Studying the Use of SZZ with Non-functional bugs

Sophia Quach

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Applied Science (Software Engineering) at

Concordia University

Montréal, Québec, Canada

May 2021

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Sophia Quach**

Entitled: **Studying the Use of SZZ with Non-functional bugs**

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Software Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

______________________ Chair
Dr. Name of the Chair

______________________ Examiner
Dr. Dr. Jinqiu Yang

______________________ Examiner
Dr. Dr. Emad Shihab

______________________ Supervisor
Dr. Weiyi Shang

Approved by ______________________
Lata Narayanan, Chair
Department of Computer Science and Software Engineering

____

May 6, 2021

____
Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Studying the Use of SZZ with Non-functional bugs

Sophia Quach

Non-functional bugs bear a heavy cost on both software developers and end-users. Tools to reduce the occurrence, impact, and repair time of non-functional bugs can therefore provide key assistance for software developers racing to fix these issues. Classification models that focus on identifying defect-prone commits, referred to as *Just-In-Time (JIT) Quality Assurance* are known to be useful in allowing developers to review risky commits. JIT models, however, leverage the SZZ approach to identify whether or not a past change is bug-inducing. However, the due to the nature of non-functional bugs, their fixes may be scattered and separate from their bug-inducing locations in the source code. Yet, prior studies that leverage or evaluate the SZZ approach do not consider non-functional bugs, leading to potential bias on the results.

In this thesis, we conduct an empirical study on the results of the SZZ approach on the non-functional bugs in the NFBugs dataset, and the performance bugs in Cassandra, and Hadoop. We manually examine whether each identified bug-inducing change is indeed the correct bug-inducing change. Our manual study shows that a large portion of non-functional bugs cannot be properly identified by the SZZ approach. We uncover root causes for false detection that have not been previously found. We evaluate the identified bug-inducing changes based on criteria from prior research. Our results may be used to assist in future research on non-functional bugs, and highlight the need to complement SZZ to accommodate the unique characteristics of non-functional bugs. Furthermore, we conduct an empirical study to evaluate model performance for JIT models by using them to identify bug-inducing code commits for performance related bugs. Our findings show that JIT defect prediction classifies non-performance bug-inducing commits better than performance bug-inducing commits. However, we find that manually correcting errors in the training data only slightly improves the models. In the absence of a large number of correctly labelled performance bug-inducing commits, our findings show that combining all available training data yields the best classification results.

Acknowledgments

First and foremost, I am profoundly grateful to my supervisor, Dr. Weiyi Shang, for his guidance, encouragement, and contributions during my research journey. My research would have been impossible to complete without his help and support, and I feel extremely lucky to have an intelligent and friendly mentor who guides me in exploring innovative ideas and achieving research goals.

I would also like to show my sincere gratitude to my committee members, Dr. Jinqiu Yang, and Dr. Emad Shihab, for taking their precious time to consider my work and offer insightful comments.

I would like to send my appreciation to Dr. Weiyi Shang, Dr. Maxime Lamothe, Dr. Yasutaka Kamei, and Dr. Bram Adams, from whom I've received guidance, valuable knowledge, as well as perseverance towards research, which will benefit both my academic and non-academic life.

Last but not least, I would like thank my fellow lab members from the SENSE lab and our neighbouring SPEAR lab, for the support and encouragement, as well as for the wonderful moments spent together.

Related Publications

The following is a list of our publications that are on the topic SZZ, non-functional Bugs, performance bugs, and JIT models:

- Sophia Quach, Maxime Lamothe, Yasutaka Kamei, and Weiyi Shang. An Empirical Study on the Use of SZZ for Identifying Inducing Changes of Non-functional Bugs. This work was accepted for publication under Empirical Software Engineering 2021, and is described in Chapter 3.

- Sophia Quach, Maxime Lamothe, Bram Adams, Yasutaka Kamei, and Weiyi Shang. Evaluating the impact of falsely detected performance bug-inducing changes in JIT models. This work is currently under major revision at Empirical Software Engineering, and is described in Chapter 4.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the first software bug was discovered in 1945, software engineers have been developing techniques to attempt to fix and prevent them Gyimothy, Ferenc, and Siket (2005); Hassan (2009); Kamei et al. (2013). Bugs are costly to fix, and increase maintenance effort LaToza, Venolia, and DeLine (2006). In a world that is ever more reliant on software Grubb and Takang (2003), it appears more important than ever before to have quality software and to be able to fix bugs in a timely manner when they do appear. To this end, researchers have developed several approaches to identify prior bug-inducing changes to help development teams avoid future bugs by learning from their mistakes Gyimothy et al. (2005); Hassan (2009); Kamei et al. (2013). Defect prediction models are a well-known technique used by practitioners to identify defect-prone files and packages during quality assurance. Once the defect-prone files or packages have been identified, developers still need to spend time modifying and examining code. This creates time-consuming and impractical tasks, especially for large software systems Kamei et al. (2013).

Just-In-Time Quality Assurance (JIT) provides an effort-reducing way to focus on the bug-inducing changes and thus reduce the costs of developing high-quality software. JIT quality assurance relies on the SZZ approach, and must therefore grapple with the drawbacks of SZZ. The SZZ technique helps with bug localization, and attempts to find the source-code changes that first induces a software bug S. Kim, Zimmermann, Pan, and Whitehead (2006); Sliwerski, Zimmermann, and Zeller (2005). However, the SZZ approach, like other bug localization techniques M. Kim and Lee (2018), is not perfect. Previous studies have shown that the SZZ approach can mislabel some changes as bug-inducing S. Kim et al. (2006). These mislabels include semantically equivalent changes, directory or file renames, and initial code importing changes da Costa et al. (2017); Fan et al. (2019); Neto, Costa, and Kulesza (2018). An evaluation framework exists to evaluate the various implementations of the SZZ approach that attempt to remedy these issues da Costa et al. (2017). However, the SZZ evaluation framework and the existing SZZ approaches concentrate on mixed bugs or functional bugs, without verifying the validity of the approach on non-functional bugs. Indeed, the Just-In-Time training data is based off of results provided by SZZ, which may include falsely identified past bug-inducing changes. These are then used to predict future inducing changes, giving flawed predictions.

Non-functional requirements are a class of software constraints relating to the quality of a program, different than functional requirements Radu and Nadi (2019). Functional requirements relate to the functionality of a program Kotonya and Sommerville (1998). Non-functional requirements impact aspects of a program such as efficiency, portability, and maintainability, and therefore are a key role in software development Radu and Nadi (2019). A non-functional bug as a problem in a piece of source code impairing some aspect of a program's non-functional requirements Radu and Nadi (2019). Functional changes and their software fixes are mainly localized, while non-functional bugs may be scattered and require fixes in various parts of the software Hamill and Goseva-Popstojanova (2014). For example, if a code change introduces a security vulnerability, security measures to counteract this may be implemented elsewhere Liu Ping, Su Jin, and Yang Xinfeng (2011); Mahrous and Malhotra (2018); Williams, McGraw, and Migues (2018). Developers often spend more time fixing performance bugs than fixing non-performance bugs Zaman, Adams, and Hassan (2011). Due to the unique nature of performance bugs, we suspect it is possible for performance bugs to present differently in source code. Zaman et al. find that more developers are assigned to fix performance bugs than functional bugs Zaman et al. (2011). If a code change introduces a performance issue, this performance issue may be fixed and improved in a different part of the system Jin, Song, Shi, Scherpelz, and Lu (2012); Nistor, Jiang, and Tan (2013), for example by changing configuration parameters. Non-functional bugs can be harder to fix than their functional counterparts. The SZZ approach would seemingly be useful in helping developers locate where to fix a performance bug in the source code. Due to the differing nature of non-functional bugs and functional bugs Glinz (2007); Radu and Nadi (2019), it is possible for non-functional bugs to present differently in source code, and therefore have different tooling requirements. Functional changes and their software fixes are mainly localized, while non-functional bugs, such as performance bugs may be scattered and require fixes in various parts of the software Hamill and Goseva-Popstojanova (2014). Hamill and Goseva-Popstojanova Hamill and Goseva-Popstojanova (2014) found that a significant number of software failures required fixes in multiple software components and/or multiple software artifacts (i.e., 15% and 26%, respectively). Due to the scattered nature of non-functional bugs, we suspect that the SZZ approach might perform worse on non-functional bugs than mixed bugs, namely functional bugs which can have a single concrete inducing commit that can be tracked down through the SZZ approach.

Additionally, if performance and non-performance bug-inducing commits have different characteristics, a model only based on performance bugs can theoretically label performance bug-inducing changes better. Because there is no verified ground truth for performance bug-inducing commits, we further manually verify the commits identified through the SZZ approach, through two reviewers. We then evaluate Just-In-Time models solely on the manually verified performance commits identified by the reviewers, by using four different combinations of training data. Of the model combinations, we include one where the training data is only comprised of manually labelled bug-inducing commits, to see whether we need a separate model for predicting performance bugs. We seek to determine how different training data influences the models' power to predict performance bug-inducing commits.

In the first part of the thesis, we seek to determine the usefulness of the SZZ approach in finding the cause of non-functional bugs. This thesis provides an evaluation of the SZZ approach with respect to non-functional bugs, while previous studies Kamei et al. (2013); S. Kim and Whitehead (2006); S. Kim et al. (2006); Pan, Kim, and Whitehead (2009) are evaluated on mixed bugs (both functional and non-functional) without distinction. We leverage the NFBugs dataset as a source of identified non-functional bugs Radu and Nadi (2019). The NFBugs dataset identifies bugs that specifically affect non-functional requirements, as well as the root cause of these bugs, for 65 open-source projects Radu and Nadi (2019). This dataset presents a vetted source of bugs and their root causes with which to test the SZZ approach. We therefore use this dataset to test the effectiveness

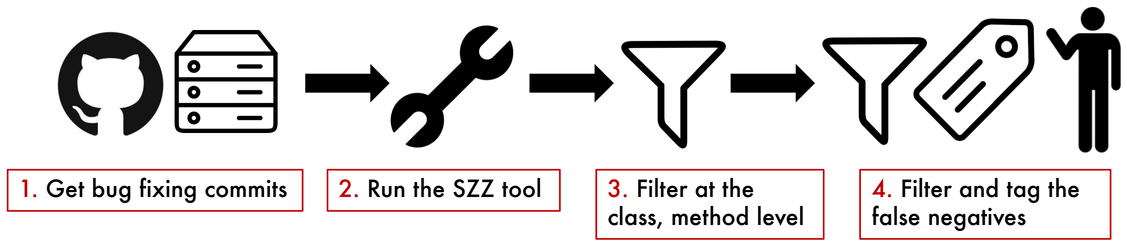

1. Get bug fixing commits
2. Run the SZZ tool
3. Filter at the class, method level
4. Filter and tag the false negatives

Figure 1.1: Procedure followed to evaluate bug-inducing commits identified by the SZZ approach

of the SZZ approach on non-functional bugs. To determine the usefulness of the SZZ approach with respect to non-functional bugs, we evaluate the SZZ approach based on three criteria: (1) the ability of the SZZ approach to identify bug-inducing changes for non-functional bugs; (2) the differences of inducing changes for non-functional bugs when compared to functional bugs; and (3) the characteristics of bug-inducing commits falsely identified by an SZZ approach as bug-inducing for non-functional bugs. We also aim to know whether or not performance bugs are well predicted by JIT defect prediction. We conduct an empirical study on the results of the SZZ approach used for JIT defect prediction, concentrating on the use of JIT defect prediction to identify the inducing changes of performance related bugs in Cassandra and Hadoop. For the purpose of our thesis, we perform our evaluation of the SZZ approach on a MA-SZZ implementation. We validate whether the bug-inducing changes found by MA-SZZ are truly bug-inducing changes by manually examining these identified changes in order to generate clean datasets. We conduct manual analysis to verify cross referenced fix commits and JIRA issue reports and we evaluate the performance related data for JIT models. Since manual analysis is time-consuming, we want to determine whether verified data makes a significant difference in training a better model to classify performance bug-inducing commits.

