On the Characteristics of Buggy Code Clones: A Code Quality Perspective

Md Rakibul Islam
University of New Orleans, USA
Email: mislam3@uno.edu

Minhaz F. Zibran
University of New Orleans, USA
Email: zibran@cs.uno.edu

***Abstract*—Code clone is an immensely studied code smell. Not all the clones in a software system are equally harmful. Earlier work studied various traits of clones including their stability and relationships with program faults against non-cloned code. This paper presents a comparative study on the characteristics of buggy and non-buggy clones from a code quality perspective.**

In the light of 29 code quality metrics, we study buggy and non-buggy clones in 2,077 revisions of three software systems written in Java. The findings from this work add to the characterization of buggy clones. Such a characterization will be useful in cost-effective clone management and clone-aware software development.

I. Introduction

Source code reuse by copy-paste is a common practice that software developers adopt to increase productivity. Software systems typically have 9%-17% [48] duplicated code, and the proportion is sometimes found to be even 50% [35] or higher [6]. Such duplicated code, known as code clones, are considered notorious code smell [7] due to their negative impacts on software development and maintenance.

In the past, several studies examined the comparative stability of clones as opposed to non-cloned code [4], [8], [11], [21], [22], [28], [33], comparative vulnerabilities in cloned and non-cloned code [15], [16], relationships of clones with bug-fixing changes [5], [12], [17], [18], [19], [23], [32], [41], [45], change-proneness of clones [29], and the impacts of clones on program's changeability [9], [24], [25]. There have also been studies on clone removal in program history [38], [49], [50]. Existing literature suggest that, (i) clones are problematic in many cases, (ii) not all clones are equally harmful [20], and (iii) it is not practically feasible to remove *all* clones from a system through aggressive refactoring [37], [46], [47]. Therefore, for cost-effective clone management, we must distinguish the characteristics of clones, which make them problematic (e.g., buggy).

Not much work is done along this direction. Majority of earlier work made comparisons between clones and non-cloned code with respect to certain criteria (e.g., stability, vulnerability, bug-proneness). Only a few earlier studies suggested merely a handful of characteristics that might make certain clones detrimental. Such characteristics include late-propagation of clones [4], [5] and stability/change-proneness of clones [33], [29].

In this paper, using 29 code quality metrics, we study the characteristics of buggy and non-buggy clones aiming to identify certain quality metrics, which can indicate potential bug-proneness of clones. In particular, we address the following four research questions.

RQ1: *Are buggy cloned methods more complex than non-buggy cloned methods or vice versa*? —Failure-prone software entities are statistically correlated with code complexity measures [30]. However, there is no single set of complexity metrics that can indicate defect [30]. Here, we investigate 15 complexity metrics to statistically measure their dominance in buggy and non-buggy cloned code.

RQ2: *Are buggy cloned methods larger than non-buggy cloned methods or vice-versa?* — Earlier studies [39], [48] found cloned methods to be smaller than the non-cloned methods. However, it is unknown whether there is any substantial size difference between buggy and non-buggy clones. Using seven metrics, we investigate the size difference between buggy and non-buggy cloned methods.

RQ3: *Are buggy cloned methods more documented than non-buggy cloned methods or vice-versa?* — Its is suggested that when cloning a piece of code the variations should be well documented in order to facilitate bug fix propagation [20]. As such, undocumented or poorly documented clones can have high possibility of causing bugs. Hence, we investigate this possibility by analyzing five source code documentation metrics.

RQ4: *Are buggy cloned methods more coupled than non-buggy cloned methods?* — Low coupling and high cohesion are two prominent software design qualities expected to keep a software system's inherent complexity manageable. In other words, highly coupled code is harder to manage and could be bug-prone. Thus, we investigate whether coupling metrics' values are significantly higher in buggy cloned code compared to non-buggy cloned code.

II. Study Setup

The procedural steps of our empirical study are summarized in Figure 1, and described in the following subsections.

A. Subject Systems

We study 2,077 revisions of three open-source software systems written in Java. These subject systems, as listed in Table I, are available at the GitHub repository. In Table I,

