Comparing Bug Replication in Regular and Micro Code Clones

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*—Copying and pasting source code during software development is known as code cloning. Clone fragments with a minimum size of 5 LOC were usually considered in previous studies. In recent studies, clone fragments which are less than 5 LOC are referred as micro-clones. It has been established by the literature that code clones are closely related with software bugs as well as bug replication. None of the previous studies have been conducted on bug-replication of micro-clones. In this paper we investigate and compare bug-replication in between regular and micro-clones. For the purpose of our investigation, we analyze the evolutionary history of our subject systems and identify occurrences of similarity preserving co-changes (SPCOs) in both regular and micro-clones where they experienced bug-fixes. From our experiment on thousands of revisions of six diverse subject systems written in three different programming languages, C, C# and Java we find that the percentage of clone fragments that take part in bug-replication is often higher in micro-clones than in regular code clones. The percentage of bugs that get replicated in micro-clones is almost the same as the percentage in regular clones. Finally, both regular and micro-clones have similar tendencies of replicating severe bugs according to our experiment. Thus, micro-clones in a code-base should not be ignored. We should rather consider these equally important as of the regular clones when making clone management decisions.**

***Index Terms*—Code Clones, Micro-clones, Replicated Bugs, Severe 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 the 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. Recently, Islam et al. [26] performed a comparative study on the bug-proneness between regular and micro-clones. They found that micro-clones are more bug-prone than regular clones. Moreover, the amount of severe bugs in micro-clones is comparatively higher than in regular clones. However, they did not investigate bug replication of micro-clones. This motivates us to investigate the bug replication in both micro and regular code clones. In order to explore the effects of bug replication 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 replication 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 [27]. We analyze the evolution history of both micro and regular code clones, and investigate whether they contain replicated bugs and to what extent.

In our experiment, the minimum and maximum size of a micro-clone fragment can be 1 LOC and 4 LOC respectively. We consider those micro-clones that are not parts of regular

TABLE I
RESEARCH QUESTIONS

SL	Research Question
RQ 1	What percentage of the clone fragments in Regular and Micro-clones takes part in bug-replication?
RQ 2	What is the extent of bug-replication in the buggy clone classes of Regular code clones and Micro-clones?
RQ 3	What percentage of bugs experienced by regular and micro-clones are replicated bugs?
RQ 4	Are the replicated bugs in micro-clones more likely to be severe than the replicated bugs in regular 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				

clones. We detect regular code clones of at least 5 LOC because Wang et al. [28] reported this size as the best minimum threshold for detecting regular clones.

To investigate bug replication in regular and micro-clones, we observe bug-fix commits reported by the developers from thousands of commits in our candidate open source projects. We observe the intensity of bug-replication in both micro-clones and regular code clones and answer four important research questions listed in Table I. The major findings from our research are as follows.

- The percentage of clone fragments that experienced bug-replication in micro-clones is often higher than that of regular code clones.
- The percentage of bugs that get replicated in micro-clones is almost the same as of the percentage of bugs that get replicated in regular code clones.
- The replicated bugs in both regular and micro-clones have similar tendencies of being severe.

From these findings we see that micro-clones are equally important as regular code clones. Thus, we should also consider micro-clones when making clone management decisions.

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 discusses the related work and compares our study with the existing ones, Section VI discusses possible threats to validity, and Section VII 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 identifiers and/or changing 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 minimum 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. Similarity Preserving Co-change (SPCO) of two or more clone fragments

Let us consider two code fragments, CF1 and CF2, which are clones of each other in revision R of a subject system. A commit operation was applied on revision R and both of these two fragments were changed (i.e., the clone fragments co-changed) in such a way that they were again considered as clones of each other in the next revision R+1 (i.e., created because of the commit). In other words, the clone fragments preserved their similarity even after experiencing changes in the commit operation. Thus, we call this co-change of clone fragments (i.e., change of more than one clone fragment together) a Similarity Preserving Co-change (SPCO).