Upon evaluation of the SZZ approach, we filter out commits wrongly identified as bug-inducting commits by the SZZ approach (false-positives). Furthermore, we conduct a manual verification of the results of SZZ on our dataset to determine whether bug-inducing commits identified by the SZZ approach do in fact cause their related non-functional bugs. We use 132 bug fixes from the NFBugs dataset as our benchmark. For each fixing change, we run the SZZ approach to find how many bug-inducing changes are identified. In total, for all 132 bug fixes, we find that there are a total of 376 candidate bug-inducing commits. We manually observe the commits identified by the SZZ approach to determine whether they are truly bug-inducing. We use the additional insights gained through our manual verification of the SZZ approach to create an extension of the NFBugs dataset. Our findings show that among the 376 identified bug-inducing commits by the SZZ approach, only 79 of them are true positive bug-inducing commits. We manually break down the falsely identified bug-inducing commits into three reasons: multi-purposes bug-fixing commits, 2) bug already being there, and 3) not related to the bug. Our findings show that non-functional bug-inducing commits differ from functional bugs inducing commits, where guidelines to identify falsely detected bug-inducing commits in functional bugs cannot be used reliably when using SZZ to detect non-functional bugs.

In the second part of the thesis, we study the impact from falsely identified performance issues inducing changes from SZZ on the results of Just-in-Time defect prediction. We train a model with data from the SZZ approach and then test it on manually verified performance bug-inducing commits. Our findings show that JIT models perform better on classifying non-performance commits than performance commits, with AUC values of 0.842 to 0.869 for non-performance commits, and only 0.486 to 0.518 for performance commits. Furthermore, we find that manually vetting the results of the SZZ approach to produce JIT models generally does not impact the models' classification power, and when it does, the changes only have a small effect size (Cohen's d = 0.437) on the classification power of these models. Finally, we find that while ideally a large number of correctly labelled performance bug-inducing commits would be available, in the absence of this, it is still preferable to use all commit data in the training data, i.e., truly performance bug-inducing commits, non-performance bug-inducing commits, and non-bug-inducing commits, in order to obtain the best classification results.

The following are the primary contributions of the thesis:

- To the best of our knowledge this is the first study to focus exclusively on the use of the SZZ approach to identify the inducing commits of non-functional bugs.
- We manually verify the validity of the SZZ approach on non-functional bugs, and determine potential problems with the approach in dealing with them.
- We augment the NFBugs dataset by including bug fix descriptions that contain the commits

where the true bug-inducing changes reside[1].

- To the best of our knowledge this is the first study to focus exclusively on performance bug-inducing changes in the context of Just-In-Time defect prediction models.
- We manually verify the validity of the SZZ approach on performance bug-inducing changes.
- We evaluate the results of the Just-In-Time model bug prediction before and after our modifications after manually checking whether a commit that is labelled as bug-inducing is truly bug-inducing.[2]

Chapter organization. Chapter 2 introduces the SZZ approach and other background concepts. Chapter 3.1 presents our subject dataset, applications of the SZZ approach, and the steps of our study. Chapter 3.2 presents our approach to determine the bug-inducing commits for non-functional bugs. Chapter 3.3 presents our empirical study of the SZZ approach when used on non-functional bugs. Chapter 3.2 presents a manual investigation of the results of our empirical study, as well as discussion of these results. Chapter 2 also discusses prior work related to the work presented in this thesis. Chapter 3.5 presents the threats to the validity of our work. Chapter 2 introduces background concepts including the SZZ approach and Just-In-Time defect prediction. Chapter 4.1 describes the design of our study. Chapter 4.2 presents our case study results addressing our research questions. Chapter 4.3 presents the threats to the validity of our work. Chapter 2 presents prior work related to the work presented in this thesis. Finally, Chapter 5 concludes the thesis.

[1]Our extension of the NFBugs dataset is publicly available and can be found at: https://github.com/senseconcordia/NFBugsExtended

[2]Our data files and scripts used are publicly available and can be found at: https://github.com/senseconcordia/Perf-JIT-Models

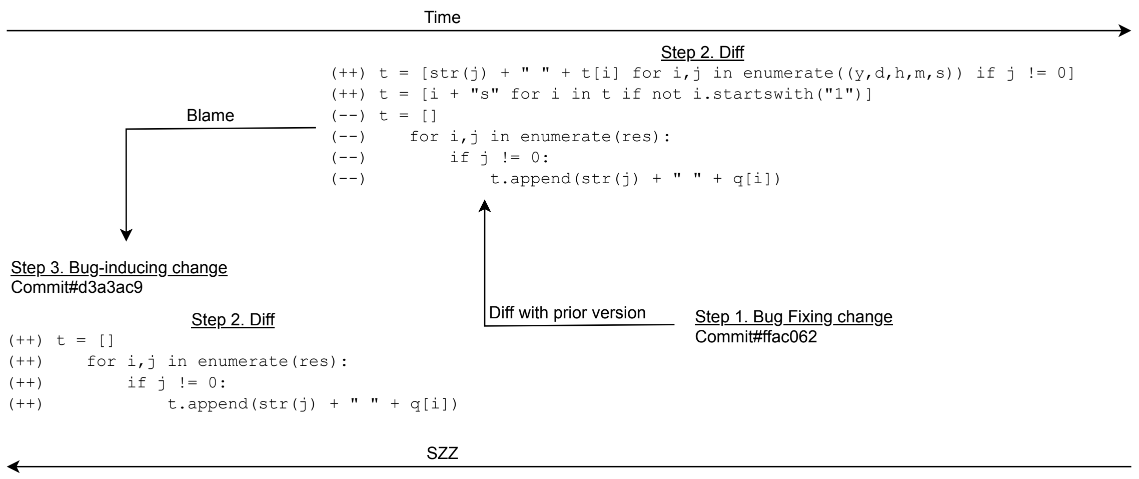
Chapter 2

Related Work

2.1 Overview of the SZZ approach

The SZZ approach is used to identify the changes that introduce bugs. The approach starts from a bug-fixing change, i.e., a change that is known to have fixed a bug da Costa et al. (2017). For each identified bug-fixing change, the SZZ approach analyzes the lines of code that were updated to introduce the fix. In order to identify the change that originally introduced the bug, the SZZ approach traces through the history of the source code management system da Costa et al. (2017). The *git annotate* function, now replaced by *git blame* Borg, Svensson, Berg, and Hansson (2019) that is provided by most SCM systems is used by the approach to identify the last time a given line of code was changed before the bug-fixing commit da Costa et al. (2017). Figure 2.1 shows a bug fix and a corresponding identified bug-inducing change from the SZZ approach.

Because no SZZ approach provides perfect precision and recall of bug-inducing commits, it is necessary to understand the nature of the results given by the SZZ approach. In an ideal scenario, the SZZ approach can identify the exact changes that introduce a bug. In this ideal case, we consider the results to be changes identified by the SZZ approach that are truly bug-inducing changes, i.e., true positives. These results can directly be used by developers to find the root cause of a known bug. However, if the SZZ approach identifies changes that are not truly bug-inducing changes as bug-inducing, we consider those to be false positives. False positives can cause developers to needlessly look at faultless code. False negatives are truly bug-inducing changes that were missed by the SZZ



```
(++) t = [str(j) + " " + t[i] for i,j in enumerate((y,d,h,m,s)) if j != 0]
(++) t = [i + "s" for i in t if not i.startswith("1")]
(--) t = []
(--)    for i,j in enumerate(res):
(--)        if j != 0:
(--)            t.append(str(j) + " " + q[i])
```

```
(++) t = []
(++)    for i,j in enumerate(res):
(++)        if j != 0:
(++)            t.append(str(j) + " " + q[i])
```

Figure 2.1: Overview of the SZZ approach. The SZZ approach first looks at the changes made in a bug-fixing change (Step 1). It then uses *git diff* to localize the exact fix (Step 2). Finally, the deletions are traced back to the origin of the deleted code (Step 3). The origin of the deleted code is a potential bug-inducing change.

approach. False negatives would require different tools or manual investigation to find the root cause of a bug. Various modifications of the SZZ approach attempt to improve its true positive rate and reduce its false positives and false negative rates.

2.2 Applications of the SZZ approach

Kamei et al. Kamei et al. (2013) study defect prediction models that focus on identifying defect-prone software change level, rather than file or package level, referred to as "Just-In-Time Quality Assurance", where developers can review and test these risky changes while they are still fresh in their minds. In this case, the SZZ approach is used to link each defect fix to the source code change introducing the original defect by combining information from the version archive with the bug tracking system. Findings from Kamei et al. Kamei et al. (2013) indicate that "Just-In-Time Quality Assurance" may provide an efficient way to focus on the most risky changes and thus reduce the costs of developing high-quality software.

Current adoption of techniques that predict software quality remains low. One of the reasons for the low adoption rate of current analytics and prediction techniques is the lack of actionable and publicly available tools. Rosen et al. Rosen, Grawi, and Shihab (2015) present Commit Guru Rosen et al. (2015), a publicly available, language agnostic, analytics and prediction tool that identifies and predicts risky software commits mined from any Git repository. Additionally, Commit Guru Rosen et al. (2015) automatically identifies risky (i.e., bug-inducing) changes and builds a prediction model to assess the likelihood of a recent commit being bug-inducing in the future. A similar approach to the SZZ approach is used to determine bug-fixing commits Rosen et al. (2015).

The wide application of the SZZ approach motivates our research of the usefulness of the SZZ approach when used to identify the inducing changes on non-functional bugs.

In this chapter, we situate our work within the context of past studies that have evaluated SZZ approaches and studying non-functional bugs.

2.3 Implementations of SZZ

The first SZZ approach B-SZZ (the basic SZZ implementation) was defined by Sliwerski et al. Sliwerski et al. (2005), to identify the changes that introduce bugs. SZZ requires a code change that fixes a bug found in the code as a base input, also known as bug-fixing changes. Figure 2.1 shows an example of the SZZ approach, with a bug fix and a corresponding identified bug-inducing change. Since then, due to B-SZZ's limitations, there have been several improvements proposed: including the AG-SZZ implementation by Kim et al S. Kim et al. (2006). Costa et al. da Costa et al. (2017) later proposed the MA-SZZ implementation, which is built on top of AG-SZZ. MA-SZZ improves upon AG-SZZ by removing potential bug-introducing changes that are meta-changes. Meta-changes are source code independent changes, such as source code management branch changes, source code merges, and changes to file properties such as end-of-line changes da Costa et al. (2017).

Costa et al. da Costa et al. (2017) find that B-SZZ has the lowest disagreement ratio in general (0%-9%), followed by the MA-SZZ (0%-17%) da Costa et al. (2017). The bugs analyzed by MA-SZZ have the shortest time-span of bug-introducing changes, while B-SZZ has the longest time-span of bug-introducing changes [7]. Costa et al. [7] also report that MA-SZZ returns the second highest count of future bugs. We choose the MA-SZZ implementation proposed by Costa et. al da Costa et al. (2017), as it is similar to B-SZZ with some improvements, such as excluding style changes and including usage of an annotation graph S. Kim et al. (2006), and the removal of meta-changes da Costa et al. (2017). MA-SZZ is also used in several studies including work on Just-In-Time defect prediction, to identify bug-inducing changes Kamei et al. (2013); McIntosh and Kamei (2018) as a ground truth for building the prediction models. Additionally, prior work in refactoring changes, such as RA-SZZ Neto et al. (2018), uses MA-SZZ by incorporating it with RefDiff to propose a refactoring aware SZZ implementation Neto et al. (2018). Although newer approaches introduce refactoring awareness, we do not know how those interact with software performance. We therefore err on the side of caution by using MA-SZZ, a more established and more commonly used implementation of the SZZ approach.