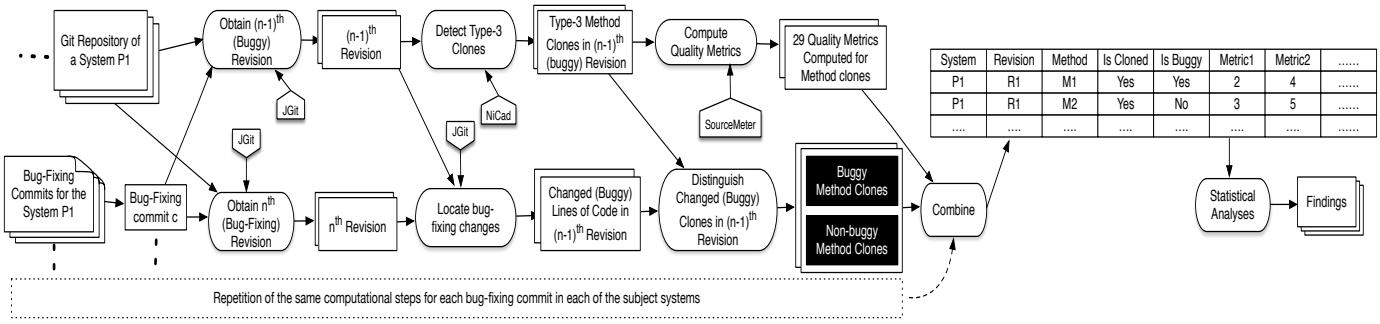

Fig. 1. Procedural Steps of the Empirical Study

we present the total number of revisions and number of bug-fixing revisions of each subject system along with the number of source lines of code (LOC) in the last revision. We choose these three subject systems as these systems have variations in application domains, sizes, number of revisions, and are also used in another study [34].

TABLE I
SUBJECT SYSTEMS

Subject System	Application Domain	LOC (last rev.)	Total # of Revisions	# of Bug-Fixing Rev.
Netty	Network	1,078,493	8,534	1,103
Presto	SQL	2,869,799	11,909	841
Facebook-android-SDK	Social Networking	172,695	671	133
Total over all the systems		4,120,987	21,114	2,077

B. Clone Detection

Code clones at different levels of syntactic similarities appear in source code. Identical pieces of source code with or without variations in whitespaces (i.e., layout) and comments are called *Type-1* clones [37]. *Type-2* clones are syntactically identical code fragments with variations in the names of identifiers, literals, types, layout and comments [37]. Code fragments, which exhibit similarities as of *Type-2* clones and also allow further differences such as additions, deletions or modifications of statements are known as *Type-3* clones [37].

By definition, *Type-3* clones include both *Type-1* and *Type-2*. In this work, we study *Type-3* clones at the granularity of method bodies. We use the `NiCad` [36] clone detector (version 3.5), to detect method/function clones having at least five LOC. In detection clones, 'blind renaming' option of `NiCad` is kept enabled and UPIT (i.e., dissimilarity threshold) is set to 0.3. Further details on `NiCad`'s tuning parameters and their influences on clone detection can be found elsewhere [36].

C. Distinguishing Buggy Clones

Consider a bug-fixing commit c resulting in the n^{th} revision of a system. If a particular method m is modified in the bug-fixing commit c, then it implies that the modification is necessary to fix the bug. Thus, the method m in the $(n-1)^{\text{th}}$ revision is considered a buggy method. In this work, we use the bug-fixing commits identified by Ray et al. [34]. These bug-fixing commits are distinguished through matching keywords (e.g., bug, defect, issue) in the commit messages and are reported to be 96% accurate [34].

For a project $\mathcal{P}$, we determine the buggy cloned methods by performing the following steps: (i) we collect all the bug-fixing commits for $\mathcal{P}$ from the dataset of Ray et al. [34]. (ii) For each bug-fixing commit c, using `JGit` [1], we obtain the n^{th} revision of $\mathcal{P}$, which is the result of the commit c. (iii) Then, we obtain the $(n-1)^{\text{th}}$ revision of $\mathcal{P}$. (iv) The changes between the n^{th} and $(n-1)^{\text{th}}$ revisions of $\mathcal{P}$ are captured using `JGit` along with changed lines' numbers in the java source files. (v) Using `NiCad` [36], we detect *Type-3* method clones in the $(n-1)^{\text{th}}$ revisions of $\mathcal{P}$ and record their locations and boundaries (start and end line numbers in source files). (vi) Whether a commit c affected a cloned method is determined by checking if any of changed lines' number identified in the step-iv falls within the boundary of the clone. (vii) Clones that are affected by the bug-fixing commits are identified as buggy clones while the rest other clones are considered non-buggy.

Several other studies [13], [14], [29], [32] also adopted similar approaches for distinguishing buggy source code.

D. Computation of Source Code Quality Metrics

We use total 29 source code quality metrics grouped into four categories- (i) *complexity metrics*, which measure the complexity of source code elements; (ii) *size metrics*, which measure the basic properties of the analyzed system in terms of different cardinalities (e.g., number of code lines). (iii) *documentation metrics*, which measure the amount of comments and documentation of source code elements in the system; and (iv) *coupling metrics*, which measure the interdependencies of source code elements. All the metrics are listed and briefly described in Table II. Detail descriptions of those metrics can be found in the user manual of `SourceMeter` [2], which is a proprietary static source code analyzer tool we use in this work. We opt out of describing the metrics in details due to space limitations.

Using a free version of `SourceMeter` [2] (version 8.2.0-x64-linux), we compute all the metrics in Table II for each of the buggy and non-buggy clones. The tool's configuration parameters are kept at their defaults.