D. 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 tendencies of replicating bugs.

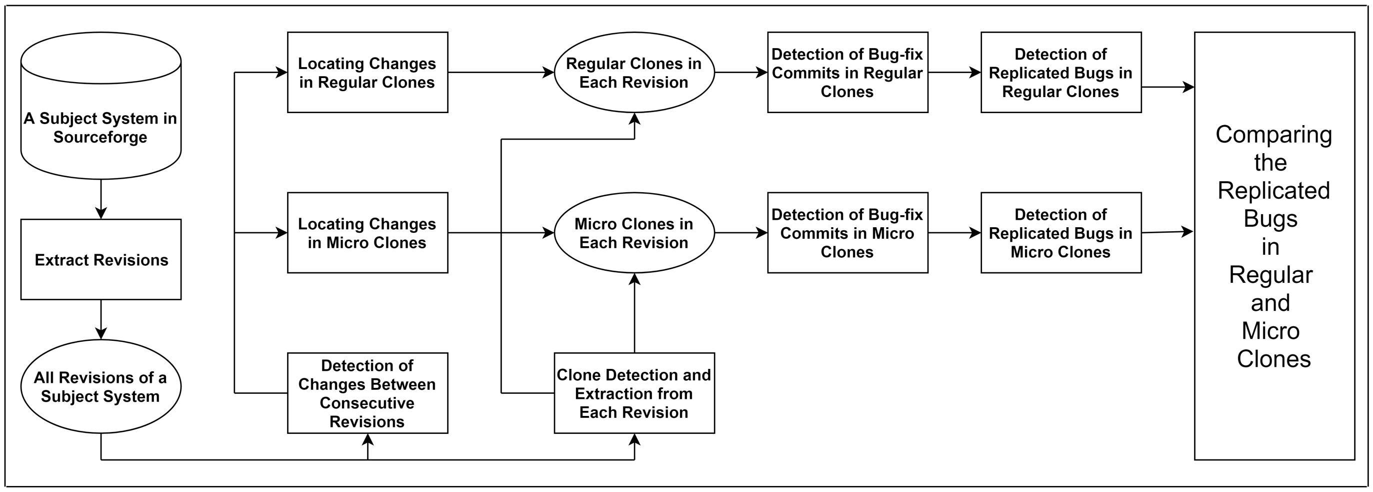
A Subject System in Sourceforge
Extract Revisions
All Revisions of a Subject System
Locating Changes in Regular Clones
Locating Changes in Micro Clones
Detection of Changes Between Consecutive Revisions
Regular Clones in Each Revision
Micro Clones in Each Revision
Clone Detection and Extraction from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

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

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 [29]. 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 [27] 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 [30]. 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 [27] for detecting clones since it can detect code clones with high precision and recall [31], [32]. Using NiCad, we detect block clones (both exact and near-miss) of at least 5 lines with 30% 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 30% dissimilarity threshold and blind renaming of identifiers as was detected by Mondal et al. [25]. NiCad settings for detecting both micro and regular code clones are shown in Table III.

TABLE III
NiCad Settings for Detecting Regular and Micro-clones

Clones	Clone Granularity	Min Line	Max Line	Identifier Renaming	Dissimilarity Threshold
Regular Clones	Block	5	500	Blind Rename	30%
Micro Clones	Block	1	4	Blind Rename	30%

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 [33] 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 [34] 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 replicated bugs in regular and micro-clones.

C. Identifying replicated bugs in code clones

In Section II we defined the Similarity Preserving Co-change (SPCO) of two or more clone fragments from same clone class. To identify replicated bugs in code clones we analyze clone evolution history of a software system. We perform following two steps for the identification of bug-replication.

Step 1: First we detect all the bug-fix commit operations of a subject system. The detail procedure of this step is described in Section III-B.

Step 2: For each of the bug-fix commits found in step 1, we check if any of the clone fragments experienced a Similarity Preserving Co-change (SPCO) in this commit. Suppose a bug-fix commit BFC was applied on revision R of a candidate system. If two or more clone fragments from a clone class in revision R experienced an SPCO in the commit BFC, then we consider it as the case of fixing a replicated bug. In an SPCO the participating clone fragments are likely to be changed consistently (i.e. in the same way) [35]. To detect SPCOs of clone fragments in regular code clones and micro-clones we use SPCP-Miner [30] tool.

We identify all cases of fixing replicated bugs for both regular code clones and micro-clones by analyzing the clone evolution history of each of our subject systems listed in Table II. Figure 1 summarizes all the steps of our experiment.

IV. Experiment Results and Analysis

Our research questions are listed in Table I. In this section, we analyze our experiment results for six subject systems and answer the research questions from our analysis.

A. Answering the first research question (RQ 1)

RQ 1: *What percentage of the clone fragments in Regular and Micro-clones takes part in bug-replication?*

Motivation. Measuring the percentage of clone fragments which take part in bug replication is an important way to contrast between micro and regular code clones. If a particular category of code clones (regular clones or micro-clones) contains more replicated bugs, we should be more careful about that category while making clone management decisions. The software bug that replicates through clone code have a ripple effect in the whole system. Thus answering this research question has an important impact on clone research since code clones that gravitate to replicate bugs should be emphasized during software maintenance.

Methodology. To answer this research question we identify bug-replication in code clones by mining the clone evolution history of each of our subject systems. Detail procedure of detecting replicated bugs in regular and micro-clones is elaborated in section III-C. We automatically detect the bug-fix commits and then identify the similarity preserving co-changes (SPCOs) of clones in these bug-fix commits. If an SPCO of two or more clone fragments occurs in a bug-fix commit, then it is an indication of fixing of a replicated bug.

We use the following two measures to calculate the percentage of clone fragments containing replicated bugs.

CFRB: The number of clone fragments that experienced similarity preserving co-change (SPCO) i.e. clone fragments which contain replicated bugs.

BFCR: The number of distinct bug-fix commits where regular and/or micro-clones experienced similarity preserving co-changes (SPCOs).