Borg et al. Borg et al. (2019) propose an open implementation of the SZZ approach for git repositories. The authors include a usage example for the Jenkins project and conclude with a case study on JIT bug prediction. The SZZ Unleashed implementation is based on Sliwerski et al.'s Sliwerski et al. (2005) work, as well as later enhancements by Williams and Spacco Williams and Spacco (2008). Because MA-SZZ is also based on Sliwerski et al.'s Sliwerski et al. (2005) work, our findings on non-functional bugs may benefit SZZ Unleashed Borg et al. (2019) as its purpose is for git repositories, which can contain a mixture of functional and non-functional bugs.

2.4 Evaluating SZZ approaches

Kim et al. present algorithms to identify bug-inducing changes automatically and accurately. They compare their algorithms to the SZZ S. Kim et al. (2006) approach. They removed false positives and false negatives by using annotation graphs, and ignored non-semantic code changes and outlier fixes. They also manually inspected the commits listed as bug fixing to determine if they were indeed changes that fixed a bug in the code. In Chapter 3, we evaluate our dataset with MA-SZZ, while Kim et al. S. Kim et al. (2006) evaluated the approach on the first version of the SZZ approach. In Chapter 4, we evaluate our Cassandra and Hadoop datasets with the bug inducing changes found by SZZ. Furthermore, we concentrate on the SZZ approach S. Kim et al. (2006), as it is used in JIT defect prediction.

Williams et al. revisit the SZZ approach by outlining several improvements to the approach Williams and Spacco (2008). They replace annotation graphs with linear number maps to track unique source code lines as they change over software evolution. Their enhanced approach uses weights to map the evolution of a line. They also use DiffJ, a Java syntax-aware diff tool to ignore comments and ignore cosmetic changes Jpace (n.d.). Furthermore, they verify how often bug-inducing changes identified by the SZZ approach are truly bug-inducing changes. We want to compare an improved SZZ approach implementation: MA-SZZ, as the study performed by Williams et al. compared their improvements to the first SZZ approach implementationSliwerski et al. (2005). In Chapter 4, we want to verify whether the SZZ approach can provide true bug inducing changes on Cassandra and Hadoop performance bugs and the impact on JIT models.

Costa et al. da Costa et al. (2017) introduced a framework to evaluate the results of SZZ approach implementations. They note that little effort has been made to evaluate SZZ's results, despite its role as the foundation of several research areas in software engineering da Costa et al. (2017). The framework evaluates the approach with three criteria: the earliest bug appearance, the future impact of changes, and the realism of bug introduction da Costa et al. (2017). The framework is evaluated on five SZZ implementations using data from ten open source projects. Their findings show that previous proposed improvements to SZZ approaches tend to inflate the number of false positive bug-inducing changes. A single bug-inducing change may be blamed for introducing hundreds of future bugs and SZZ implementations report that at least 46% of the bugs are caused by bug-inducing changes that are years apart from one another da Costa et al. (2017). Our study builds on the work from Costa et al. by using their evaluation criteria as well as new evaluation criteria to evaluate SZZ approaches on non-functional bugs rather than on a mixed dataset containing both functional and non-functional bugs. Similarly to Costa et al. da Costa et al. (2017), in Figure 3.5 we evaluate our data on the earliest bug appearance, in Figure 3.4 we evaluate our data on the future impact of changes. In Chapter 3, we build on the work from Costa et al. by evaluating the identified bug inducing changes from SZZ on non-functional bugs rather than on a mixed dataset containing both functional and non-functional bugs.

Fan et al. Fan et al. (2019) studied the impact of mislabelled changes of the SZZ approach on JIT prediction. They analyze four different SZZ implementations and build the JIT prediction models using the labeled data of these four variants Fan et al. (2019). For MA-SZZ, Fan et al. Fan et al. (2019) find that the labeled data has low false positive and false negative rates, compared to AG-SZZ, which contains a much larger number of false negatives. The low false positive and false negative rates may not be likely to impact the prediction of the MA models Fan et al. (2019). Checking the impact of performance bugs on the NFBugs data is an avenue for future work to advance studies pertaining non-functional bugs.

2.5 Just-In-Time defect prediction

Defect prediction models are a well-known technique for identifying defect-prone files or packages for practitioners to allocate quality assurance efforts. One underlying problem is that once the critical files or packages have been identified, developers still need to spend considerable time examining and modifying source code, which is time consuming and impractical for large software systems.

Kamei et al. Kamei et al. (2013) study prediction models that focus on identifying defect-prone software at the change level, rather than file or package level, referred to as *Just-In-Time Quality Assurance*, where developers can review and test these bug-inducing changes while they are still fresh in their minds. Kamei et al. Kamei et al. (2013) use a wide range of factors based on the characteristics of a software change, such as the number of added lines, and developer experience. They perform a large scale study on JIT models to see if they can predict whether or not a change will lead to a defect Kamei et al. (2013). To know whether or not a change introduces a defect, Kamei et al. Kamei et al. (2013) use the SZZ approach, linking each defect fix to the source code change introducing the original defect by combining information from the version archive with the bug tracking system. They find that using only 20 percent of the effort it would take to inspect all changes, they can identify 35 percent of all defect-inducing changes. Their findings indicate that JIT may provide an effort-reducing way to focus on the most bug-inducing changes and thus reduce the costs of developing high-quality software Kamei et al. (2013). We build on this work by evaluating the JIT models specifically on performance related bug-inducing changes rather than all bug-inducing changes.

JIT models identify bug-inducing code changes and are trained using techniques that assume past bug inducing changes are similar to future ones. However, this assumption may not hold, e.g., as system complexity tends to accrue, expertise may become more important as systems age. McIntosh et al. McIntosh and Kamei (2018) study JIT models as systems evolve. Through a longitudinal case study of open source systems, they find that fluctuations in the properties of fix-inducing changes can impact the performance and interpretation of JIT models. They find that the discriminatory power (AUC) and calibration (Brier) scores of JIT models drop considerably one year after being trained. Our work focuses on studying JIT performance related bug-inducing changes and how they can impact the performance of JIT models. Our thesis also focuses on the data itself that is produced by the SZZ approach, i.e., the bug-inducing changes detected by the SZZ approach.

2.6 Performance Bugs

Jin et al Jin et al. (2012) define a performance bug as a bug that causes a perceivable negative performance impact. For clarification between the distinction of performance and non-performance bugs, we provide example code snippets of a performance bug and a non-performance bug, in Figure 2.2 and Figure 2.3, respectively.

Performance bug in Figure 2.2: The commit message for the changes shown is: **Make repair coordination less expensive by moving MerkleTrees off heap Removal of for loop in hash() of MerkleTrees.java**. This code snippet shows the code changes in the bug-fixing commit **2117e2a**. The for loop in the *hash()* function of MerkleTrees.java class is now less expensive, by using a Javva entryset instead of a keyset in the modified for loop. The red lines were initially added by the bug-inducing commit **0dd50a6**, and were responsible for the creation of the MerkleTrees.java class and the *hash()* method.

Non-performance bug in Figure 2.3: The commit message for the changes shown is: **fix callback when repair request times out**. This code snippet shows the code changes in the bug-fixing commit **74c464a**. The *get()* method callback now uses *getMessageCount()* instead of *isDataPresent()*. The red line was initially added in the bug-inducing commit **71ccb7d**.

To identify performance bugs, Ding et al. Ding, Chen, and Shang (2020) use keywords as the

```
- for (Range<Token> rt : merkleTrees.keySet())
- {
-   if (rt.intersects(range))
-   {
-       byte[] bytes = merkleTrees.get(rt).hash(range);
-       if (bytes != null)
-       {
-           baos.write(bytes);
-           hashed = true
-       }
-   }
- }
+ for (Map.Entry<Range<Token>, MerkleTree> entry : merkleTrees.entrySet())
+   if (entry.getKey().intersects(range))
+       hashed |= entry.getValue().ifHashesRange(range, n -> baos.write(n.hash()));
+ return hashed ? baos.toByteArray() : null;
```

Figure 2.2: Simplified example of code changes in a fix commit for a performance bug.

```
- return resolver.isDataPresent() ? resolver.resolve() : null;
+ return resolver.getMessageCount() > 0 ? resolver.resolve() : null;
```

Figure 2.3: Simplified example of code changes in a fix commit for a non-performance bug.

heuristics to identify performance issue reports. Ding et al. Ding et al. (2020) start by using the keywords that are used in prior research Jin et al. (2012); Zaman, Adams, and Hassan (2012). In order to avoid missing performance issues, the list of keywords was expanded by using word embedding. Ding et al. Ding et al. (2020) adopt a word2vec model trained over textual data from Stack Overflow posts to identify the words that are semantically related to the existing list of keywords. Examples of the uncommon words that are related to performance issues include "sluggish", and "laggy", which may not be used in previous research, but can help collect performance issue reports. In order to ensure that there exists a performance improvement after the issue fixes, Ding et al. Ding et al. (2020) only focus on the issue reports that have the type Bug and are labeled as *Resolved* or *Fixed*. In total, Ding et al. Ding et al. (2020) find 121 performance-related issue reports in Cassandra and 83 in Hadoop. We use this vetted performance-related data in our thesis.

2.7 Predicting performance bugs

Software quality research and practice concerns itself with a variety of different types of bugs. Non-functional bugs, including performance and security bugs, can be particularly costly bugs Zaman et al. (2011). Tools can help reduce the cost overhead caused by these bugs Jin et al. (2012). In Chapter 3, we focus on the applicability of the SZZ approach to determine the root cause of these non-functional bugs.

Jin et al. Jin et al. (2012) have studied 109 real-world performance bugs from five software systems in order to better guide software practitioners. Their findings show that developers need tool support to automatically fix such types of performance issues Jin et al. (2012). They also find that performance issues of newer software versions can be inherited easily from previous versions. This study calls for further and more detailed research on performance diagnosis, performance testing, and performance issue detection. Our study contributes to furthering software performance research by evaluating a tool to help developers automatically locate buggy code in the software. In theory, the SZZ approach should be able to locate at which point in time non-functional bugs, including performance issues, are introduced from previous versions, given that the performance issue has been detected.

Zaman et al. Zaman et al. (2011) conduct both qualitative and quantitative studies on 400 performance and non-performance issues in Mozilla Firefox and Google Chrome, two open source web browsers. The study aims to understand the difference between performance issues and non-performance issues. Their findings show that developers spend more time fixing performance issues rather than non-performance issues Zaman et al. (2011). The study shows the importance of identifying root causes for performance issues and evaluating the impact of changes on performance issues. Our thesis analyzes the performance bugs in Cassandra and Hadoop, and the SZZ approach's ability to determine the bug inducing changes and concentrate on the impact of these changes on predictive models.

Nistor et al. Nistor et al. (2013) studied software performance since performance is critical for how users perceive the quality of software products. Performance bugs lead to poor user experience and low system throughput Bryant and O'Hallaron (2015); Molyneaux (2009). Their study includes how performance bugs are discovered, fixed, and compares the results with those for non-performance bugs from three popular code bases: Eclipse JDT, Eclipse SWT and Mozilla Firefox Guindon (n.d.); team (n.d.). Their results include suggestions of techniques to help developers reason about performance and suggest that better profiling techniques are needed for discovering performance bugs. Our study on the evaluation of a SZZ approach on performance bugs, can help determine whether it is reliable for developers to use references to past inducing code from past performance bugs to locate and fix new bugs with the help of a SZZ-implemented tool.

