An Insight into Subsequent Bug-Resolution Commits

Abstract—**Bug-resolution is one of the most challenging and time-consuming tasks of software development and maintenance. The challenge exacerbates when multiple iterations of code changes are needed to solve a single bug. Although bug-resolution has been an old topic, to date, subsequent bug-resolution commits have not been well studied. In this paper, we analyze 66,261 bug-resolution commits from 1,000 software projects, identify both parent and subsequent commits from them, and then study the interplay between types of bug and bug-resolution commits. We report that developers fail more often to localize bugs if a replacement operation (e.g., replacing a string with another) is involved in bug-resolution. Furthermore, single-statement bugs such as Change Identifier Used, Wrong Function Name, Change Numeric Literal, and Change Modifier are associated with 74% of the subsequent bug-resolution commits. Since subsequent bug-resolution is a common phenomenon, our work has the potential to deliver valuable, actionable insights to both software practitioners and researchers.**

I. Introduction

Bug resolution is one of the crucial tasks of software development and maintenance. Reportedly, it consumes ≈50% of the development time, costing billions of dollars each year [1]. In 2017 alone, 606 software bugs and failures in 314 companies caused a financial loss of $1.7 trillion affecting 3.6 billion people [2].

Once a bug report is submitted, the assigned developer localizes the bug, changes the faulty code to resolve the bug, and then submits the changed code to a version control system (e.g., Git). However, a single set of changes (a.k.a., commit) might always not be sufficient to resolve a bug. Often subsequent commits are required to resolve the bug altogether. For example, once a developer modifies the faulty code and submits it for code reviews, many corner-cases could emerge, and subsequent patches might be required to resolve them. It is also common to inaccurately misinterpret particular code as faulty and then modify, which is likely to introduce new bugs during these subsequent commits.

Even though each bug is unique at its core, we suspect that bug-resolving commit-sequences share some contextual similarity among each other. Having a firm grip on this contextual information (e.g., the type of the bug being resolved) is essential to avoid similar commit sequences in the future.

In this paper, we report an exploratory study using 66,261 bug-resolving commits from 1,000 open-source projects and analyze the nature of subsequent bug-resolving commits. We construct a set of trees based on the parent-child relationship of the commits and then identify the commit sequences that are sufficiently long. We analyze these sequences both statistically and manually in order to answer two research questions as follows.

- **RQ$_1$**: In what context developers either (a) introduce new bugs or (b) leave similar bugs unsolved during bug resolution?
- **RQ$_2$**: Do certain types of single-statement bugs appear more in subsequent bug-resolving commits?

We identify that developers poorly perform when replacement operations (e.g., replacing an identifier, string, or number with another) are involved in the bug resolution, which is likely to leave similar bugs unsolved. Furthermore, while dealing with overloaded methods or variables with similar types and usage, developers introduce new bugs by misusing variables or methods.

We also spot out that among the 16 bug-types, *Change Identifier Used*, *Wrong Function Name*, *Change Numeric Literal*, and *Change Modifier* are associated with 74% of the subsequent bug-resolving changes. They are also more likely to trigger subsequent bug-resolving commits than the other types.

II. Experimental Dataset

In the MSR challenge 2021 dataset, *bugs_large* [3] contains 153,652 single-statement bug-resolving changes in 66,261 commits that are mined from 1,000 popular open-source Java projects. About 41.60% (63,923) of these changes match with at least one of the 16 predefined bug templates, which make up a subset dataset, namely *sstubs_large*. Code changes from this dataset are distributed across 24,486 commits. We use these two datasets in this work. For each change in the datasets, the corresponding source code compiles both before and after the bug-resolution commit. Thus, even after resolving a bug, new bugs could be hidden in the compiling code, which could be challenging to detect.

The dataset contains 14 features in total. However, we use `fixCommitSHA1`, `fixCommitParentSHA1` and `bugType` in our experiment due to their strong relevance to our research questions. `fixCommitSHA1` is the SHA1 hash of a bug-resolving commit whereas `fixCommitParentSHA1` is the SHA1 hash of its parent commit. The `bugType` value is one of 16 bug types reported by the dataset authors.