To determine the percentage of clone fragments that takes part in bug-replication we use Equation 1.

$$PCFRB = \frac{100 \times CFRB}{BFCR} \tag{1}$$

Here, PCFRB is the percentage of clone fragments containing replicated bugs per revision. We compute the overall percentage of the clone fragments that experienced bug-replication for both regular and micro-clones using the following equation.

$$OPCFRB = \frac{100 \times \sum_{all\ systems} CFRB}{\sum_{all\ systems} BFCR} \tag{2}$$

Table IV shows the number of distinct clone code fragments which took part in bug-replication in both regular and micro-clone code for six subject systems. From this table we observe that the number of distinct clone fragments is higher in micro-clones for three subject systems i.e. for Brlcad, Freecol and Jabref. For other three subject systems i.e. for Ctags, MonoOSC and Carol number of replicated clone fragments is higher in regular code clones than that of micro-clones. Figure 2 shows the percentage of distinct clone fragments which are related to bug-replication in regular and micro-clones. In overall, percentage of clone fragments that experienced bug-replication is slightly higher in micro-clones than the regular code clones.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 1. We perform Mann-Whitney-Wilcoxon (MWW) test [36], [37] to check the statistical significance of the result. We are interested

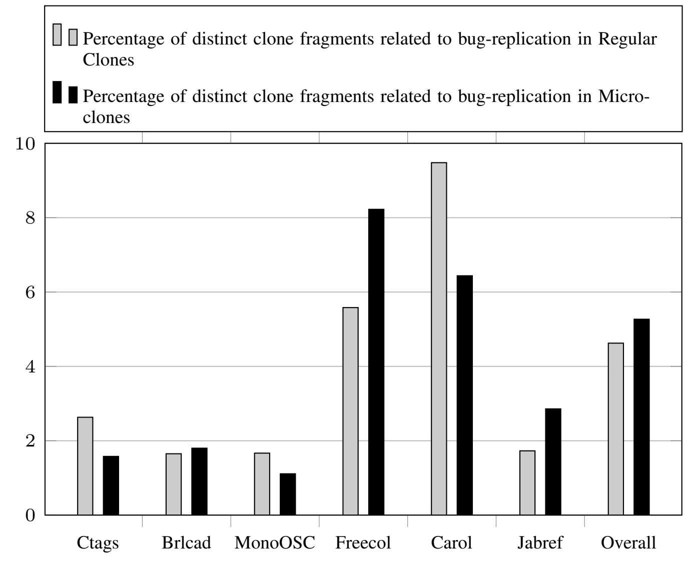

Fig. 2. Percentage of clone fragments that related to bug-replication in regular and micro-clones.

TABLE IV
NUMBER OF CLONE FRAGMENTS THAT EXPERIENCED BUG-REPLICATION IN REGULAR AND MICRO-CLONES

Subject Systems	**Regular Clones (CFRB)**	**Micro Clones (CFRB)**	**Revisions (BFCR)**
Ctags	50	30	19
Brlcad	33	36	20
MonoOSC	15	10	9
Freecol	547	806	98
Carol	455	309	48
Jabref	121	200	70

to know that if the percentage of the clone fragments that takes part in bug-replication in micro-clones is significantly higher than in regular clones. MWW test is non-parametric and does not require normal distribution of data. The significance level of our MWW test is 5% i.e. if the p-value is less than 0.05 and the U-value is less than or equal to critical U-value then the difference will be significant. We found that the p-value is 0.93624 which is much greater than the significance level 0.05. Also, the U-value is 17 and the critical value of U at p<0.05 is 5. So the U-value is greater than the critical U-value. This MWW test result prevails that the difference between the two percentages of clone fragments containing replicated bugs in regular and micro-clones is insignificant.

Answer to RQ 1. The percentage of the clone fragments in micro-clones that takes part in bug-replication (5.27%) is slightly higher than the percentage of the clone fragments in regular clones that takes part in bug-replication (4.63%).

From our above experimental results we can state that the difference between two overall percentages (regular and micro-clones) is not prominent. This implies that micro-clones are as important as regular code clones. Thus, we can not disregard micro-clones during software maintenance. Developers should equally emphasize on both micro-clones and regular code clones while performing clone management.

B. Answering the second research question (RQ 2)

RQ 2: *What is the extent of bug-replication in the buggy clone classes of Regular code clones and Micro-clones?*

Motivation. After knowing the percentage of clone fragments containing replicated bugs in both regular and micro-clones, we still need to understand the extent of bug-replication in the clone classes which contains bugs for regular and micro-clones. Instinctively, the code clone that contains higher extent of bug-replication expected to be more harmful for the system. To answer this research question we perform the following procedures.

Methodology. Following the procedure of Section III-B, we first detect the bug-fix commits. For each of the bug-fix commits we detect whether a clone class was affected in that commit. This means whether one or more clone fragments from the clone class were changed for fixing a bug in that commit. Suppose a bug-fix commit was applied on the revision R of a subject system. Let us consider a clone class CC which was affected by BFC. If two or more clone fragments from CC are experienced a similarity preserving co-change (SPCO) in BFC then we select this clone class as an *Eligible Clone Class* (ECC). In other words, an ECC contains replicated bug. We discard all the clone classes which did not experienced similarity preserving co-change (SPCO) despite of being affected by BFC.

For each eligible clone class (ECC) we calculate following two measures:

CF: The total number of clone fragments in the eligible clone class i.e. in the ECC.

CFRB: The number of clone fragments that experienced similarity preserving co-change (SPCO) i.e. clone fragments which contain replicated bugs (same as illustrated in Section IV-A).