III. ANALYSIS AND FINDINGS

We carry out our analyses in the light of each of the 29 quality metrics separately averaged for buggy and non-buggy clones. For testing statistical significance we apply *Mann-*

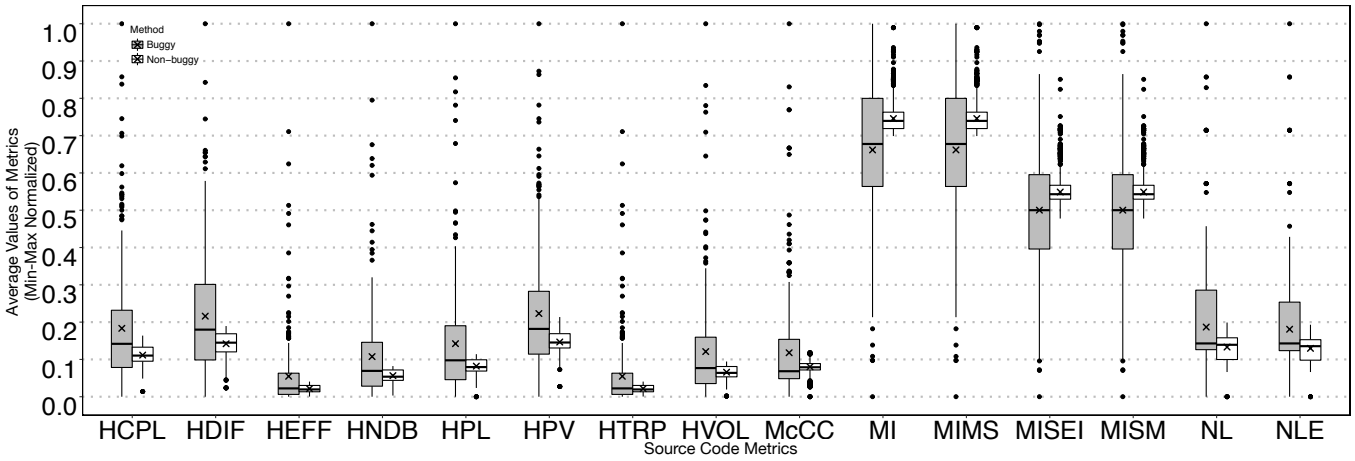

Fig. 2. Distribution of Complexity Metrics' Values in Buggy and Non-buggy Cloned Code

TABLE II
SOURCE CODE QUALITY METRICS USED IN THIS STUDY

Category	Metric	Description
Complexity metrics	↓ HCPL	Hal. Calculated Program Length
	↓ HDIF	Hal. Difficulty
	↓ HEFF	Hal. Effort
	↓ HNDB	Hal. Number of Delivered Bugs
	↓ HPL	Hal. Program Length
	↓ HPV	Hal. Program Vocabulary
	↓ HTRP	Hal. Time Required to Program
	↓ HVOL	Hal. Volume
	↑ MIMS	Maintainability Index (MS)
	↑ MI	Maintainability Index (OV)
	↑ MISEI	Maintainability Index (SEIV)
	↑ MISM	Maintainability Index (SV)
	↓ McCC	McCabe's Cyclomatic Complexity
	↓ NL	Nesting Level
	↓ NLE	Nesting Level Else-If
Size metrics	↓ LOC	Lines of Code
	↓ LLOC	Logical Lines of Code
	↓ NUMPAR	Number of Parameter
	↓ NOS	Number of Statements
	↓ TLOC	Total Lines of Code
	↓ TLLOC	Total Logical Lines of Code
	↓ TNOS	Total Number of Statements
Documentation metrics	↑ CD	Comment Density
	↑ CLOC	Comment Lines of Code
	↑ DLOC	Documentation Lines of Code
	↑ TCD	Total Comment Density
	↑ TCLOC	Total Comment Lines of Code
Coupling metrics	↓ NII	Number of Incoming Invocations
	↓ NOI	Number of Outgoing Invocations

Hal.=Halstead; MS= Microsoft version; OV=Original version
SEIV=SEI version; SV=SourceMeter version.
↑ by a metric indicates the higher the better for that metric
↓ by a metric indicates the lower the better for that metric.

Whitney-Wilcoxon (MWW) test [3] with $\alpha = 0.05$. To measure effect size, we compute *Cliff's delta* d [3].

Both of these non-parametric statistical tests do not require normal distribution of data, and thus suit well for our purpose. We consider *significant difference* exists between distributions if p-value of a *MWW* test is found to be less than α and *Cliff's delta* d value is not negligible (i.e., $|d| > 0.15$). In drawing the box-plots for our analyses, we normalize the metrics values using the widely used *Min-Max* [42] method.

A. Complexity of Buggy and Non-buggy Clones

In the box plot of Figure 2, we present the distribution of the average complexity metrics' values for buggy (grey boxes) and non-buggy (white boxes) clones for each studied revision of the subject systems. The 'x' marks in the box plots indicate the means over all the revisions across the subject systems.

As seen in Figure 2, for buggy clones, there are more variations in values of all the complexity metrics compared to those for non-buggy clones. Most importantly, considering the averages (marked with 'x'), all the 15 complexity metrics' values are worse for buggy clones. Buggy clones exhibit lower values for the four *maintenance index* related complexity metrics (i.e., MI, MIMS, MISEI, and MISM). For these four metrics higher values are desirable as indicated in Table II. For the rest 11 complexity metrics, buggy clones are found to have higher values while lower values are desirable for these 11 metrics. These observations indicate that buggy clones are more complex and less maintainable compared to non-buggy clones.