III. Experimental Design

Based on our manual investigation, we used an important heuristic in our experiment. That is, if the `fixCommitSHA1` of commit *A* is present as the `fixCommitParentSHA1` of commit B, then we assumed one of two possible scenarios. First, commit A could not resolve the bug properly and thus triggered another change in commit B. Second, while resolving the bug, commit A introduced another bug that was resolved in commit B. Using this heuristic, we explored the parent-child relationships of the commits.

MSR Challenge datasets are given in JavaScript Object Notation (JSON) format. To ease our analysis and improve memory efficiency, we exported the relevant data into an SQLite database. Then, using the `fixCommitSHA1` and `fixCommitParentSHA1` features, we built a forest (i.e. a set of trees) where each commit was treated as a node. Then we analyzed the parent-child relationships and identify the paths with a length of at least three. A path length of three indicates that the first commit is the parent of the second commit, whereas the second commit is a bug-resolving commit. Finally, the third commit is a child of the second commit, which makes it a bug-resolving sequence. We found 528 paths with a length of at least three, and they connect 1,110 commits. We also annotated each of these commits with their depth in the corresponding trees and whether they are part of a bug-resolving sequence or not. Finally, we exported into a CSV file for an in-depth analysis.

We also merged the data from both the SQLite database and the exported CSV based on their `fixCommitSHA1` values. To answer our RQ_1 that involves manual analysis, we extracted all commit-sequences with a minimum length of 5 from the *bugs_large* dataset. We found 33 such paths connecting 270 commits. Then, we manually analyzed 1,308 changes found in these 270 commits to answer our RQ_1. To ease our manual analysis, we built a web-application that facilitates searching with any feature from the dataset and highlighting the changes.

Afterwards, from the *sstubs_large* dataset, we discarded all changes that are not part of a bug-resolving sequence. This step leaves us with 4,593 changes distributed in 1,110 commits. We performed both statistical and manual analysis on these changes to answer RQ_2.

IV. Answering RQ1: Contextual Analysis of Subsequent Bug-Resolution Commits

In this section, we analyzed 1,308 changes grouped in 33 bug-resolving sequences and attempt to find in which contexts subsequent bug-resolving commits appear more. During the analysis, we made four important observations described as follows.

A. Most of the subsequent changes are replacements

Among the 33 bug-resolving sequences, 17 sequences involve different kinds of replacement operations. That is when replacement operation is needed to fix a bug, developers often fail to perform the replacement in all the required places. The problem exacerbates when the replacement operation involves a string. Out of 17 cases, 9 cases were related to string replacement. For example, in the *liferay.liferay-portal* project, to resolve a bug, developers needed to place the string – `"contains(@class,'cke_button cke_button__cut') and"` – in multiple files. Unfortunately, the developers experienced difficulties and the bug took nine commits to get fully resolved.