Intrinsic bugs are bugs where a bug-introducing change can be identified using the version control system of a software, while extrinsic bugs are caused by external changes of a software such as errors in internal APIs, compatibility changes or even changes in the specifications Rodriguezperez, Nagappan, and Robles (2020). Extrinsic bug introducing-changes cannot be ident ified by the version control system of the software. Rodriguez-Perez et al. Rodriguezperez et al. (2020) study the impact of extrinsic bugs in JIT models, by replicating Kamei et al.'s recent thesis in which they analyze the performance of JIT models Kamei et al. (2013). Rodriguez-Perez et al. Rodriguezperez et al. (2020) remove the extrinsic bugs from their data, as all bugs were previously considered to be intrinsic. Findings from Rodriguez-Perez et al. Rodriguezperez et al. (2020) show that extrinsic bugs are of a different nature than intrinsic bugs, and that extrinsic bugs are more similar to issues that are not bugs rather than to intrinsic bugs. Our work also focuses on studying JIT models, but we concentrate on performance related bug-inducing changes and how they can impact the performance of JIT models. We investigate whether or not performance bugs in our dataset are extrinsic bugs.

Tsakiltsidis et al. Tsakiltsidis, Miranskyy, and Mazzawi (2016) use four machine learning algorithms to build classifiers to predict performance bugs on a real time system used in the mobile advertisement. Their findings show that the best model is based on logistic regression, using lines of code changed, file age and size as explanatory variables to predict performance bugs. They also find that reducing the number of changes delivered on a commit can decrease the chance of performance bug injection. While Tsakiltsidis et al. Tsakiltsidis et al. (2016) have a special goal to predict performance bugs, our study focuses on the prediction of performance bug-inducing chances within a general JIT model.

Yang et al. use 14 code-change based metrics to build simple unsupervised and supervised models in effort-aware JIT defect prediction Yang et al. (2016). They find that many simple unsupervised models perform better than the state-of-the-art supervised models in effort-aware JIT defect prediction. Their study used the data sets provided online by Kamei et al. Kamei et al. (2013), which employ the SZZ approach. Our study investigates the predictive power of supervised models in effort-aware JIT defect prediction Kamei et al. (2013) as well, however our labelling of defect-prone commits is done by employing the MA-SZZ approach. We also manually correct the labelling of defect-prone commits, found by the MA-SZZ approach, and compare the models.

Chen et al. Chen, Shang, and Shihab (2020) propose an approach that automatically predicts whether a test would manifest performance regressions given a code commit, using both traditional metrics and performance-related metrics from the commit changes that are associated with each test. For each commit, they build random forest classifiers that are trained from all prior commits to predict in this commit whether each test would manifest performance regression Chen et al. (2020). We also use traditional metrics from the commit changes to build classifiers trained from prior commits to predict if a commit is bug-inducing or not, focusing on performance bugs.

Software quality research and practice concerns itself with a variety of different types of bugs. Non-functional bugs, including those of performance and security can be particularly costly bugs Zaman et al. (2011). Tools can help reduce the cost overhead caused by these bugs Jin et al. (2012). In this study we concentrate on the applicability of the SZZ approach to determine the root cause of these non-functional bugs.

The absence of bug-inducing knowledge in issue trackers forces researchers to rely on alternative sources of information, such as the SZZ approach, which can be used as a heuristic approach to identify bug-inducing changes Borg et al. (2019). In a recent systematic literature review, it was determined that few researchers have made their SZZ implementations publicly available, causing extra research effort to be spent, as new projects based on SZZ output need to initially re-implement the approach Borg et al. (2019). The repeated re-implementation of SZZ also raises the risk that newly developed SZZ implementations have not been properly tested Borg et al. (2019). Borg et al. Borg et al. (2019) present SZZ Unleashed, an open implementation of the SZZ approach for Git repositories. Our thesis uses the same implementation of SZZ as McIntosh et al. McIntosh and

Kamei (2018).

Automatic identification of the differences between two versions of a file is a common and basic task in several applications of mining code repositories, commonly by using the git diff command Nugroho, Hata, and Matsumoto (2019). Nugroho et al. Nugroho et al. (2019) empirically analyze the impact of diff algorithms in three major applications: code churn metrics of the SZZ approach, and patches extraction. The results of locating bug-inducing changes using the SZZ approaches relies on the diff results. Nugroho et al. Nugroho et al. (2019) find that 25% of purposes of using the git diff command is for identifying bug-inducing change identification in the SZZ approach. Meanwhile, the work presented in this thesis evaluates the detected-bug-inducing changes of the SZZ approach.

Their results include suggestions of techniques to help developers reason about performance and suggest that better profiling techniques are needed for discovering performance bugs. Our study on the evaluation of an SZZ approach on non-functional bugs, can help determine whether it is reliable for developers to use references to past inducing code from past performance bugs to locate and fix new bugs with the help of an SZZ-implemented tool.

Chapter 3

An Empirical Study on the Use of SZZ for Identifying Inducing Changes of Non-functional Bugs

In this section, we present the design of our exploratory study. We first present the dataset as the subject of our study. Afterwards, we layout the two steps of our study and their motivations.

3.1 Study Design

3.1.1 Subject dataset

Because SZZ requires known bugs as inputs, our research hinges on the availability of vetted non-functional bugs. Datasets of non-functional bugs have been produced and vetted in prior research Ohira et al. (2015); Radu and Nadi (2019). However, non-functional bugs and their causes, sometimes require domain knowledge to be understood and detected. It is therefore beneficial for the root-causes of each non-functional bug to be clearly indicated as provided by the NFBugs dataset Radu and Nadi (2019). Such information is crucial in our study to verify the bug-inducing changes identified by the SZZ approach.

Fortunately, recent research by Radu and Nadi Radu and Nadi (2019) initiated an open repository that contains a dataset of real-world non-functional bugs, with each bug's detailed information (see Listing 3.1). The NFBugs dataset contains bugs from 65 open source GitHub projects: 40 Java projects and 25 Python projects. These projects contain 89 listed Java non-functional bugs and 43 listed Python non-functional bugs. For each project, NFBugs lists at least one bug, its respective fix and its detailed root causes. Each listed bug has been manually identified and has a corresponding YAML file, with the file and method that pinpoints the bug as well as a short description of the bug, however it contains no mention of commits that induced the bug. An example of a YAML file is shown in Listing 3.1. This study makes use of the NFBugs dataset to advance the state-of-the-art and allow future replication.

```
source:
    name: github-search
project:
    name: VS_test
    url: https://github,com/georgeriz/VS_test/
fix:
    tag: performance
    description: Replacing .append loop with list comp improves performance because it
        does not have to access the instance method
    commit message: > project duration:list comprehension instead of for loop
    commit: https://github,com/georgeriz/VS_test/commit/ffac062
location:
    file: duration/duration.py
    method: duration(seconds)
api: builtins.list
api change:                                     builtins.list.append -> builtins.
    list list comprehension
rule: use list comprehension instead of list.append loops to create lists efficiently
```

Listing 3.1: An example YAML file from the NFBugs dataset for a non-functional bug **ffac062** in VS_test

3.1.2 Implementation of SZZ

Since its creation, SZZ has been modified and re-implemented with various modifications da Costa et al. (2017); Davies, Roper, and Wood (2014); S. Kim et al. (2006); Sliwerski et al. (2005). In this thesis we concentrate on the MA-SZZ (meta-change aware SZZ) implementation of SZZ da Costa et al. (2017). Meta-changes are source code independent changes; source code management branch changes, source code merges, and changes to file properties such as end-of-line changes are all examples of meta-changes da Costa et al. (2017). We concentrate on the MA-SZZ implementation of SZZ because it is commonly used in prior research da Costa et al. (2017) and is language agnostic. The MA-SZZ da Costa et al. (2017) approach is an implementation of SZZ that adds meta-change awareness to the AG-SZZ approach S. Kim et al. (2006). MA-SZZ uses an annotation-graph to represent the evolution of each line of code within source files. Depth-first search is performed on the annotation-graph to find potential bug-inducing changes. We use our own implementation of the MA-SZZ approach to perform our evaluation of the SZZ approach on non-functional bugs. Because prior research has used MA-SZZ to observe functional bugs, we also use MA-SZZ to observe non-functional bugs. We did not want to introduce a different SZZ approach (e.g., RA-SZZ Neto et al. (2018)) because although RA-SZZ has shown improvements over other SZZ approaches Neto et al. (2018), changing the approach may introduce confounding factors and make the results impossible to compare fairly. However, while we suspect that refactorings actually can have an effect on non-functional bugs, as they do on functional bugs, we have not yet found a study that presents the effects of refactoring on non-functional software bug incidence detection. Therefore, to be conservative, we use an MA-SZZ implementation of the SZZ approach rather than an RA-SZZ implementation because using RA-SZZ would potentially change the performance of the method and make the results difficult or impossible to compare with prior studies.

The SZZ approach requires a bug-fixing change as an input. The approach then performs its depth-first search to find potential bug-inducing changes. Therefore, we require bug-fixing changes, specifically bug-fixing commits, to use as inputs for the SZZ approach. The NFBugs dataset lists the bug-fixing commit hash of each non-functional bug instance. Most of the closed bug reports identify the fix commit that closed the bug report. We therefore use these bug-fix commits as input data for our study.

3.1.3 Steps of our study

The steps of our study can be found in Figure 1.1. Using the non-functional bug dataset and the MA-SZZ implementation of SZZ, we first run the SZZ approach to identify bug-inducing commits. Then we carry out our study in two steps.

Step 1: Manually verifying the bug-inducing commits identified by SZZ

Prior studies have manually evaluated the results that are generated by the SZZ approach da Costa et al. (2017). However, those studies do not make a distinction between functional and non-functional bugs during their evaluation. Nonetheless, it has been shown that non-functional bugs present different characteristics than functional bugs Nistor et al. (2013). In particular, non-functional requirements describe the quality attributes of a program, as opposed to its functionality Kotonya and Sommerville (1998). Therefore, the prior manual evaluation results for SZZ approaches may not generalize to non-functional bugs.

In addition, in order to further improve the SZZ approach on non-functional bugs, it is necessary to first obtain a dataset of non-functional bugs with correctly identified bug-inducing commits. The correct bug-inducing commits are paramount for any further analysis. In this step, we therefore seek to manually verify the inducing changes for non-functional bugs and complement the existing dataset of these non-functional bugs by incorporating their corresponding true inducing changes.

Step 2: Automatically evaluating the bug-inducing commits identified by SZZ

Manually evaluating results from the SZZ approach is time consuming and almost impossible to scale in practice. However, practitioners may always face the challenge of having falsely identified bug-inducing changes from SZZ approaches. To address such a challenge, prior work by Costa et al. da Costa et al. (2017) provides characterizations of SZZ results using three characteristics of bug-inducing changes as guidelines: 1) *Earliest bug appearance*, 2) *Future impact of a change*, and 3) *Realism of bug introduction*. These bug characteristics can be used to provide a fine-grained evaluation of the SZZ approach. *Earliest bug appearance* measures when a bug was introduced. *Future impact of a change*, analyzes the number of future bugs that a given bug-inducing change introduces. Finally, *realism of bug introduction* analyzes whether the bug-inducing changes found by

SZZ approaches realistically correspond to the actual bug introduction context. If these automated guidelines cannot reliably identify the falsely identified bug-inducing commits for non-functional bugs, practitioners and researchers may not adopt these guidelines for evaluating the results of SZZ approaches on other datasets.

The results of **Step 1: Manually verifying the bug-inducing commits identified by SZZ** are used in Chapter 3.2. The results of **Step 2: Automatically evaluating the bug-inducing commits identified by SZZ** are used in Chapter 3.3.

3.2 Manually verifying bug-inducing commits for non-functional bugs identified by the SZZ approach

In this step, we manually verify the bug-inducing commits that are automatically identified by the SZZ approach.