TABLE III
MWW TESTS OVER THE DISTRIBUTION OF *Complexity* METRICS FOR BUGGY AND NON-BUGGY CLONES

Source Code Metrics	P-value	Cliff's delta d	Significant?
HCPL	7.011×10^{-12}	0.2297 (small)	Yes
HDIF	2.23×10^{-11}	0.2239 (small)	Yes
HEFF	0.2787	Not applicable	No
HNDB	6.017×-05	0.1307 (negligible)	No
HPL	1.23×10^{-06}	0.1601 (small)	Yes
HPV	9.75×10^{-14}	0.2499 (small)	Yes
HTRP	0.5575	Not applicable	No
HVOL	2.75×10^{-05}	0.1371 (negligible)	No
MI	2.20×10^{-16}	-0.2882 (small)	Yes
MIMS	2.20×10^{-16}	-0.2882 (small)	Yes
MISEI	1.48×10^{-14}	-0.2614 (small)	Yes
MISM	1.48×10^{-14}	-0.2614 (small)	Yes
McCC	0.1872	Not applicable	No
NL	4.17×10^{-07}	0.1675 (small)	Yes
NLE	1.65×10^{-10}	0.2136 (small)	Yes

To determine whether our observations are statistically significant, for each of the 15 complexity metrics, we separately conduct a one-sided pair-wise *MWW* test between the metric's values computed for buggy and non-buggy clones. The p-values obtained from these tests are presented in Table III. The measurements of effect sizes (i.e., *Cliff's delta* d) corresponding to the MWW tests are also included in Table III.

As seen in Table III, the p-values obtained from MWW tests are less than α for all the complexity metrics except for three (HEFF, HTRP, and McCC). However, for two (HNDB and HVOL) of these 12 complexity metrics, the effect sizes computed in *Cliff's delta* d are found to be negligible. For rest of the 10 complexity metrics, the MWW tests indicate statistical significance in the differences of the metrics' values for buggy and non-buggy clones while the *Cliff's delta* d values also suggest that the effect sizes are not negligible.

Hence, from our observations and statistical tests, we now derive the answer to *RQ1* as follows:

Ans. to RQ1: *Buggy clones have significantly higher complexity and lower maintainability compared to non-buggy code clones.*

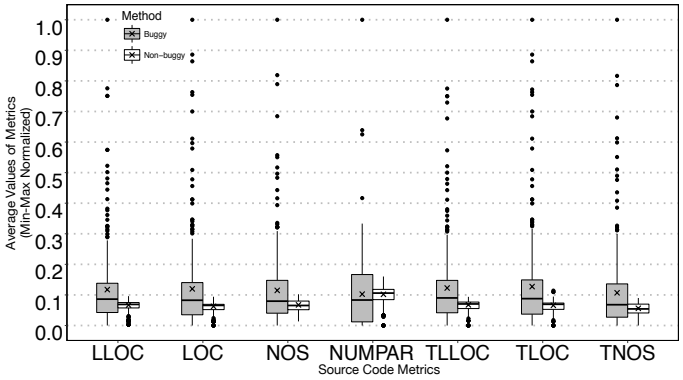

Fig. 3. Size Metrics' Values in Buggy and Non-buggy Clones

B. Size Difference of Buggy and Non-buggy Clones

In Figure 3, we plot the distribution of the seven size metrics' values computed for buggy and non-buggy clones. Similar to the case of complexity metrics (discussed in Section III-A), we see that the variations in all the size metrics is higher in buggy clones than in non-buggy ones.

TABLE IV
MWW TESTS OVER AVERAGE VALUES OF *Size* METRICS IN BUGGY AND NON-BUGGY CLONED CODE

Source Code Metrics	P-Value	Cliff's delta d	Significant?
LLOC	1.34×10^{-10}	0.2147 (small)	Yes
LOC	1.47×10^{-11}	0.2260 (small)	Yes
NOS	5.48×10^{-06}	0.1500 (small)	Yes
NUMPAR	3.44×10^{-06}	−0.1529 (small)	Yes
TLLOC	8.88×10^{-11}	0.2169 (small)	Yes
TLOC	1.12×10^{-11}	0.2275 (small)	Yes
TNOS	1.94×10^{-06}	0.1570 (small)	Yes

Average values of all the size metrics are higher (i.e., worse) for buggy method clones except for NUMPAR (i.e., number of parameters). Surprisingly, the average value of this particular metric appear to be slightly higher for non-buggy clones. Again, to verify the significance of our observations and effect size, we conduct one-sided pair-wise *MWW* test and *Cliff's delta* d for each of the seven size metrics between their values computed for buggy and non-buggy clones. The results of the tests are presented in Table IV. The results in Table IV suggest statistical significance (with non-negligible effect size) in the differences of all the seven size metrics' values computed for buggy and non-buggy clones. Based on the findings, we now answer the *RQ2* as follows:

Ans. to RQ2: *Compared to non-buggy clones, the buggy method clones have significantly higher code size measured in terms of the number of lines and statements. Surprisingly, non-buggy cloned methods are found to have higher number of parameters.*