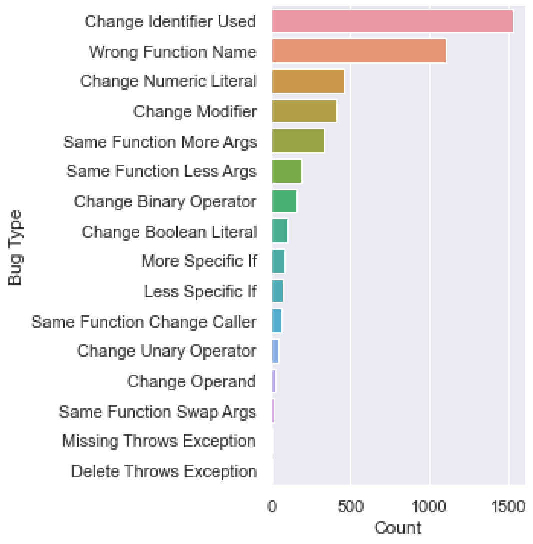

Fig. 1. Occurrence of each bug type in bug-resolving sequences

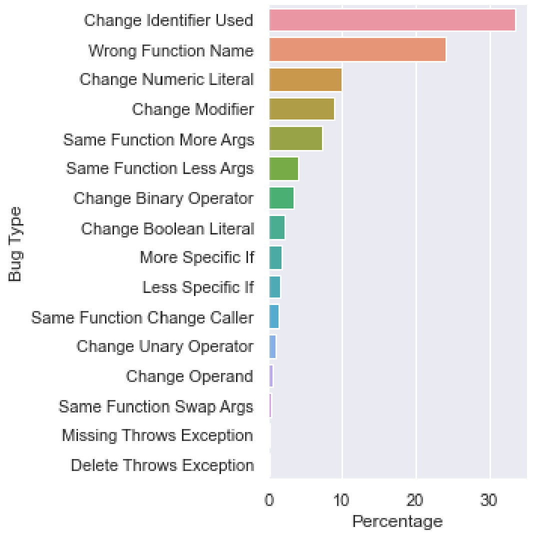

Fig. 2. Percentage of occurrence of each bug types in bug-resolving sequences

B. Overloaded methods and similar variables introduce new bugs

We specifically paid attention to scenarios where its subsequent commit reverts the change in a particular commit. We noticed that this is most common in the case of overloaded methods. For example, in the *koush.ion* project, a particular commit changed `ret.setComplete(response)` to `ret.setComplete(e,response)`. The immediate commit reverted it. New bugs also trigger while dealing with identifiers of similar type and use. For example, in the *aosp-mirror.platform_frameworks_base* project, a commit changed `mActiveAgents.remove(info)` to `mObsoleteAgents.remove(info)` which was reverted by immediate next commit. From the name, we can assume

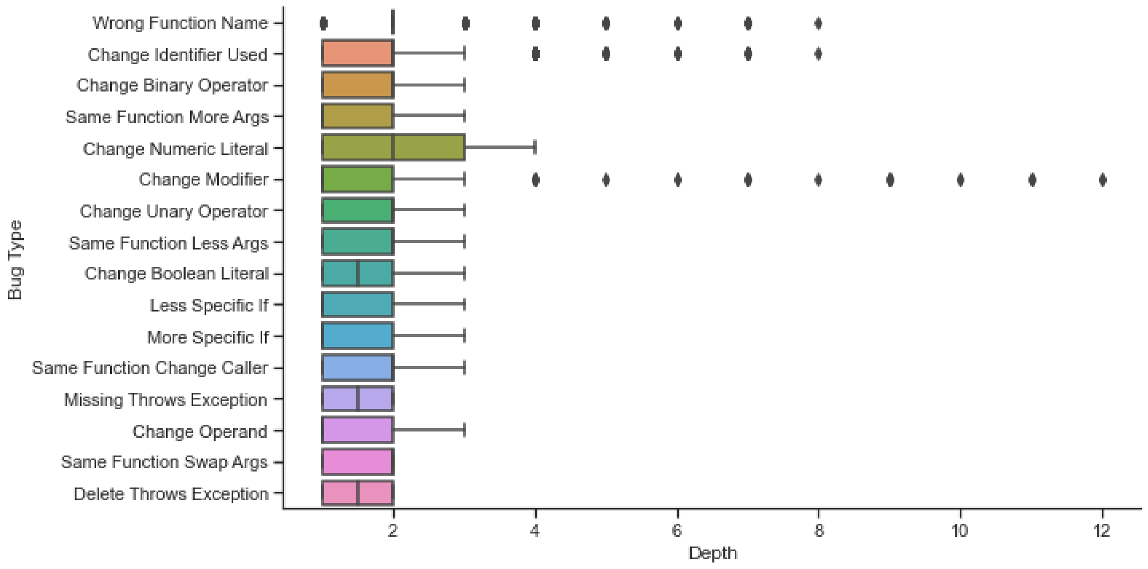

Fig. 3. Box plot of the position where different types of bugs occurred.

that both *mActiveAgents* and `mObsoleteAgents` as similar variable. They should have similar type since both versions of the code compile. In such a scenario, developers get confused about which one to use and introduce bugs.