From these two measures of an eligible clone class (ECC), we calculate the extent of bug-replication in that ECC using the following equation.

$$EBR = \frac{100 \times CFRB}{CF} \tag{3}$$

Here, EBR stands for the extent of bug-replication for a particular ECC. We get the percentage of the extent of bug-replication (PEBR) by considering all the ECCs of regular or micro-clones from all the replicated bugs with respect to all bugs of a subject system. We use Equation 4 for finding the value of PEBR.

$$PEBR = \frac{100 \times \sum_{all} CFRB}{\sum_{all} CF} \quad (4)$$

The overall percentage of the extent of bug-replication for all subject systems is calculated using following equation.

$$OPEBR = \frac{100 \times \sum_{all\ systems} CFRB}{\sum_{all\ systems} CF} \quad (5)$$

In Equation 5, OPEBR refers to the overall percentage of the extent of bug-replication with respect to all bugs. The OPEBR is calculated for both regular and micro-clones.

Table V shows the extent of bug-replication experienced by both regular and micro-clones. Figure 3 depicts the extent of replicated bugs in the buggy clone classes for regular and micro-clones. From Table V and Figure 3 we observe that the extent of bug-replication is a bit higher in regular code clones than that of micro-clones for each of the subject systems. The highest extent (97.06%) of replicated bugs found in Brlcad for regular code clones which is nearly 100%. Second highest extent (88.24%) of the bug-replication found in MonoOSC for regular clone code. Other than these two cases, for rest of the cases, extent of bug-replication for regular clones varies from 5% to 17% and for micro-clones it varies from 0.25% to 2%.

In this research question we did not perform the MWW test since we believe that the actual number of extension of replicated bugs is more important than the percentage of extension. This is because we found that the number of clone fragments is very high in micro-clones compared with regular clones. For instance, from Table V we see that in Freecol, total number of clone fragments in eligible clone classes is 207765 in micro-clones whereas in regular clones it is only 10507 which is almost 20 times. However, the number of clone fragments experienced bug-replication is only 806 and 547 in micro clones and regular clones respectively. For the purpose of clone refactoring or clone tracking, we have to deal with the actual number of clone fragments (all the bugs need to be fixed) rather than percentages. Thus, though the extent of bug-replication in the buggy clone classes is higher in regular clones than micro-clones, we should not neglect micro-clones because of its characteristic of having higher number of clone fragments compared to regular clones.

Answer to RQ 2. The extent of bug replication in the buggy clone classes of regular code clones is higher than the extent in micro-clones. The overall extent of replicated bugs in regular clones is 8.21% and in micro-clones the overall extent of bug-replication is 0.43%.

Clone classes containing replicated bugs are considered more harmful. While refactoring clone classes of code clones, developers should focus on clone classes that have replicated bugs. In this research question, our findings show that both regular code clones and micro-clones have replicated bugs in their clone classes. Hence, both regular and micro-clones should be equally considered carefully while clone management (such as refactoring).

TABLE V
EXTENT OF REPLICATED BUGS THAT EXPERIENCED BY CODE CLONES IN REGULAR AND MICRO-CLONES

Subject	**Regular Clones**		**Micro Clones**	
Systems	**CFRB**	**CF**	**CFRB**	**CF**
Ctags	50	795	30	1745
Brlcad	33	34	36	6189
MonoOSC	15	17	10	517
Freecol	547	10507	806	207765
Carol	455	2784	309	29907
Jabref	121	742	200	79019
Total	1221	14879	1391	325142

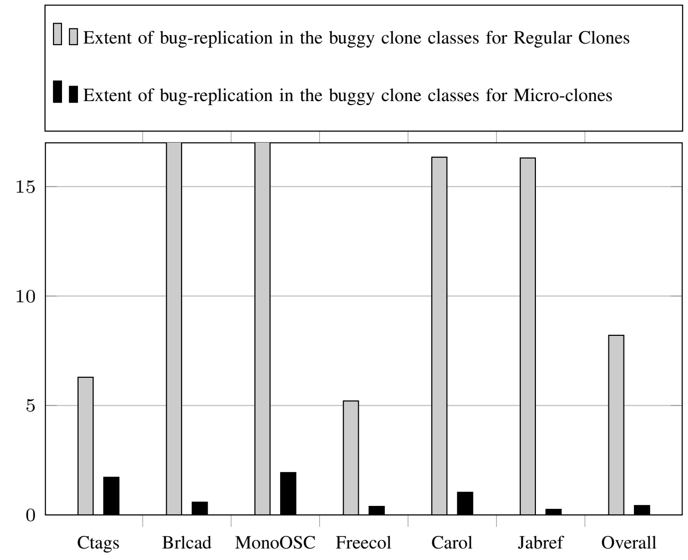

Fig. 3. Extent of bug-replication in the buggy clone classes for regular and micro-clones.

C. Answering the third research question (RQ 3)

RQ 3: *What percentage of bugs experienced by regular and micro-clones are replicated bugs?*

Motivation. We are interested to identify whether most of the bugs in clone code are replicated bugs or not. Finding the answer of this research question will help us to understand that if bug-replication is a recurrent event in software development. Intuitively, if almost all bugs are found to be replicated bugs then developers should be more concerned about copy/pasting source code without containing any bugs.

Methodology. To answer this research question, we calculate the total number of bugs and also the total number of bugs that got replicated respectively for each of the subject systems. We calculate these numbers for both regular and micro-clones. We use following two counters to demonstrate our methods.