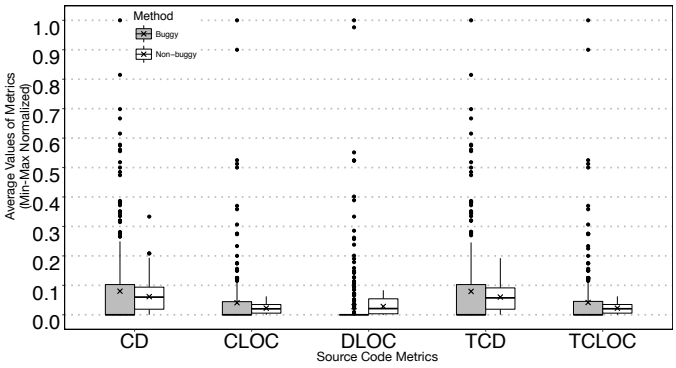

Fig. 4. Documentation Metrics' Values in Buggy and Non-buggy Clones

C. Documentation in Buggy and Non-buggy Clones

Figure 4 depicts the distribution of average values of the five documentation metrics for buggy and non-buggy clones. As seen in the figure, the medians of all the documentation metrics' values are consistently *higher* (better) for non-buggy clones. On the contrary, for non-buggy clones, the mean values are *lower* (worse) for all the metrics except for DLOC (Documentation Lines of Code).

TABLE V
MWW TESTS OVER AVERAGE VALUES OF *Documentation* METRICS IN BUGGY AND NON-BUGGY CLONED CODE

Source Code Metrics	P-Value	Cliff's delta d	Significant?
CD	2.20×10^{-16}	−0.2809 (small)	Yes
CLOC	3.68×10^{-16}	−0.277 (small)	Yes
DLOC	2.20×10^{-16}	−0.7243 (**large**)	Yes
TCD	9.58×10^{-16}	−0.2730 (small)	Yes
TCLOC	3.12×10^{-15}	−0.2680 (small)	Yes

Now, for the distributions of each of the five documentation metrics computed for buggy and non-buggy clones, we separately conduct a one-sided pair-wise *MWW* test and also measure effect size using *Cliff's delta* d to test our hypothesis that buggy clones have inferior documentation (i.e., lower documentation metrics value) than the non-buggy clones. The

obtained p-values and *Cliff's delta* d values are presented in Table V. As we see in Table V, for all the five documentation metrics, the p-values indicate statistical significance and *Cliff's delta* d values suggest non-negligible effect sizes. Thus the statistical tests fail to reject our hypotheses for each of the documentation metrics. We, therefore, answer the research question *RQ3* as follows:

Ans. to RQ3: *The quality of documentation in buggy clones is significantly inferior to that in non-buggy clones.*

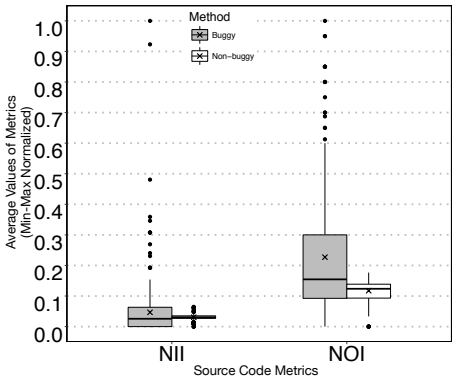

Fig. 5. Coupling Metrics' Values in Buggy and Non-buggy Clones

D. Coupling in Buggy and Non-buggy Clones

In Figure 5, we plot the distribution of average values of the two coupling metrics for buggy and non-buggy clones. As we see in Figure 5, for both the coupling metrics, the variations in buggy clones are higher compared to non-buggy clones. Again, for both the metrics, averages are higher for buggy clones, and the difference is more substantial for the NOI (Number of Outgoing Invocations) metric. The median for NOI is higher for buggy clones while for the NII (Number of Incoming Invocations) metric the medians are nearly equal for both buggy and non-buggy clones.

TABLE VI
MWW TESTS OVER AVERAGE VALUES OF *Coupling* METRICS IN BUGGY AND NON-BUGGY METHOD CLONES

Source Code Metrics	P**-Value**	**Cliff's delta** d	**Significant?**
NII	0.006414	-0.0926 (negligible)	No
NOI	2.20×10^{-16}	0.2972 (small)	Yes

Similar to previous analyses, we conduct one-sided pair-wise *MWW* test and compute *Cliff's delta* d separately for each of the coupling metrics to test the hypothesis that buggy clones are highly coupled (i.e., have higher coupling metrics' values) than the non-buggy clones. The results of the tests are presented in Table VI. As seen in the table, the p-values of MWW tests for both the metrics indicate statistical significance with $p < \alpha$. However, the *Cliff's delta* d values indicate non-negligible effect size for NOI, but negligible effect size for the NII metric.