C. *Multiple sequences representing the same change*

During our analysis, we found multiple sequences that represent the same change in the same project. It indicates that, in some cases, developers might assume that a bug has been resolved due to several iterations (e.g., bug resolution commits) and thus might start working on feature enhancements. However, the bug could emerge, and bug-resolving changes might be added to the feature-related changes. Since the feature related changes were not a part of the dataset, our analysis reported a long commit sequence as two similar bug-resolution sequences.

D. *Intentional commit-sequence*

There were several cases where a commit-sequence was not caused by a failure in bug-resolution. Instead, it was produced intentionally. Sometimes, developers perform similar changes in a document and commit them to keep the changes grouped. For example, in the *liferay.liferay-portal* project, developers needed to replace `@id` with `@name` in multiple documents to resolve a bug. Instead of performing all changes in the same commit, developers made several commits where each commit contains the changes to a single document.

V. Answering RQ2: Statistical Analysis of Commit Sequence Lengths

In this section, we analyze different aspects of bug-resolving commit sequences (e.g., number of changes for each bug-types) to identify which bug-types are more likely to trigger bug-resolving commit-sequences.

A. *Number Of changes*

First, we analyzed the number of times each of the 16 bug types occurs in the bug-resolving sequences. These counts provide us with a rough idea about whether certain types of bugs are more likely to occur than others in the bug-resolving sequences.

Figure 1 shows the number of times changes is subsequent commits were made to solve bugs from each of the 16 bug templates in our dataset. Here, we can see that the first five bug-types – *Change Identifier Used*, *Wrong Function Name*, *Change Numeric Literal*, *Change Modifier*, and *Same Function More Args* – require more subsequent commits than others. Figure 2 shows the ratio of occurrences for each bug type. While Change Identifier Used and Wrong Function Name are the dominant ones causing 33% and 24% changes respectively, the remaining three types caused 9%, 8%, and 7% of changes respectively. Thus, these five types are responsible for about 81% of the changes in the commit sequences. From our manual analysis (Section IV), we also identify that three bug-types – *Change Identifier Used*, *Wrong Function Name*, and *Change Numeric Literal* – are closely related to different types of replacement operations (e.g., replacing one variable with another). Furthermore, we found that developers often fail to perform changes in all necessary places when replacement operation is needed to fix a bug. Thus, it is explainable why these bug types are frequent in bug-resolution commits. Given the above evidence, our empirical and analytical findings also agree with each other.

B. *Variability & Distribution*

Figure 3 shows the box plots for each of the 16 bug types. Here we see that, for all the bug-types, most of their depth values revolve around 1 and 2. For most of the bug types, the third quartile is at or before 3. Interestingly, several bug-types from our above analysis – *Change Identifier Used*, *Wrong*

