

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

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

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

[14] A. Lozano and M. Wermelinger, “Tracking clones’ imprint,” in *Proc. IWSC*, 2010, pp. 65–72.

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

[16] 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.

[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] M. Beller, A. Zaidman, and A. Karpov, “The Last Line Effect,” in *Proc. ICPC*, 2015, pp. 240–243.

[24] R. van Tonder and C. L. Goues, “Defending against the attack of the micro-clones,” in *Proc. ICPC*, 2016, pp. 1–4.

[25] M. Mondal, C. K. Roy, and K. A. Schneider, “Micro-clones in Evolving Software,” in *Proc. SANER*, 2018, pp. 50–60.

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

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

[28] 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.

[29] 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.

[30] 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.

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

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

[33] Mann-Whitney U Test. [Online]. Available: https://en.wikipedia.org/wiki/Mann%E2%80%93Whitney_U_test

[34] Mann-Whitney U Test. [Online]. Available: http://www.socscistatistics.com/tests/mannwhitney/Default2.aspx

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

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

[37] Difference Checker. [Online]. Available: https://www.diffchecker.com/

[38] PVS-Studio. [Online]. Available: https://www.viva64.com/en/pvs-studio/

[39] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories,” in *Proc. ICSE*, 2013, pp. 422–431.

[40] 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.

[41] D. Chatterji, J. C. Carver, B. Massengil, J. Oslin, and N. A. Kraft, “Measuring the Efficacy of Code Clone Information in a Bug Localization task: An Empirical Study,” in *Proc. ESEM*, 2011, pp. 20–29.

[42] K. Inoue, Y. Higo, N. Yoshida, E. Choi, S. Kusumoto, K. Kim, W. Park, and E. Lee, “Experience of Finding Inconsistently-changed Bugs in Code Clones of Mobile Software,” in *Proc. IWSC*, 2012, pp. 94–95.

[43] S. Xie, F. Khomh, and Y. Zou, “An Empirical Study of the Fault-proneness of Clone Mutation and Clone Migration,” in *Proc. MSR*, 2013, pp. 149–158.

[44] Y. Higo, K. Sawa, and S. Kusumoto, “Problematic code clones identification using multiple detection results,” in *Proceedings 16th Asia-Pacific Software Engineering Conference (APSEC)*, 2009, pp. 365–372.

[45] F. Rahman, C. Bird, and P. Devanbu, “Clones: What is that Smell?” in *Proc. MSR*, 2010, pp. 72–81.

[46] L. Jiang, G. Misherghi, and S. G. Z. Su, “Deckard: Scalable and accurate tree-based detection of code clones,” in *Proc. ICSE*, 2007, pp. 96–105.

[47] T. Kamiya, S. Kusumoto, and K. Inoue, “CCFinder: a multilinguistic token-based code clone detection system for large scale source code.” in *Proceedings IEEE Transactions on Software Engineering*, 2002, pp. 28(7):654–670.

[48] G. M. K. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, “Studying the Impact of Clones on Software Defects,” in *Proc. WCRE*, 2010, pp. 13–21.

[49] M. Mondal, C. K. Roy, and K. A. Schneider, “Identifying Code Clones having High Possibilities of Containing Bugs,” in *Proceedings of the 25th International Conference on Program Comprehension (ICPC)*, Buenos Aires-Argentina, May 2017, pp. 99–109.

[50] ——, “Bug Propagation through Code Cloning: An Empirical Study,” in *Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai, China, September 2017, pp. 227–237.

[51] M. S. Rahman and C. K. Roy, “On the Relationships between Stability and Bug-proneness of Code Clones: An Empirical Study,” in *Proceedings 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2017, pp. 131–140.

[52] M. R. Islam and M. F. Zibran, “On the Characteristics of Buggy Code Clones: A Code Quality Perspective,” in *Proceedings of the 12th International Workshop on Software Clones (IWSC)*, Campobasso, Italy, 2018, pp. 23–29.

[53] SourceMeter: Static source code analysis solution for Java, C/C++, C, Python and RPG. [Online]. Available: https://www.sourcemeter.com

[54] 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.

[55] 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.

[56] 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.