NBC (The number of bugs experience by the code clones): Suppose a bug-fix commit is denoted by BFC and it was applied on the revision R of a candidate system. We increase the counter NBC by one if one or more clone classes residing in this revision R were affected by the commit BFC.

NBRC (The number of bugs that were replicated in the code clones): Suppose a bug-fix commit was applied on the revision R of a subject system and one or more clone classes in this revision were affected by this commit. We increase the counter NBRC by one if any of these affected clone classes experienced a similarity preserving co-change (SPCO) in this commit. According to our previous discussion in Section III-C experiencing a similarity preserving co-change (SPCO) in a bug-fix commit operation is an evidence of the existence of replicated bugs in the code clones.

The percentage of replicated bugs with respect to all bugs found in code clones for both regular and micro-clones of a subject system is calculated by Equation 6.

$$PNBRC = \frac{100 \times NBRC}{NBC} \tag{6}$$

We calculate overall percentage of replicated bugs with respect to all bugs for six subject systems using following equation.

$$OPNBRC = \frac{100 \times \sum_{all\ systems} NBRC}{\sum_{all\ systems} NBC} \tag{7}$$

Table VI shows the number of replicated bugs with respect to all bugs for both regular and micro-clones for each of the subject systems. Here, we observe that for three subject system i.e. Ctags, Brlcad and Carol in regular code clones all of the bugs are replicated bugs. In case of micro-clones this scenario is true for MonoOSC. For the rest of the subject systems, in both regular and micro-clones majority number of bugs are found to be replicated bugs. Figure 2 depicts the percentage of replicated bugs in regular and micro-clones. In this figure we see that for all of the subject systems, percentage of replicated bugs is slightly higher in regular clones than in micro-clones except for the MonoOSC. The overall percentage of replicated bugs in regular code clones is 96.53% and in micro-clones it is 91.87%. To better understand the difference of these two percentages we perform following MWW test.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 3. We want to know whether the difference between two percentages of replicated bugs with respect to all bugs in regular and micro-clones is significant or not. To investigate statistical significance we perform Mann-Whitney-Wilcoxon (MWW) test [36], [37] on the results. We found that for two-tailed, 5% significance level the p-value is 0.20054 and critical U-value is 5. Here, p-value is greater than the significance level 0.05. Also, we found that the U-value is 9.5 which is greater than the critical U-value 5. Hence, the MWW test result reveals that the difference of percentages is not significant.

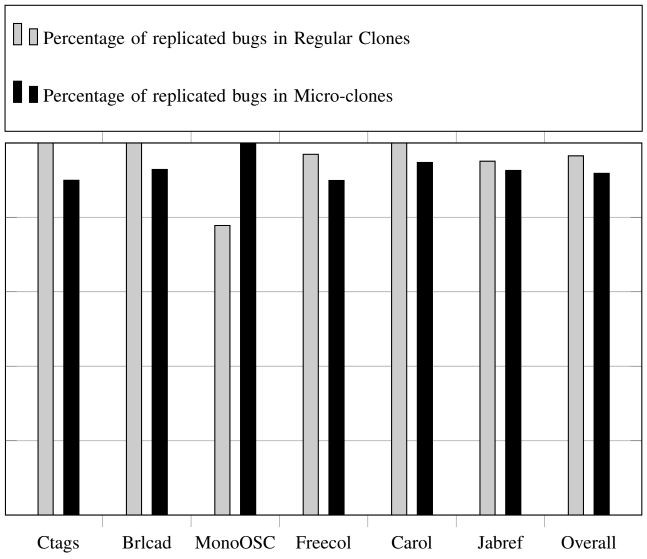

Fig. 4. Percentage of replicated bugs in regular and micro-clones.

TABLE VI
NUMBER OF REPLICATED BUGS THAT EXPERIENCED BY CODE CLONES IN REGULAR AND MICRO-CLONES

Subject	Regular Clones		Micro Clones	
Systems	NBRC	NBC	NBRC	NBC
Ctags	15	15	9	10
Brlcad	11	11	13	14
MonoOSC	7	9	4	4
Freecol	64	66	80	89
Carol	31	31	36	38
Jabref	39	41	50	64

Answer to RQ 3. A substantial number of bugs in both regular code clones and micro-clones are found to be replicated bugs. The overall percentage of replicated bugs is higher by 4.66% in regular clones than micro-clones.

Since the difference of the two percentages (regular and micro-clones) is insignificant, we can say that both micro-clones and regular clones contain a high percentage (over 90%) of replicated bugs in code bases. Hence, both micro-clones and regular code clones carry equivalent importance in software maintenance.

D. Answering the fourth research question (RQ 4)

RQ 4: *Are the replicated bugs in micro-clones more likely to be severe than the replicated bugs in regular clones?*

Motivation. Finding the severity of replicated bugs is an important criterion to compare between regular and micro code clones. Answering this research question will let us know which code clone requires more attention than the other during fixing a bug. Since severe bugs need to be fixed immediately, if severe replicated bugs occur more in a certain type (regular or micro-clones) then developers can be more careful about that type of clone in advance during software maintenance.