In combination of the two metrics, we can say that buggy method clones are comparatively more coupled than non-buggy clones. Hence, we derive the answer to the research question *RQ4* as follows:

Ans. to RQ4: *Buggy method clones have significantly higher number of outgoing dependencies (i.e., outgoing invocations) compared to non-buggy method clones. However, there is no significant differences in the incoming dependencies (i.e., incoming invocations) of buggy and non-buggy cloned methods. Overall, buggy clones are more coupled than non-buggy clones.*

IV. THREATS TO VALIDITY

Construct Validity: We use the bug-fixing commits that are identified by Ray et al. [34]. These bug-fixing commits are distinguished based on matching of keywords (e.g., bug-fix, bug, issue, bug id) in the commit messages. There is a possibility that some portions of the bug-fixing commits could be regular commits (e.g., relevant to new feature implementations and improvements) and might not be genuinely relevant to bug-fix. However, this dataset of bug-fixing commits is reported to be 96% accurate [34]. In the identification of bug-fixing changes affected by a bug-fixing commit, there is a possibility that not all the affected lines of code are indeed responsible for the bug fixed in the bug-fixing commit. However, for this purpose, this is an acceptable approach also widely adopted in other studies [13], [14], [29], [32].

When a cloned method m is affected by bug-fixing changes in revision n, the method m is considered buggy at revisions $n-1$. For rest of the revisions the method m is considered non-buggy. The content of the method m can safely considered non-buggy in any later revision ρ with $\rho > n$ since the bug is fixed in revision n. However, prior to the bug-fixing revision n, the method m may or may not be buggy, but in our work, we considered it non-buggy. This assumption can be argued as a threat to the construct validity of this work.

Internal Validity: In the detection of clones, we have used the `NiCad` clone detector, which is reported to be highly accurate [36], [43] and is widely used in many studies [13], [33], [48], [47]. The library `JGit` used in our work for locating changes between two revisions is also reliably used for similar purposes in other studies [10], [31]. Moreover, we manually verified the correctness of the computations probing with random samples. Thus, we develop high confidence in the internal validity of this study.

External Validity: Although our study includes a large number of revisions of three subject systems, all the systems are open-source and written in Java. Thus the findings from this work may not be generalizable for industrial systems and source code written in languages other than Java.

Reliability: The methodology of this study including the procedure for data collection and analysis is documented in this paper. The subject systems being open-source, are freely accessible while the tool `NiCad` and library `JGit` are available online. All the bug-fixing commits are also available [34]. Therefore, it should be possible to replicate this study.

V. Related Work

Several attempts are made to explore fault-proneness of clones by relating them with bug-fixing changes obtained from commit history. Very recently, Mondal et al. [29] claimed that code clones, which were recently changed or created had high possibilities of containing bugs. Again, Rahman and Roy [33] found that stability and bug-proneness of code clones are related. The studies of Mondal et al. and Rahman and Roy only focused on recency of code changes and stability of cloned code respectively. On the other hand, our work, for the first time has investigated a comprehensive set of source code metrics to identify their relationships with bug-proneness of cloned code.

Selim et al. [41] investigated bug-proneness of code clones by combining source code and clone related metrics. However, they used only four source code metrics (e.g., Lines of Code, number of tokens, Nesting levels and Cyclomatic Complexity). Moreover, their investigation was based only on *Type-1* and *Type-2* method clones, thus missed out *Type-3* clones. In contrast to their work, we have used *Type-3* clones for this study that also includes *Type-1* and *Type-2* clones. Moreover, we have used 29 source code metrics to conduct the analyses in this study.

Wang et al. [44] related eight source code metrics with harmfulness of cloned code. They defined harmfulness of a piece of clone code based on maintenance cost. The higher the maintenance cost, the higher the clone code is harmful. In contrast, we have studied real buggy clones identified thorough bug-fixing changes and examined them using 29 software metrics.

Juergens et al. [18] studied inconsistent clones to relate with bugs. They used manual investigation to identify bugs in inconsistent clones, and concluded that unintentionally made inconsistent clones are more likely to contain defects. They hadn't conducted any statistical tests of significance of their finding. In contrast to their approach, we have mined source code repositories to identify bugs and conducted statistical tests of significance for all of our findings.

Rahman et al. [32] compared bug-proneness of clone and non-clone code and found non-clone code to be more bug-prone than clone code. However, their investigation was based on monthly snapshots of their subject systems, and thus, they had the possibility of missing buggy commits. In our study, we consider all the bug-fix revisions for each of the subject systems, thus, we have included all bug-fix commits.

Barbour et al. [5] suggested that late propagations due to inconsistent changes are prone to introduce software defects. While Lozano and Wermelinger [24] suggested that having a clone may increase the maintenance effort for changing a method, Hotta et al. [11] reported code clones not to have any negative impact on software changeability. Lozano et al. [25] reported that a vast majority of methods experience larger and frequent changes when they contain cloned code. Mondal et al. [28] also reported code clones to be less stable. However, opposite results are found from several other studies [4], [9], [8], [21].

Recently, Islam et al. [16] conducted a comparative study of *security vulnerabilities* in cloned and non-cloned code. Earlier Islam and Zibran [15] and Sajnani et al. [40] conducted two comparative studies of *code smells* in cloned and non-cloned code. However, they defined *code smells* as *vulnerability* and *bug patterns* in their respective studies. By targeting *code stability* and *dispersion of code changes* Mondal et al. also performed comparative studies in cloned and non-cloned code [26], [27].