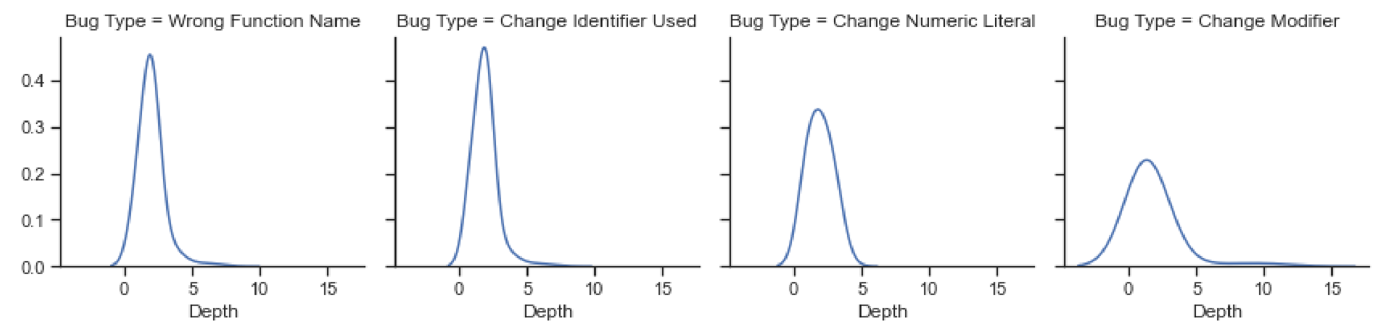

Fig. 4. Kernel density estimation of the four dominant bug-types

Function Name, *Change Numeric Literal*, *Change Modifier* – have the maximum variance in their bug-resolution sequences. This phenomenon suggests that these four bug-types not only have a high likelihood of occurrences but also tend to produce longer bug resolving sequence.

We further visualized the distribution of depth values for the four dominant bug-types using Kernel Density Estimation(KDE) [4]. It shows the distribution of observations in a dataset using a continuous probability density curve. Figure 4 shows the KDE plot for the four bug types mentioned above. From the KDE plots we can see that *Change Identifier Used*, and *Wrong Function Name*, and *Change Modifier* have a long tail towards higher depths. That is, while the majority of mass lies on the depth value of three, these bug-types are likely to occur in the longer bug-resolution sequences.

VI. Key Findings

From our manual analysis in RQ_1 and the statistical analysis of occurrence, variability, and distribution in RQ_2, we can summarize several findings as follows.

Replacement operations are likely to leave similar bugs. While dealing with several bugs such as *Change Identifier Used*, *Wrong Function Name*, *Change Numeric Literal* and *Change Modifier*, developers often perform poorly in bug-resolution, which leads to long sequences of bug-resolving commits. Interestingly, the first three involve different forms of replacement tasks (e.g., replacing a variable with another). From our manual analysis (Section IV), we also report that software developers often fail to detect all the places that need a replacement operation to solve a bug, which leads to subsequent bugs.

Overloaded methods and similar variables introduce new bugs. While dealing with overloaded methods or variables with similar types and usages, developers often misuse them and trigger bugs. Subsequent changes are committed to resolving the new bugs.

Bugs of type *Change Modifier* causes longer sequences. From the box plots in Figure 3, we see that changes related to *Change Modifier* tend to have longer sequences of commits than other bug-types. We manually analyzed the changes from this bug type that have a commit sequence length of more than five to investigate the reason.

We found that these changes were marking a method as static. We assume that after all the feature developments and bug-resolution are done, the developer knows a particular method will not use the `this` variable.

Therefore, the developer might have marked a method as static as the last step of the bug-resolution.

VII. Discussion & Conclusion

Although bug-resolution has been an old topic, to date, subsequent bug-resolution commits have not been well studied. In this paper, we study 66,261 bug-resolving commits from 1,000 open-source projects and examine the nature of subsequent bug-resolving commits. We answer two research questions using a set of commit trees that are constructed from the parent-child relationships among the commits. We find that developers are likely to produce future bugs (and subsequent commits) when replacement operations are needed to fix a bug. Furthermore, among the 16 bug-types that we analyzed, several types – *Change Identifier Used*, *Wrong Function Name*, *Change Numeric Literal*, and *Change Modifier* – are more likely to trigger new bugs and thus long bug-resolution commit sequences. We plan to analyze semantic information from bug resolving changes to determine their contribution to bug-resolving sequences in the future.

References

[1] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbo- gen. 2013. Reversible debugging software. Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep (2013)

[2] Report: Software failure caused \$1.7 trillion in financial losses in 2017, 2019. URL https://tek.io/2FBNl2i.

[3] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset. In Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20). Association for Computing Machinery, New York, NY, USA, 573–577. DOI:https://doi.org/10.1145/3379597.3387491

[4] Parzen, Emanuel. On Estimation of a Probability Density Function and Mode. Ann. Math. Statist. 33 (1962), no. 3, 1065–1076. doi:10.1214/aoms/1177704472. https://projecteuclid.org/euclid.aoms/1177704472