Methodology. To find the severe replicated bugs we automatically perform a heuristic search in bug-fix commit messages of regular and micro-clones for the six subject systems. According to Lamkanfi's [38] proposal, a list of keywords can identify severe and non-severe bugs from textual information. We use these keywords to identify severe bugs in replicated bug-fixing commit messages of our candidate systems.

NSBRC (The number of severe bugs that were replicated in the code clones): We denote the number of severe replicated bugs of a subject system by NSBRC. We increase the value of NSBRC by one if a replicated bug found in code clones is severe.

We calculate the percentage of severe replicated bugs with respect to all replicated bugs in regular code clones and micro-clones using Equation 8. Here, NBRC is the number of bugs that were replicated in code clones as defined in previous Section IV-C.

$$PNSBRC = \frac{100 \times NSBRC}{NBRC} \tag{8}$$

We also calculate overall percentage of severe replicated bugs with respect to all replicated bugs in regular and micro-clones using the following way.

$$OPNSBRC = \frac{100 \times \sum_{all\ systems} NSBRC}{\sum_{all\ systems} NBRC} \tag{9}$$

Here, OPNSBRC refers to overall percentage of severe bugs which are related to bug-replication with respect to all bugs related to bug-replication in regular and micro-clones.

Table VII shows the number of severe bugs that experienced bug-replication in code clones for both regular and micro-clones in six subject systems. Figure 5 shows the percentage of severe replicated bugs with respect to all replicated bugs for regular and micro-clones. For subject system MonoOSC and Carol percentage of severe replicated bugs is higher (i.e. 25% and 16.67% respectively) in micro-clones than that (i.e. 14.29% and 3.23% respectively) of regular code clones. For the rest of the subject systems percentage of severe replicated bugs is higher in regular code clones than the micro-clones. In overall, percentage of severe replicated bugs is slightly higher in regular code clones than micro-clones.

Mann-Whitney-Wilcoxon (MWW) Test for RQ 4. To understand that if the difference between the percentages of severe replicated bugs in regular and micro-clones is significant or not, we conduct MWW test [36], [37] on RQ4's result. For significance level of 5% and two-tailed MWW test we find the critical value of U is 5. Our MWW test result shows that the p-value is 1 which is far greater than the significance level 0.05 and U-value is 17.5 which is greater than critical U-value

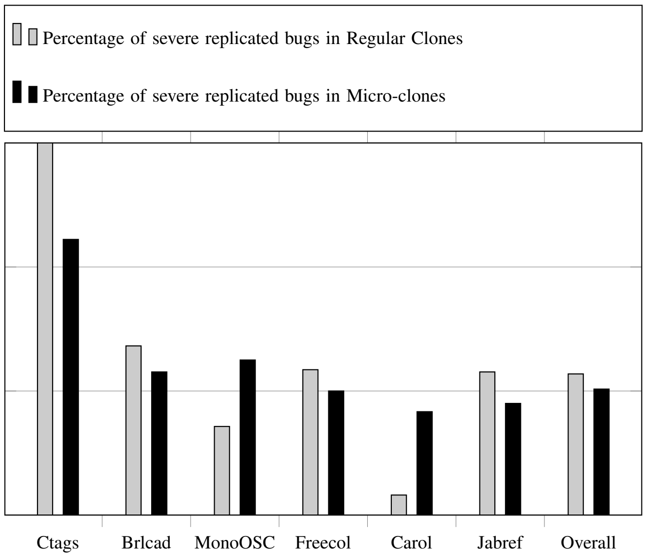

Fig. 5. Percentage of severe replicated bugs in regular and micro-clones.

TABLE VII
NUMBER OF SEVERE REPLICATED BUGS THAT EXPERIENCED BY CODE CLONES IN REGULAR AND MICRO-CLONES

Subject Systems	Regular Clones		Micro Clones	
	NBRC	NSBRC	NBRC	NSBRC
Ctags	15	9	9	4
Brlcad	11	3	13	3
MonoOSC	7	1	4	1
Freecol	64	15	80	16
Carol	31	1	36	6
Jabref	39	9	50	9

5. This proves that the percentage of severe replicated bugs in regular and micro-clones is nearly equal. Table VIII shows all the result of MWW tests for three research questions. Here, we can see that for all RQs p-values are greater than significance level 0.05 and U-values are greater than critical U-value.

Answer to RQ 4. Replicated bugs in regular code clones are more severe than the replicated bugs in micro-clones. The overall percentage of severe replicated bugs for regular clones is 22.75% and for micro-clones it is 20.31%.

Since the difference between the percentage of severe replicated bugs in regular and micro-clones is minor i.e. 2.44%, we can say that both regular and micro-clones should be taken care of during clone management from the bug severity perspective.

For all research questions, we perform manual analysis to check our implementation is correct. For all subject systems, we investigated at least first 50 clone fragments and/or revisions manually to confirm our analysis.