Along the comparative studies, Saini et al. [39] conducted a study to compare source code quality metrics for cloned and non-cloned code. The study used 27 software quality metrics, categorized in three groups e.g., complexity, modularity, and documentation (code-comments). They did not find any statistically significant difference between the quality of cloned and non cloned methods for most of the metrics. However, we have compared the significance of differences of 29 software quality metrics' (categorized in four groups) values in buggy and non-buggy cloned methods instead of comparing them in cloned and non-cloned code.

In another study, Islam et al. [14] found the percentage of changed files due to bug-fix commits is significantly higher in clone code compared with non-clone code. They also found the possibility of severe bugs occurring is higher in clone code than in non-clone code. While all these above studies provide valuable insights about the characteristics of code clones, the clone literature lacked the comparative study of buggy and non-buggy cloned code characteristics in terms of source code quality metrics. This study has filled that gap in some extent and identify which source code quality metrics have contributed to buggy cloned code.

VI. Conclusion

In this paper, we have presented a comparative study of buggy and non-buggy *Type-3* method clones in terms of 29 code quality metrics grouped into complexity metrics, size metrics, documentation metrics, and coupling metrics. Our quantitative study is based on 2,077 bug-fixing revisions of three open-source software systems written in Java.

In our study, we have found that buggy clones have higher complexity and lower maintainability compared to non-buggy cloned methods. Moreover, buggy clones are found to be larger in size measured in terms of the number statements and lines of code. Surprisingly, we have found that the non-buggy method clones have higher number of parameters while too many parameters in functions are generally considered problematic and recognized as a code smell [7].

As expected, compared to non-buggy clones, documentation quality of buggy clones are found inferior. Overall, buggy clones are found to be more coupled than non-buggy clones. However, it is interesting to have found that the buggy cloned methods have significantly higher outgoing dependencies (i.e., outgoing invocations) compared to non-buggy clones. In case of incoming dependencies, no significant difference is found between buggy and non-buggy clones.

The results are validated with statistical tests of significance. However, there is a need for qualitative analyses to draw further insights into the reasons of our findings, especially for case of higher method parameters in non-buggy method clones and the for the case of higher outgoing dependencies in buggy clones. We plan to extend this work with qualitative analyses along with higher number of subject systems and their revisions.

The findings from this study advance our understanding of the characteristics and impacts of buggy clones on code quality. It appears that some of the code quality metrics can be good indicators for potentially buggy clones, and thus can be applied for identifying problematic clones for removal by refactoring or other especial treatments. For doing such, we need to derive a thresholding mechanism, which would indicate at what values of the quality metrics certain clones would be reported as potentially harmful. These remain within our plans for expanding this work.

Acknowledgement: This work is supported in part by the SCoRe grant at the University of New Orleans.