Before applying the SZZ approach on our dataset, we first exclude nine bugs (five from Java and four from Python) where the bug-fix commits are part of merge commits. We exclude merge commits since studies shows that the SZZ approach should not take merge commits into account due to the noise that can be introduced by a code merge Fan et al. (2019).

Afterwards, we run the SZZ approach for the remaining 123 bugs. The SZZ approach produces a list of bug-inducing candidate commits for each bug. After running the SZZ approach on the remaining bugs, we obtained a total of 284 candidate bug-inducing commits for Java and 92 for Python. Each bug in the NFBugs database has been manually identified and has a corresponding YAML file listing the file(s) and method(s) for the bug as well as a short description of the bug, as shown in Listing 3.1. Each of these remaining candidate bug-inducing commits, are manually verified by three of the authors of this thesis independently, to avoid introducing any bias.

The steps performed in this thesis for the manual analysis of bug-inducing commits are as follows:

- **Step A**: The reviewers read the description of a non-functional bug from the NFBugs dataset.
- **Step B**: The reviewers examine the code from the mentioned bug-fix commit.

- **Step C**: For each of the candidates identified as bug-inducing commits by the SZZ approach, the reviewers examine the code that is changed in the commit and determine whether it induces the corresponding bug.

After all steps are individually completed by the first, third, and fourth authors for all bugs remaining in the dataset, the reviewers then meet to discuss disagreements. All three reviewers must have the same classification (i.e., bug-inducing or not bug-inducing) for a candidate commit, otherwise this is marked as a disagreement. The disagreements are resolved as follows:

- **Step D**: The reviewers re-read the bug-fix and the bug-inducing commit in question.
- **Step E**: Each reviewer states the reason why they think the identified commit is bug-inducing or not bug-inducing.
- **Step F**: The reviewers discuss until all three agree on a final decision.

All agreements and disagreements are recorded and used to calculate the Multi Kappa Fleiss score, a robust statistic useful for either interrater or intrarater reliability testing McHugh (2012). Afterwards, the three individuals meet and discuss any differences and reach a consensus.

Finally, we manually investigate all the false candidate of bug-inducing commits for non-functional bugs, in order to uncover reasons of such faults.

Results.

For the manual examination of the candidate bug-inducing commits, there were a total of 27 candidate bug-inducing commits disagreements, 20 from Java and 7 from Python bugs. To quantitatively evaluate how often we agreed during manual evaluation, we use the Multi Kappa Fleiss score McHugh (2012). The Multi Kappa Fleiss score was 0.728 - a moderate level of agreement for Java, and 0.815 - a strong level of agreement for Python McHugh (2012). All 27 candidate bug-inducing commit disagreements were resolved through discussion between the three individuals. In many cases, the disagreements split 1:2 were resolved due to one individual missing some critical information while manually reviewing the code. After a second look at the code, the reviewers' response allowed the three reviewers to reach consensus.

Table 3.1: Java and Python Projects: True-Positives and False-Positive after filtering based on relevant method and class based on the description provided

Language	No. Pairs of identified bug-inducing commits	TP	FP
Java	284	109	175
Python	92	50	42

Only 41 out of 123 bugs have fully correct bug-inducing changes identified by the SZZ approach. 27 bugs have fully wrong identified bug-inducing changes. **For the bugs where the SZZ approach was not able to identify any truly bug inducing commits, we manually look in the repository to find the commits that were bug-inducing for bugs: accounting for 27 out of 123 bugs.** 19 bugs have a combination of correct bug-inducing changes and incorrect bug-inducing changes, e.g., in the case of multi-purpose bug-fixing commits, some code changes in the commit are not done to fix the bug, tracking such changes may result in falsely identified bug-inducing commits.

Out of the 376 bug-inducing commits, 217 commits were ruled out as false positive bug-inducing commits which were not in the same method or class from the total 376 identified bug-inducing commits. For the remaining 159 bug-inducing commits, there are a total of 80 bug-inducing commits that are false positives and therefore were falsely labelled as bug-inducing commits by the SZZ approach shown in Figure 3.1. Based on our findings on the NFBugs dataset, there are 55 performance bugs that have 94 bug-inducing commits with 45 truly bug-inducing commits, and there are five security bugs that have five bug-inducing commits with four truly bug-inducing commits.

Prior studies have reported that the SZZ approach still needs improvements to accurately identify bug-inducing changes da Costa et al. (2017). For MA-SZZ, Costa et al. da Costa et al. (2017) report a 0% to 17% disagreement ratio, where they count a bug as a disagreement if all of the candidate bug-inducing changes identified by the SZZ approach for that bug are classified as incorrect. However, based on our manual analysis results, the SZZ approach performs even worse on non-functional bugs.

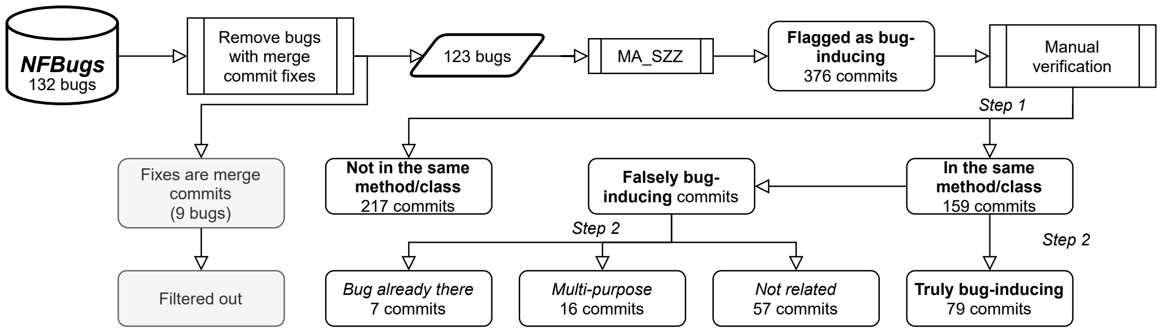


Figure 3.1: Breakdown of false positive bug-inducing commits which were not in the same method-/class mentioned from the bug-fix description from NFBugs.

Our findings show that among the 376 identified bug-inducing commits for 123 bugs, only 79 true positive bug-inducing commits are identified by the SZZ approach. Only 40 bugs have fully correctly identified bug-inducing changes.

We manually identify three reasons that account for all of falsely identified bug-inducing commits for non-functional bugs: 1) multi-purposes bug-fixing commits, 2) bug already being there, 3) not related to the bug shown in Table 3.2 and Figure 3.1. Reasons for not related to the bug include modifications of Javadoc or comments, the additional or removal of Java modifiers to variables, reverting changes, and some where we cannot find overlapping lines between removed lines in the fix commit and added lines in the identified bug-inducing commit.

Table 3.2: Breakdown of falsely identified non-functional bug-inducing commits into multi-purposes bug-fixing commits, 2) bug already being there, 3) not related to the bug.

	Total	Multi-purpose	Bug already being there	Not related to the bug
Java	54	8	7	39
Python	26	8	0	18

Multi-purposes bug-fixing commits. Our SZZ approach tracks all the code changes in the bug-fixing commits to identify bug-inducing commits. However, if a bug-fixing commit has multiple purposes, (i.e., some code changes in the commit are not done to fix the bug), tracking such changes would result in falsely identified bug-inducing commits. Unfortunately, we find a large number of cases where the bug-fixing commits are not dedicated to fixing a non-functional bug. For example, the commit message of commit **2391544** from the *Catacomb-Snatch* (in Java) project is "*Code*

Cleanup: Closed resource leaks, and removed or commented out unused code/resources, and did some code layout clean up (braces on ifs and correct indentation)". The first part of the commit message is clearly related to the non-functional bug of system resource leaks; while on the other hand, removing the unused code and the code layout cleanup may introduce noise to the SZZ approach.

In total, eight out of 54 falsely identified bug-inducing commits in Java and eight out of 26 in Python are due to multi-purpose bug-fixing commits. We would like to further verify whether these falsely identified bug-inducing commits are from a small number of bugs that are fixed in a multi-purpose manner. We find that these falsely identified bug-inducing commits originated from only five bugs in Java and one bug Python. Since there exists five bugs in Java and two bugs in Python with multi-purpose fixes, that have all true-positive identified bug-inducing commits. By manually looking at these bug-fixing commits, we find that all of the changes that are not associated with bug fixing are adding lines of source code, not affecting results of SZZ approaches. On the other hand, since the SZZ approach has become an application of automated bug-detection tools, multi-purpose bug-fixing commits have become a shortcoming of automatically applying SZZ for other downstream tasks, such as Just-In-Time Quality Assurance Kamei et al. (2013). Our results can be used to understand how badly the input noise actually affects the results of the SZZ approach and later impact its downstream tasks.

We also want to see whether commits of single-purpose or multi-purpose affect the performance of the SZZ approach in identifying bug-inducing commits. To do this, we also breakdown the single-purpose fixes based on whether the identified bug-inducing commits are false positive-bug-inducing commits or true positive-bug-inducing commits in Table 3.3. 23 out of 48 and 17 out of 25 single-purpose bug fixing commits, for Java and Python, respectively, lead to false positives. The results are comparable with the ones with multi-purpose bug fixing commits.

Bug already being there. For seven out of 54 falsely identified bug-inducing commits in Java and no cases in Python, when we examine the bug-inducing commit, we find that the non-functional bug already exists. Such a finding shows that in many cases, after the non-functional bugs are induced, developers may change the same lines of code, while not realising there exists a non-functional bug. In some other cases, the developers refactor or reformat the same line of code

Table 3.3: Number of single-purpose and multi-purpose bugs with false positive and all true positive bug-inducing changes.

	Single-purpose		Multi-purpose	
Language	with FP	with all TP	with FP	with all TP
Java	23	25	5	5
Python	17	8	1	2

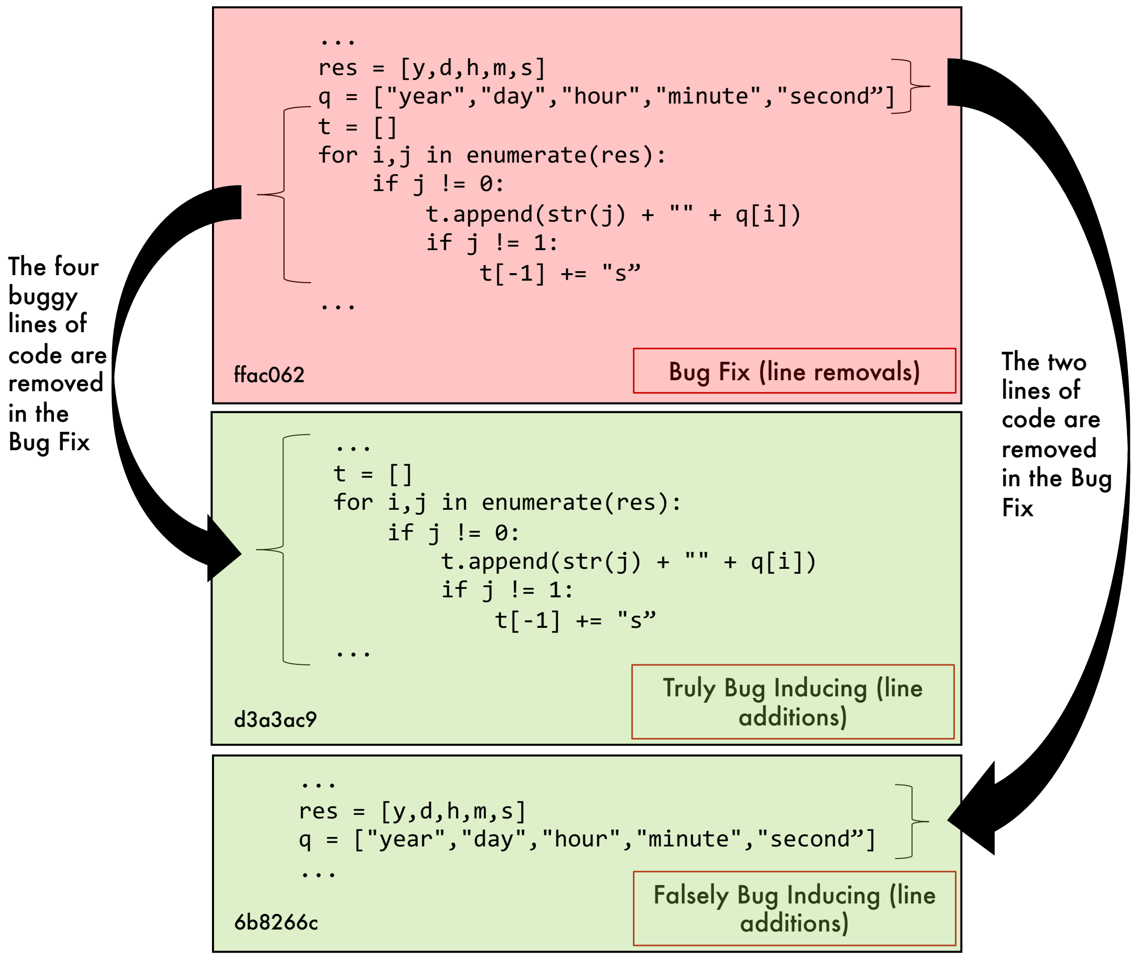
```
- results = ['total: %d' % sum(c.values())] + map(lambda n: '%s: %d' % (n[0], n[1]), c.items())
+ results = ['total: {}'.format(sum(c.values()))] + map(
+ lambda n: '{}: {}'.format(n[0], n[1]), c.items())
```