TABLE VIII
MANN-WHITNEY-WILCOXON TEST RESULT FOR RQ1, RQ3 AND RQ4

Research Question No.	p-value	U-value	Critical Value of U
RQ1	0.93624	17	5
RQ3	0.20054	9.5	5
RQ4	1	17.5	5
Considering level of significance is 5%. For all RQs, U-value > Critical value of U			

V. 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. On the other hand, in our study we consider all lines of a micro-clone fragment. Tonder et al. [24] agreed with Beller and proposed detection and removal of micro-clones in large scale. Both Beller's and Tonder's paper used PVS-Studio [39] static analysis tool to detect faulty micro-clones. 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. None of these three studies on micro-clones [23]–[25] showed any comparison between micro-clones and non-micro clones or regular clones. Recently, Islam et al. [26] performed a comparative study on bug-proneness in between regular and micro-clones and showed that micro-clones are more bug-prone than regular clones. However, they did not compare the bug-replication in micro-clones and regular clones. In a previous study Islam et al. [40] investigated bug-replication on three types of regular clones (i.e. Type 1, Type 2 and Type 3) and found that bug-replication is a common phenomena in code cloning. They did not consider micro-clones in their study. Whereas in our paper, we compare bug-replication between regular clones and micro clones.

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. [41] 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. [42] 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. [43] 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. [44] investigated fault-proneness of Type 3 clones in three open-source software systems. They 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-replication of micro-clones and regular code clones. Mondal et al. [22] did not investigate the bug-replication of micro-clone code in their study. Higo et al. [45] 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. [46] 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 [47] clone detector. However, they considered monthly snap-shots (i.e., revisions) of their systems and thus, they have the possibility of missing buggy commits. In our study, we consider all the snap-shots/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-replication 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-replication possibilities of regular and micro-clones. Selim et al. [48] did not perform such an investigation.

A number of studies have also been done on the late propagation in clones and its relationships with bugs. Aversano et al. [4] investigated clone evolution in two subject systems and reported that late propagation in clones is directly related to bugs. Barbour et al. [5] investigated eight different patterns of late propagation considering Type 1 and Type 2 clones of three subject systems and identified those patterns that are likely to introduce bugs and inconsistencies to the code-base. However, we emphasized on bug-replication rather than late propagation in our 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. However, Mondal et al. [49], [50] did not investigate bug-replication of micro-clones. Another study [51] compared software bugs in clone and non-clone code and found that clone code is more bug-prone than non-clone code. In contrast with this study, we compare software bugs in micro and regular code clones.

On the other hand, from a different perspective Rahman and Roy [52] 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. However, they did not investigate bug-replication of micro-clones in their study. We investigate bug-replication of micro-clones in our study. Rakibul and Zibran [53] 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 [54] they have observed 29 source code quality metrics to characterize the buggy clone code. They have found that buggy clones have significantly higher complexity and lower maintainability than non-buggy clone code. While Rakibul and Zibran investigated regular code clones, we investigate the bug-replication 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 replicated 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-replication of micro-clone and regular clone code.

VI. Threats to Validity

We used the NiCad clone detector [27] for detecting both micro and regular clones. While all clone detection tools suffer from the *confounding configuration choice problem* [28] 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 [31]–[33]. Thus, we believe that our findings on the bug-replication 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 [34] and also used by Barbour et al. [56]. The technique proposed by Mocus and Votta [34] 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-replication 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.

VII. Conclusion

In this paper, we investigate and compare the aspects of bug-replication in between regular code clones and micro-clones. After investigating on six diverse subject systems, we found that micro-clones are as equally important as regular code clones. We have found that the percentage of clone fragments which are related with bug-replication is some times higher in micro-clones than that of regular code clones. Moreover, the percentage of replicated bugs in micro-clones is almost the same as the percentage in regular clones. Additionally, both regular code clones and micro code clones have the similar tendencies of replicating severe bugs. We believe that these findings are important from the clone management perspective. Both micro-clones and regular clones should be considered for software evaluation and maintenance. Since only a limited number of studies have been performed on micro-clones, more research is needed to elaborate micro-clone more profoundly. In future we would like to explore micro-clones for more number of systems of various programming languages so that we can perform a language centric empirical study on bug-proneness of micro-clones.

Acknowledgment

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), and by two Canada First Research Excellence Fund (CFREF) grants coordinated by the Global Institute for Food Security (GIFS) and the Global Institute for Water Security (GIWS).

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. IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 18–33.

[2] C. K. Roy, "Detection and analysis of near-miss software clones," in *Proc. IEEE International Conference on Software Maintenance (ICSM)*, 2009, pp. 447–450.

[3] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," *Technical Report 2007-541*, pp. 1 – 115, 2007.

[4] L. Aversano, L. Cerulo, and M. D. Penta, "How clones are maintained: An empirical study," in *Proc. 11th European Conference on Software Maintenance and Reengineering (CSMR)*, 2007, pp. 81–90.

[5] L. Barbour, F. Khomh, and Y. Zou, "Late Propagation in Software Clones," in *Proc. IEEE International Conference on Software Maintenance (ICSM)*, 2011, pp. 273–282.

[6] N. Göde and R. Koschke, "Frequency and risks of changes to clones," in *Proc. 33rd International Conference on Software Engineering (ICSE)*, 2011, pp. 311–320.

[7] N. Göde and J. Harder, "Clone Stability," in *Proc. 15th European Conference on Software Maintenance and Reengineering (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. ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, 2010, pp. 73–82.

[9] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do Code Clones Matter?" in *Proc. 31st International Conference on Software Engineering (ICSE)*, 2009, pp. 485–495.