REFERENCES

[1] *JGit*. https://www.eclipse.org/jgit/, verified: Jan 2018.
[2] *SourceMeter: Static source code analysis solution for Java, C/C++, C#, Python and RPG*. https://www.sourcemeter.com, verified: Jan 2018.
[3] D. Anderson, D. Sweeney, and T. Williams. *Statistics for Business and Economics*. Thomson Higher Education, 10th edition, 2009.
[4] L. Aversano, L. Cerulo, and M. Di Penta. How clones are maintained: An empirical study. In *CSMR*, pages 81–90, 2007.
[5] L. Barbour, F. Khomh, and Y. Zou. Late propagation in software clones. In *ICSM*, pages 273–282, 2011.
[6] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109 –118, 1999.
[7] M. Fowler, K. Beck, J.Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
[8] N. Göde and J. Harder. Clone stability. In *CSMR*, pages 65–74, 2011.
[9] N. Göde and R. Koschke. Frequency and risks of changes to clones. In *ICSE*, pages 311–320, 2011.
[10] G. Greene and B. Fischer. Cvexplorer: Identifying candidate developers by mining and exploring their open source contributions. In *ASE*, pages 804–809, 2016.
[11] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is duplicate code more frequently modified than non-duplicate code in softw. evolution?: an emp. study on open source softw. In *IWPSE-EVOL*, pages 73–82, 2010.
[12] 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 *IWSC*, pages 94–95, 2012.
[13] J. Islam, M. Mondal, and C. Roy. Bug replication in code clones: An empirical study. In *SANER*, 2016.
[14] J. Islam, M. Mondal, C. Roy, and K. Schneider. A comparative study of software bugs in clone and non-clone code. In *SEKE*, 2017.
[15] M. Islam and M. Zibran. A comparative study on vulnerabilities in categories of clones and non-cloned code. In *IWSC*, pages 8–14, 2016.
[16] M. Islam, M. Zibran, and A. Nagpal. Security vulnerabilities in categories of clones and non-cloned code: An empirical study. In *ESEM*, pages 20–29, 2017.
[17] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *ESEC-FSE*, pages 55–64, 2007.
[18] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *ICSE*, pages 485–495, 2009.
[19] E. Juergens, B. Hummel, F. Deissenboeck, and M. Feilkas. Static bug detection through analysis of inconsistent clones. In *TESO*, pages 443–446, 2008.
[20] C. Kapser and M. Godfrey. "Cloning considered harmful" considered harmful: patterns of cloning in software. *Empirical Software Engineering*, 13:645–692, 2008.
[21] J. Krinke. Is cloned code more stable than non-cloned code? In *SCAM*, pages 57–66, 2008.
[22] J. Krinke. Is cloned code older than non-cloned code? In *IWSC*, pages 28–33, 2011.
[23] J. Li and M. Ernst. CBCD: Cloned buggy code detector. In *ICSE*, pages 310–320, 2012.
[24] A. Lozano and M. Wermelinger. Assessing the effect of clones on changeability. In *ICSM*, pages 227–236, 2008.
[25] A. Lozano, M. Wermelinger, and B. Nuseibeh. Evaluating the harmfulness of cloning: A change based exp. In *MSR*, pages 18–21, 2007.
[26] M. Mondal, C. Roy, M. Rahman, R. Saha, J. Krinke, and K. Schneider. Comparative stability of cloned and non-cloned code: An empirical study. In *ACM-SAC (SE Track)*, pages 1227–1234, 2012.
[27] M. Mondal, C. Roy, and K. Schneider. Dispersion of changes in cloned and non-cloned code. In *IWSC*, pages 29–35, 2012.
[28] M. Mondal, C. Roy, and K. Schneider. An empirical study on clone stability. *ACM Applied Computing Review*, 12(3):20–36, 2012.
[29] M. Mondal, C. K. Roy, and K. A. Schneider. Identifying code clones having high possibilities of containing bugs. In *ICPC*, 2017.
[30] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *2006*, pages 452–461, ICSE.
[31] H. Osman, M. Lungu, and O. Nierstrasz. Mining frequent bug-fix code changes. In *CSMR-WCRE*, pages 343–347, 2014.
[32] F. Rahman, C. Bird, and P. Devanbu. Clones: what is that smell? In *MSR*, pages 72–81, 2010.
[33] M. Rahman and C. Roy. On the relationships between stability and bug-proneness of code clones: An empirical study. In *SCAM*, pages 131–140, 2017.
[34] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu. On the "naturalness" of buggy code. In *ICSE*, pages 428–439, 2016.
[35] M. Rieger, S. Ducasse, and M. Lanza. Insights into system-wide code duplication. In *WCRE*, pages 100–109, 2004.
[36] C. Roy and J. Cordy. NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *ICPC*, pages 172–181, 2008.
[37] C. Roy, M. Zibran, and R. Koschke. The vision of software clone management: Past, present, and future. In *CSMR-18/WCRE-21 Software Evolution Week (SEW'14)*, pages 18–33, 2014.
[38] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, and K. Schneider. Evaluating code clone genealogies at release level: An empirical study. In *SCAM*, pages 87–96, 2010.
[39] V. Saini, H. Sajnani, and C. Lopes. Comparing quality metrics for cloned and non-cloned java methods: A large scale empirical study. In *ICSME*, pages 256–266, 2016.
[40] H. Sajnani, V. Saini, and C. Lopes. A comparative study of bug patterns in java cloned and non-cloned code. In *SCAM*, pages 21–30, 2014.
[41] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, and Y. Zou. Studying the impact of clones on software defects. In *WCRE*, pages 13–21, 2010.
[42] R. Snelick, A. Mink, M. Indovina, and A. Jain. Large-scale evaluation of multimodal biometric authentication using state-of-the-art systems. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 27(3):450–455, 2005.
[43] J. Svajlenko and C. K. Roy. Evaluating modern clone detection tools. In *ICSME*, pages 321 – 330, 2014.
[44] X. Wang, Y. Dang, L. Zhang, D. Zhang, E. Lan, and H. Mei. Can i clone this piece of code here? In *ASE*, pages 170–179, 2012.
[45] S. Xie, F. Khomh, and Y. Zou. An empirical study of the fault-proneness of clone mutation and clone migration. In *MSR*, pages 149–158, 2013.
[46] M. Zibran and C. Roy. Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach. In *ICPC*, pages 266 – 269, 2011.
[47] M. Zibran and C. Roy. Conflict-aware optimal scheduling of code clone refactoring. *IET Software*, 7(3):167–186, 2013.
[48] M. Zibran, R. Saha, M. Asaduzzaman, and C. Roy. Analyzing and forecasting near-miss clones in evolving software: An empirical study. In *ICECCS*, pages 295–304, 2011.
[49] M. Zibran, R. Saha, C. Roy, and K. Schneider. Evaluating the conventional wisdom in clone removal: A genealogy-based empirical study. In *ACM-SAC (SE Track)*, pages 1123–1130, 2013.
[50] M. Zibran, R. Saha, C. Roy, and K. Schneider. Genealogical insights into the facts and fictions of clone removal. *ACM Applied Computing Review*, 13(4):30–42, 2013.