Figure 3.2: Simplified example of an ”Already there/Re-factoring” bug-inducing code line that was modified to semantically equivalent code

without actually changing the functionality. In either scenario, the SZZ approach may consider the later changes as bug-inducing instead of the original changes. This phenomenon is intuitive since non-functional bugs often take a long time to be discovered and fixed Nistor et al. (2013). Therefore, considering the most recent code change before the bug reporting date may not be a suitable heuristic for non-functional bugs. We did not find any cases of bugs already being there in our Python dataset.

We present an example of a semantically equivalent change wrongly identified as a bug-inducing change in Figure 3.2. The bug-inducing code lines that the SZZ approach looks for are **results = [‘total: ’.format(sum(c.values()))] + map(** and **lambda n: ‘{}: {}’.format(n[0], n[1]), c.items()**). These lines are shown as additions in this commit in green, however, in this same commit, the lines are also shown as a removal in red. The difference between these two is the string formatting, no logic is altered. The commit message is: “**String formatting**”. The SZZ approach stops at this commit and flags it as bug-inducing, since the bug-inducing code has technically been added in this line. However, although this is technically correct, this code was actually first introduced further back in time with different formatting in a different commit. The approach wrongly suspects the beautifying commit as a bug-inducing commit, which we describe as a case of “Bug already being there”.

Not related to the bug. For Java we find 39 out of 54 falsely identified bug-inducing commits in Java and 18 out of 26 in Python that are not related to the non-functional bug. Figure 3.3 shows


```
...
res = [y,d,h,m,s]
q = ["year","day","hour","minute","second”]
t = []
for i,j in enumerate(res):
    if j != 0:
        t.append(str(j) + "" + q[i])
        if j != 1:
            t[-1] += "s”
...
```

```
...
t = []
for i,j in enumerate(res):
    if j != 0:
        t.append(str(j) + "" + q[i])
        if j != 1:
            t[-1] += "s”
...
```

```
...
res = [y,d,h,m,s]
q = ["year","day","hour","minute","second”]
...
```
ffac062
Bug Fix (line removals)
d3a3ac9
Truly Bug Inducing (line additions)
6b8266c
Falsely Bug Inducing (line additions)
The four buggy lines of code are removed in the Bug Fix
The two lines of code are removed in the Bug Fix

Figure 3.3: Example of a bug-fix (ffac062) with two inducing commits, one bug-inducing (d3a3ac9), and one falsely bug-inducing (6b8266c)

two candidate commit bugs for a bug from the NFBugs dataset. The corresponding YAML file is shown in Listing 3.1. Upon manual analysis by the three reviewers, commit **6b8266c** in Figure 3.3 is identified as a false positive bug-inducing commit whereas commit **d3a3ac9** in Figure 3.3 is identified as a true positive bug-inducing commit. Commit **6b8266c** in Figure 3.3 was ruled out because it does not have a relation to the **builtins.list.append** and **use list comprehension instead of list.append loops to create lists efficiently** as stated in the Listing 3.1 bug description. From the description, the reviewers knew to look for a commit that refers to **builtins.list.append**.

We manually identify three reasons for falsely identified bug-inducing commits for non-functional bugs including 1) multi-purposes bug-fixing commits, 2) bug already being there, and 3) changes not related to the bug.

3.3 Automatically evaluating bug-inducing commits identified by the SZZ approach

In this step, we apply the three automated guidelines that are proposed by Costa et al. da Costa et al. (2017), i.e., 1) *Earliest bug appearance*, 2) *Future impact of a change*, and 3) *Realism of bug introduction*, in order to study whether such automated guidelines can help identify the falsely detected bug-inducing commits by the SZZ approach when used on non-functional bugs. The results are summarized in Table 3.4.

- *Earliest bug appearance.* For the corresponding candidate bug-inducing changes of each bug, we check the time of the change and the impacted version of the software. If the impacted version of the software is earlier than the candidate bug-inducing change, the candidate bug-inducing change is considered false.
- *Future impact of a change.* For each bug-inducing change, we calculate the count of induced bugs and the time-span of the induced bugs. If one change induces too many bugs or the induced bugs are across a long period of time, the bug-inducing change may be false.
- *Realism of bug introduction.* For each bug, we calculate the time-spans between the bug-inducing changes for each bug. If the time-span is too long, the bug-inducing changes may be false. We also consider bug-inducing changes after a bug-fixing change.

We use the results of the guidelines presented above for all bugs (functional and non-functional) as a baseline. For each of the two languages in NFBugs: Java and Python, we pick the top two projects with the largest number of reported bugs in the NFBugs dataset. We pick the top two projects since the project with the third highest number of reported bugs had significantly fewer reported bugs compared to the top two, making the sample size too small for comparison. In particular, we focus on four projects that are included in the NFBugs dataset, i.e, *Elasticsearch* and *Jenkins* (in Java) and *Falcon* and *Gae-boilerplate* (in Python). We choose these four projects since they contribute a large number of bugs for their respective language in the NFBugs dataset. We first extract all bugs (both functional and non-functional). For *Falcon* and *Jenkins* we look at their JIRA bug reports, meanwhile the other projects rely on GitHub's issue tracker. All of the issue trackers in our data are open to access. In order to study the functional and non-functional bugs that are around the same time period during development, we extract all the commits that are in a time period that is between six months before the reporting date of the first non-functionally bug and six months after the date of the last reported non-functional bug. We use the issue id (e.g., # with a number in GitHub issue tracker) in the commit message to link each issue and its issue fixing commit. We only consider the issues that are tagged with a *bug* label in either the GitHub issue tracker or JIRA. We assessed the functional-bug fixes in these four projects in the dataset, using a search for the issue id in the commit messages to identify the bug fixing commits, and then we further identify the bug-inducing commits related to them by running the SZZ approach. For the manual analysis of functional bugs, we pick a random sample with a confidence interval of $95\% \pm 10\%$ and end up with 96 identified bug-inducing commits in Java, and 91 identified bug-inducing commits in Python. We execute the MA-SZZ approach on the non-functional and functional bugs. Afterwards, we evaluate the three criteria: earliest bug appearance, future impact of a change, and realism of bug introduction using scripts that we created to calculate these guidelines proposed by Costa et al. da Costa et al. (2017).

Finally, we calculate the metrics that correspond to the three guidelines used by Costa et al. da Costa et al. (2017), i.e., 1) *Earliest bug appearance*, 2) *Future impact of a change*, and 3) *Realism of bug introduction*.

Results. We summarize the breakdown of results based on the three guidelines proposed Costa et al.'s da Costa et al. (2017) prior work, for identifying falsely detected bug-inducing commits by the SZZ approach. We compare our results to Costa et al.'s da Costa et al. (2017) in Table 3.4.

***Earliest bug appearance* and *Realism of bug introduction* da Costa et al. (2017) are not effective in identifying false candidate of inducing changes of non-functional bugs from SZZ**

Table 3.4: Percentages of bugs that will be identified by the three guidelines proposed by Costa et al. da Costa et al. (2017)

	Reported by the prior study	Reported by NFBugs
Future impact (commit level)	95%	54%
Earliest bug appearance (bug level)	0-3%	0-17%
Realism of a bug (bug level)	46%	3%

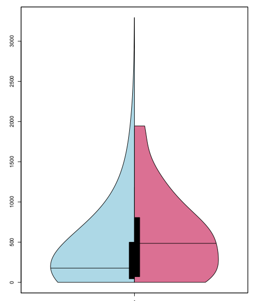
approaches. We observed only one case of *Earliest bug appearance* where there is a bug-inducing change for which the bug report for the corresponding fix was made even before the bug-inducing change date. This is unrealistic as it is not possible to report a bug in the software before the bug was introduced.

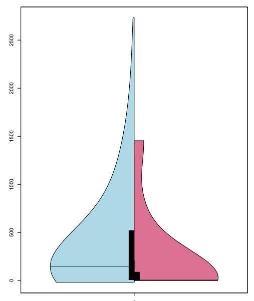
In their study, Costa et al. da Costa et al. (2017) uncovered some unrealistic changes. Costa et al. da Costa et al. (2017) found that 46% of bugs are caused by bug-inducing changes that span at least one year. It is unlikely that 46% of all bugs were caused by code changes that are years apart. We identify another type of unrealistic result in our own findings where a bug-inducing change was fixed or reported even before the bug change commit time. However, we only observed one case of *Realism of bug introduction* where for a bug, the time span between the bug-inducing changes is too long.

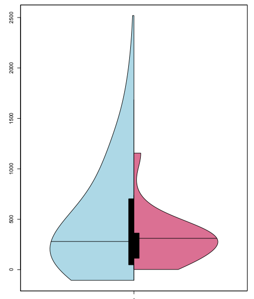
Therefore, since the *Earliest bug appearance* and *Realism of bug introduction* guidelines from Costa et al. da Costa et al. (2017) occur infrequently in the NFBugs dataset, we cannot reliably use them to identify false bug-inducing commit candidates.

***Future impact of a change* can be used as an indicator to identify false candidate of inducing changes of non-functional bugs from SZZ approaches.** We observed 36 cases of *Future impact of a change*, where one change induced many bugs or induced bugs across a long period of time. Part of Costa et al.'s da Costa et al. (2017) findings show that 29% of the bug-inducing changes lead to multiple future bugs that span at least one year. This suggests that SZZ approaches still lack mechanisms to accurately flag bug-inducing changes as it is unlikely that all 29% of the bug-inducing changes in a project introduce bugs that took years to be discovered.