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

[11] J. Krinke, "A study of consistent and inconsistent changes to code clones," in *Proc. 14th Working Conference on Reverse Engineering (WCRE)*, 2007, pp. 170–178.

[12] J. Krinke, "Is cloned code more stable than non-cloned code?" in *Proc. Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2008, pp. 57–66.

[13] J. Krinke, "Is Cloned Code older than Non-Cloned Code?" in *Proc. 5th International Workshop on Software Clones (IWSC)*, 2011, pp. 28–33.

[14] A. Lozano and M. Wermelinger, "Tracking clones' imprint," in *Proc. 4th International Workshop on Software Clones (IWSC)*, 2010, pp. 65–72.

[15] A. Lozano and M. Wermelinger, "Assessing the effect of clones on changeability," in *Proc. IEEE International Conference on Software Maintenance (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. 27th Annual ACM Symposium on Applied Computing (SAC)*, 2012, pp. 1227–1234.

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

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

[19] J. Li and M. D. Ernst, "CBCD: Cloned Buggy Code Detector," in *Proc. 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 310–320.

[20] D. Steidl and N. Göde, "Feature-Based Detection of Bugs in Clones," in *Proc. 7th International Workshop on Software Clones (IWSC)*, 2013, pp. 76–82.

[21] L. Jiang, Z. Su, and E. Chiu, "Context-Based Detection of Clone-Related Bugs," in *Proc. 6th European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (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. IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 91–100.

[23] M. Beller, A. Zaidman, and A. Karpov, "The Last Line Effect," in *Proc. 23rd International Conference on Program Comprehension (ICPC)*, Florence, Italy, 2015, pp. 240–243.

[24] R. van Tonder and C. L. Goues, "Defending against the attack of the micro-clones," in *Proc. 24th International Conference on Program Comprehension (ICPC)*, Austin, TX, USA, 2016, pp. 1–4.

[25] M. Mondal, C. K. Roy, and K. A. Schneider, "Micro-clones in Evolving Software," in *Proc. 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 50–60.

[26] J. F. Islam, M. Mondal, and C. K. Roy, "A Comparative Study of Software Bugs in Micro-clones and Regular Code Clones," in *Proc. 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Hangzhou, China, 2019, pp. 73–83.

[27] J. R. Cordy and C. K. Roy, "The NiCad Clone Detector," in *Proc. 19th International Conference on Program Comprehension (ICPC) Tool Demo*, 2011, pp. 219–220.

[28] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for Better Configurations: A Rigorous Approach to Clone Evaluation," in *Proc. 12th European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/SIGSOFT FSE)*, 2013, pp. 455–465.

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

[30] 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. 22nd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2015, pp. 484–488.

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

[32] C. K. Roy and J. R. Cordy, "A Mutation / Injection-based Automatic Framework for Evaluating Code Clone Detection Tools," in *Proc. International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, 2009, pp. 157–166.

[33] J. Svajlenko and C. K. Roy, "Evaluating Modern Clone Detection Tools," in *Proc. IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 321–330.

[34] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes using Historic Databases," in *Proc. IEEE International Conference on Software Maintenance (ICSM)*, 2000, pp. 120–130.

[35] M. Mondal, C. K. Roy, and K. A. Schneider, "Automatic Ranking of Clones for Refactoring through Mining Association Rules," in *Proc. IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, 2014, pp. 114–123.

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

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

[38] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the Severity of a Reported Bug," in *Proc. 7th IEEE Working Conference on Mining Software Repositories (MSR)*, 2010, pp. 1–10.

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

[40] J. F. Islam, M. Mondal, and C. K. Roy, "Bug Replication in Code Clones: An Empirical Study," in *Proc. 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Japan, 2016, pp. 68–78.

[41] 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. 6th conference on Symposium on Opearting Systems Design Implementation (OSDI)*, 2004, pp. 20–20.

[42] 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. International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2011, pp. 20–29.

[43] 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. 6th International Workshop on Software Clones (IWSC)*, 2012, pp. 94–95.

[44] S. Xie, F. Khomh, and Y. Zou, "An Empirical Study of the Fault-proneness of Clone Mutation and Clone Migration," in *Proc. 10th IEEE Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 149–158.

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

[46] F. Rahman, C. Bird, and P. Devanbu, “Clones: What is that Smell?” in *Proc. 7th IEEE Working Conference on Mining Software Repositories (MSR)*, 2010, pp. 72–81.

[47] L. Jiang, G. Misherghi, and S. G. Z. Su, “Deckard: Scalable and accurate tree-based detection of code clones,” in *Proc. 29th International Conference on Software Engineering (ICSE)*, 2007, pp. 96–105.

[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. 17th Working Conference on Reverse Engineering (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 *Proc. 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 *Proc. IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai, China, September 2017, pp. 227–237.

[51] J. F. Islam, M. Mondal, C. K. Roy, and K. A. Schneider, “A Comparative Study of Software Bugs in Clone and Non-Clone code,” in *Proc. 29th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, USA, 2017, pp. 436–443.

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

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

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

[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. 16th International Conference on Program Comprehension (ICPC)*, 2008, pp. 172–181.

[56] L. Barbour, F. Khomh, and Y. Zou, “An empirical study of faults in late propagation clone genealogies,” *Journal of Software: Evolution and Process*, vol. 25(11):1139-1165, 2013.