Functional and non-functional bug-inducing commits do not overlap. We find that there


(a) Elasticsearch
p-value: 0.004


(b) Jenkins
p-value: 0.112


(c) Falcon
p-value: 0.796

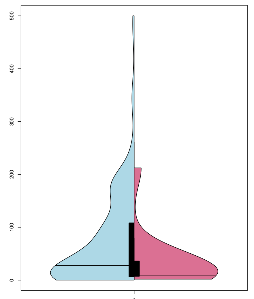
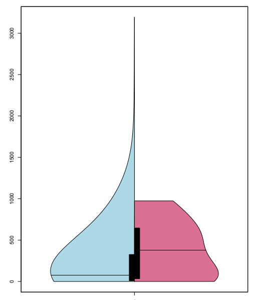
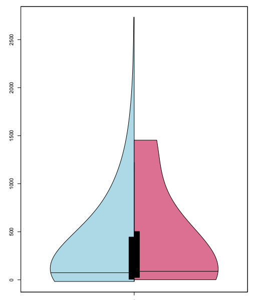
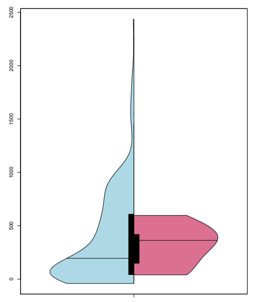

(d) Gae-boilerplate
p-value: 0.218

Figure 3.4: Comparison of Functional (left) and Non-Functional (right) bugs for days between inducing changes and fixing changes.


(a) Elasticsearch
p-value: 0.698


(b) Jenkins
p-value: 0.228


(c) Falcon
p-value: 0.796

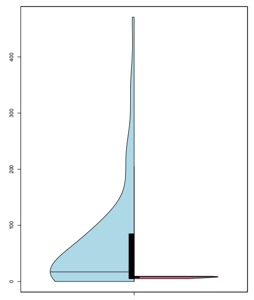

(d) Gae-boilerplate
p-value: 0.390

Figure 3.5: Comparison of Functional (left) and Non-Functional (right) bugs for days between earliest inducing changes and fixing changes.

are no cases within the four projects that we examined where a bug-inducing commit for a non-functional bug had also induced a functional bug. This indicates that non-functional bugs and functional bugs are quite different in nature. Non-functional bugs have may require different tooling requirements if they manifest differently in the source code, and are scattered across and require fixes in various parts of the software.

Based on Figure 3.4 and Figure 3.5 we can see that the median non-functional bugs appear to take a longer time on average to fix, compared to functional bugs. However, we cannot make any statistically significant conclusions due to the insufficient sample size. We use the data presented in violin plots in Figure 3.4 and Figure 3.5, and perform Wilcoxon rank sum tests, comparing the non-functional and functional bugs. Elasticsearch's comparison for functional bugs and non-functional bugs of days between all bug-inducing changes for fixing changes is the only statistically significant comparison (p=0.004).

The Cliff's Delta value for Elasticsearch's comparison for functional bugs and non-functional bugs of days between all bug-inducing changes for fixing changes is: -0.251. Cliff's Delta indicates that the type of bug (Functional or Non-Functional) has a small effect on the number of days between inducing changes and fixing changes for the Elasticsearch project.

The false results of SZZ on functional and non-functional bugs may be different. We further perform manual analysis on the functional bugs from Elasticsearch, Jenkins, Falcon, and Gae-boilerplate. Upon performing manual analysis on the functional identified bug-inducing commits following the steps listed in Chapter 3.2 we have calculated the Multi Kappa Fleiss score to be 0.736 - a moderate level of agreement for Java, and 0.822 - a substantial level of agreement for Python. Through our validation of the functional bugs, we further discarded three bugs from the Python projects and six bugs from the Java projects, because they were non-functional bugs, rather than functional bugs. For Python, out 56 of the examined bugs, 40 of them have at least one truly identified bug-inducing commit. For Java, out of 76 of the examined bugs, 31 of them have at least one truly identified bug-inducing commit. 50 out of the 88 identified bug-inducing commits were not truly bug-inducing for Python, and 59 out 89 identified bug-inducing commits in Java were not truly bug-inducing. In summation, 61.7% of the functional commits identified by the SZZ approach were not truly bug inducing compared to 80.0% for non-functional bugs. Similarly to Table 3.2, we break down the falsely identified functional bug-inducing commits in Table 3.5. Through observation, non-functional bugs experience more false positives bug-inducing commits because of multi-purposes bug-fixing commits. While it is possible for commits to be falsely identified due to multi-purpose bug-fixing commits in functional bugs, this occurs more rarely in functional bugs than in non-functional bugs. The bug already there reason for false positive bug incidence detection does appear in functional and non-functional bugs, however, they appear to experience different magnitudes of this fault: 20.2% for functional bugs, and 8.7% for non-functional bugs. Therefore, our results show that **not only is functional bug incidence detection more accurate than non-functional bug incidence detection, but functional bug false positive incidences also present themselves differently than for non-functional bugs**.

Table 3.5: Breakdown of falsely identified functional bug-inducing commits into multi-purposes bug-fixing commits, 2) bug already being there, 3) not related to the bug.

	Total	Multi-purpose	Bug already being there	Not related to the bug
Java	59	2	14	43
Python	50	0	8	42

Non-functional bug-inducing commits differ from functional bugs inducing commits. Guidelines to identify falsely detected bug-inducing commits by the SZZ approach such as Earliest bug appearance and Realism of bug introduction cannot be reliably used when using SZZ to detect non-functional bugs.

3.4 Discussion

Prior studies found that SZZ implementations still have room for improvement and suggest handling special cases to more accurately detect bug-inducing changes da Costa et al. (2017); S. Kim et al. (2006). The three cases of improvement include handling are:

- *Semantically equivalent changes*: Some implementations of the SZZ approach, including the one used for the purpose of the thesis, take into account comments, blank lines, indentation and white-space changes. However, even adjusted SZZ approaches, still have problems with other types of format changes such as reordering and renaming parameters Williams and Spacco (2008).

- *Directory or file re-names*: the SZZ approach cannot flag potential bug-inducing changes that are actually directory/file renaming changes since version control systems may not accurately track renamed files. Therefore, the SZZ approach cannot connect code changes that are performed on older versions of a renamed file da Costa et al. (2017). Broken historical links could be recovered heuristically by using repository mining techniques. Steidl, Hummel, and Juergens (2014)

- *Initial code important changes*: In cases where a project has been migrated from one version control system to another, SZZ approaches should trace back into the old version control system data since in those cases the initial commit of an version control system may not be the actual starting point of the project. However, current SZZ approaches are not able to trace back across different version control system and therefore cannot handle imported changes from a change in version control system da Costa et al. (2017).

We examine whether the above three improvements would help improve identifying the true bug-inducing commits for non-functional bugs. In order to perform such examination, we need to have all of the true bug-inducing commits for all of the non-functional bugs in NFBugs. Therefore, we examine the 28 non-functional bugs that do not have true bug-inducing commits detected from the first step of our study (cf. Chapter 3.3). We manually check the history of the source code in each project and try to identify the true bug-inducing commits for each of those bugs. In particular, the first author of the thesis proposed true bug-inducing commits for each bug and the third and fourth authors independently verified the proposed bug-inducing commits. If at least one author does not agree about a proposed bug-inducing commit, the three authors conduct further discussion about the specific case. If a proposed bug-inducing commit was deemed to be false, the first author did another round of examination to propose other commits as bug-inducing. This procedure was repeated until all three authors agreed that the bug-inducing commit was correctly identified.

We find that 60 out of 82 of the true bug-inducing commits for the non-functional bugs in Java and 21 out of 32 in Python, are actually the initial version of the corresponding code snippet. In other words, **the non-functional bugs were induced when developers first introduced the code into the project.**

Finally, we manually examine whether the above three improvements would help detect the true bug-inducing commits as bug-inducing. We only find two cases where addressing semantically equivalent cases can avoid mistakes, while the other two improvements do not have *any* impact on the results. We believe that this is the case since our subject systems all use Git as version control systems, where directory or file re-names are handled by the version control systems. In addition, the subject dataset may not have many cases of migrating version control systems. Therefore, the improvements proposed by prior studies may not help address the falsely detected bug-inducing commits for the non-functional bugs in NFBugs. However, they may indeed help when if the non-functional bugs are from a project that uses an older version control system (like Subversion) and/or has gone through migrations from one version control system to another.

3.5 Threats to Valdity

In this subchapter we discuss the threats to the validity of our research.

External validity.

Threats to external validity are concerned with the extent to which we can generalize our results. Our dataset contains a total of 65 open source GitHub projects: 40 Java projects and 25 Python projects. It is possible that our results might not generalize to all programming languages. However, since our dataset consists of both Java and Python projects, we are confident that our results should have the potential to be generalized to projects of other languages. Furthermore, to further mitigate any language bias, the SZZ approach we used is language agnostic. While Ohira et al.'s Ohira et al. (2015) dataset can be seen as complementary to the NFBugs dataset, it does not contain enough information in terms of bug descriptions, for us to fully be sure of the results when we do a manual evaluation as some information (the cause of the bug) that is available in the NFBugs dataset is missing. We require this information since we are not domain experts for the projects in the dataset. It should be possible, however, for domain experts to use Ohira et al.'s Ohira et al. (2015) dataset to replicate our study. For future work, we plan to extend our study of non-functional bugs from other datasets, e.g., the data from Ohira et al. Ohira et al. (2015).

While the reasons for falsely detected bug-inducing commits for non-functional bugs presented in this thesis are shown to have an effect on the detection of non-functional bugs, we do not claim that these reasons are unique to non-functional bugs. Because non-functional bugs are the focus of this thesis, we did not study how these reasons could affect functional bugs.

Construct validity.

Threats to construct validity are concerned with the validity of our conclusions within the constraints of the dataset we used. The dataset used for this thesis contains a total of 65 open source GitHub projects: 40 Java projects and 25 Python projects. Very few of the projects have an issue tracking system, and so for many, looking for the creation time of bug reports for a bug in the system was inapplicable. For these cases, we use the time when a bug fix was introduced instead of a bug report creation time to calculate the span between the introduction of a bug to the fix of that bug. A total of 27 of the bugs in the dataset had fully wrong identified bug-inducing changes, so we looked for false negatives, which are truly bug-inducing changes that have been missed and then included in the augmented NFBugs dataset. We may still miss some information that introduced the non-functional bugs, which we attempt to mitigate by having each reviewer study the bug fix description and the nature of the bug in order to evaluate whether the identified changes from the SZZ approach induced the bug along with the methods being reported in the bug descriptions provided by the NFBugs dataset.

Internal validity.

Threats to internal validity are concerned with how our experiments were designed. Our manual analysis of the candidate bug-inducing commits for known bug fixing commits were subject to our own opinion and could therefore be biased by the opinion of the experimenter. In order to mitigate bias, we had three reviewers analyze the candidate bug-inducing commits separately and in parallel. Following our manual analysis, we compute the Multi Kappa Fleiss scores of the agreement of the three individuals. We obtained moderate to strong levels of agreement. The reviewers later met together to discuss disagreements. These measures allow us to mitigate and measure the internal bias of our manual study. Moreover, for the bugs where the SZZ approach was not able to identify any truly bug inducing commits, we tried our best manually to look in the repository to find the commits that were bug-inducing for bugs. It is still possible that we may miss some bug inducing commits.

Nugroho et al. reported that different diff algorithms produce different bug-fix commit identification and they show that the histogram diff is better than the default diff setting in Git. However, in this thesis, we used the default diff setting in Git because we wanted to allow a comparison to prior studies da Costa et al. (2017) with as few confounding factors as possible. Future work and commercial approaches, should consider using histogram diff rather than the default diff setting to further improve the results presented in this work.

3.6 Chapter Summary

In this thesis we compared our evaluation of the SZZ approach to prior work that evaluates the approach on functional bugs. We use the NFBugs dataset as a ground truth for non-functional bugs. The NFBugs dataset contains 65 open source GitHub projects: 40 Java projects and 25 Python projects. We examine the 89 listed Java bugs and 43 listed Python bugs to uncover root causes for false bug-inducing commit detection that have not been found by previous studies. Furthermore, we conduct an empirical study to evaluate the performance of the SZZ approach in terms of its ability to locate bug-inducing commits in the code on the 132 listed bugs. Finally, we manually look at the results and discuss their implications.

Our findings show that the vast majority (297 out of 376) of the automatically identified bug-inducing commits by the SZZ approach for non-functional bugs are false positives. In addition, although there exists guidelines from prior study to assist in automatically identify falsely detected bug-inducing commits for functional bugs, these guidelines cannot be reliably used for non-functional bugs. Finally, the existing improvements to SZZ approach cannot help improve identifying the bug-inducing commits for non-functional bugs.

Our thesis is the first to focus exclusively on the use of the SZZ approach on non-functional bugs. Moreover, we augment the NFBugs dataset by adding a field to each bug description introducing bug-inducing commits that we manually analyzed as truly bug-inducing. By extending the dataset, we hope this information proves useful to help future research in locating bug-inducing commits, particularly with respect to non-functional bugs. Our findings indicate that new or adjusted tooling should be designed by considering the unique characteristics of non-functional bugs in order to accurately identify their bug inducing changes.

Chapter 4

Evaluating the impact of falsely detected performance bug-inducing changes in JIT models

4.1 Study Design

In this subchapter, we present the design of our study.

Dataset

In order to conduct our study, we need a dataset of performance bugs. However, existing datasets from prior studies on non-functional bugs Ohira et al. (2015) may not be systematically shared, e.g., the root-causes of each non-functional bug may not be clearly indicated, while such information is crucial in our study to verify the bug-inducing changes identified by the SZZ approach. Additionally, while there are existing datasets that contain a source of non-functional bugs Radu and Nadi (2019), only a small number of the bugs are concerned with performance. We therefore decided to create our own dataset using repositories highly concerned with performance.

To create our dataset, we employ the manually analyzed bugs from Cassandra Apache (2019) and Hadoop *Apache Hadoop* (n.d.) performance bugs found by Ding et al. Ding et al. (2020).

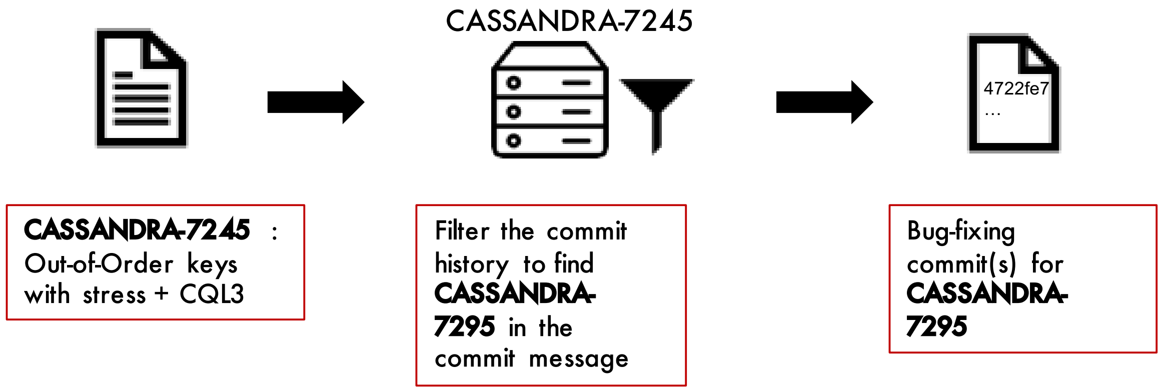

CASSANDRA-7245
4722fe7
...
CASSANDRA-7245 : Out-of-Order keys with stress + CQL3
Filter the commit history to find CASSANDRA-7295 in the commit message
Bug-fixing commit(s) for CASSANDRA-7295

Figure 4.1: An overview to extract fix commits given the JIRA issue ID: CASSANDRA-7245.

Hadoop is a free and open-source distributed system infrastructure providing processing in a reliable and efficient manner developed by the Apache Foundation. Cassandra is a free and open-source distributed NoSQL database management designed to handle large amounts of data, also developed by the Apache Foundation Syer, Shang, Jiang, and Hassan (2017). Hadoop and Cassandra are chosen as our dataset because they are highly concerned with performance and have been studied in prior research in mining performance data Chen and Shang (2017); Chen et al. (2014); Ding et al. (2020); Syer et al. (2017). Both repositories are open-source and also have JIRA issue tracking systems for identifying fix commits.

Bug-fixing Commit Extraction

To extract metrics for the bug-inducing commits, we start from bug reports, in the JIRA issue tracking systems. All of the closed bug-fixing reports identify the bug-fixing commit that closed the bug report. We therefore use these bug-fixing commits as input data for our study. If a bug report does not have a bug-fixing commit we ignore the bug-fix because we cannot ascertain which piece of code fixed the bug, and therefore do not have enough information to confidently study the bug. Figure 4.1 shows our approach to extract bug fix commits.

Data preparation

Similarly to Kamei et al. Kamei et al. (2013), we consider 13 factors grouped into four dimensions to filter our data, as presented in Table 4.1; these factors and dimensions are derived from the

Table 4.1: Summary of Change Measures from Kamei et al.'s work Kamei et al. (2013)

Dim.	Name	Definition
Diffusion	NS	Number of modified subsystems
	ND	Number of modified directories
	NF	Number of modified files
	Entropy	Distribution of modified code across each file
Size	LA	Lines of code added
	LD	Lines of code deleted
	LT	Lines of code in a file before the change
History	NDEV	The number of developers that changed the modified files
	AGE	The average time interval between the last and the current change
	NUC	The number of unique changes to the modified files
Experience	EXP	Developer experience
	REXP	Recent developer experience
	SEXP	Developer experience on a subsystem

source control repository data of a project. The rationale and related work for each measure can be found in Kamei et al.'s Kamei et al. (2013) work. The studied change metrics in Table 4.1 are grouped into four dimensions: diffusion properties, size properties, history properties, and author experience properties. We excluded the purpose dimension because we use the manually analyzed dataset from Ding et al. Ding et al. (2020) to identify the commits that are bug fix commits.

We use commit metric data provided by Commit Guru Rosen et al. (2015), a language-agnostic analytics and classification tool in line with our work. Commit Guru Rosen et al. (2015) is designed to be a prediction tool to identify and predict risky software commits, although it does not focus specifically on performance bugs. Other prior work has also used Commit Guru for building metric-based models Catolino (2017); Catolino, Di Nucci, and Ferrucci (2019); Kondo, German, Mizuno, and Choi (2020); Nayrolles and Hamou-Lhadj (2018); Tabassum, Minku, Feng, Cabral, and Song (2020). We use the tool provided by Commit Guru Rosen et al. (2015) to analyze the Cassandra and Hadoop repositories. Although Commit Guru Rosen et al. (2015) employs the SZZ approach, we use our own implementation of MA-SZZ as an initial classification to determine whether a commit induces a bug in the future. Commit Guru Rosen et al. (2015) employs JIT models upon analyzing each repository, while producing a downloadable CSV format file, where each row represents metrics and information related to a commit.

Our SZZ approach lists the mappings of bug fixing commit and identified bug inducing commits, along with which files contain the overlap of removed and added lines of code, which is not provided by Commit Guru Rosen et al. (2015). We also only consider the identified bug-inducing commits dated prior to the bug report submission date to filter out some false positives da Costa et al. (2017). The metrics and information provided by Commit Guru Rosen et al. (2015) is presented in Table 4.1. We add an additional column: **contains_bug** to label bug-inducing commits, which takes a Boolean value. For all of the models described in this thesis, we use the metrics in Table 4.1 as the independent variables to classify **contains_bug** (i.e., whether a commit induces a bug), the dependent variable.

We check the data for metrics that are highly correlated by using the Spearman statistic, ρ. The Spearman rank correlation estimates a rank-based measure of association and is resilient to data that is not normally distributed, unlike other types of correlation (e.g., Pearson) *Correlation (Pearson, Kendall, Spearman)* (n.d.). A hierarchical overview of the correlation amongst the metrics is constructed using variable clustering analysisMcIntosh and Kamei (2018). We remove metrics that are highly correlated where $\rho > 0.75$ Kamei et al. (2013).

To remove redundant commit metrics, we fit preliminary models that explain each dependent variable using the others, using the R^2 value of these models to measure how well each property is explained by the others. Similar to McIntosh et al. McIntosh and Kamei (2018), we use the *redun* function in the *rms* R package. This *redun* function iteratively drops the metric that is the most well-explained by the other metrics until either one of two conditions is satisfied:

- (1) no model achieves an $R^2 \geq 0.9$, or
- (2) removing a metric makes a previously dropped property no longer explainable, i.e., its preliminary model will no longer achieve an $R^2 \geq 0.9$

Similarly to Kamei et al. Kamei et al. (2013), we perform a logarithmic transformation to remove the effect of highly skewed change measures. A standard log transformation has been applied to each measure listed in Table 4.1.

Unbalanced classification problems may cause problems to many learning algorithms. We find

3,559 commits that are bug-inducing and 21,505 commits that are non-bug inducing for Cassandra. We also find 6,632 commits that are bug-inducing and 23,569 commits that are non-bug inducing for Hadoop. For Cassandra, there are over six times more commits that are non-bug inducing than commits that are bug-inducing and over three times more commits that are non-bug inducing than bug-inducing in the case of Hadoop.

We explore various methods to fix these data imbalances. We randomly upsample samples with replacement (i.e., replicating) of the minority class (i.e., bug-inducing) to make the minority class be the same size as the majority class (i.e., non-bug inducing) Tantithamthavorn, Hassan, and Matsumoto (2020). We also randomly downsample (i.e., reduce) samples of the majority class (i.e., non-bug-inducing) to make the number of majority modules be the same number as the minority class (i.e., bug-inducing) Tantithamthavorn et al. (2020).

A prior study Agrawal and Menzies (2018) found through a literature review that an overwhelming majority of SE thesiss (85%) use SMOTE to fix data imbalance, precisely when the data in the target class is overwhelmed by an over-abundance of information about everything else, except the target. To account for the data imbalance in commits that are bug-inducing and non bug-inducing in our models, we apply the R SMOTE function on the training data. SMOTE is an algorithm for handling unbalanced classification problems *DMwR* (n.d.). The general idea of this method is to artificially generate new examples of the minority class using the nearest neighbors of existing examples. Furthermore, the majority class examples are also under-sampled. In our case, non-bug inducing commits are under-sampled leading to a more balanced dataset *DMwR* (n.d.). For each case in the original dataset belonging to the minority class, new examples of that class will be generated by using the information from the k nearest neighbours of each example of the minority class *DMwR* (n.d.). We also use SMOTE to balance the number of performance and non-performance related commits within our BALANCED model.

Model construction

In this thesis, we build random forest models as our JIT classification models. Random forests construct each tree by using a different bootstrap sample. Assuming that the number of instances in the training set is N, a sample of these N cases is taken at random, with replacement. The random forest classifier uses a bootstrap approach internally to get an unbiased evaluation of the performance of a classifier. Contrary to standard decision trees, where each decision node is split using the best split among all variables, random forests split each node using the best subset among a randomly chosen subset of variables from each of the constructed trees Li, Shang, Zou, and E. Hassan (2017). Random forests are robust against overfitting and perform very well in terms of accuracy Li et al. (2017), suited for predicting performance related bug-inducing commits. Due to the imbalance of bug-inducing and non-bug inducing commits in our data, it is suitable to use random forest trees to not overfit the models on non-bug inducing commits. A study that compares 31 classifiers in software defect classification suggests that Random Forest outperforms other classifiers Ghotra, McIntosh, and Hassan (2015).

We also explore alternative models that may be used other than a random forest model: logistic regression modelling, support vector machine (SVM), and decision trees to determine how different models perform.

Manual analysis

For some performance fix commits, there are several corresponding identified bug-inducing commits (i.e., N:1). For example, commit `4722fe7` in Cassandra linked to the JIRA issue `CASSANDRA-7245`: *Out-of-Order keys with stress + CQL3*. There are 16 identified bug-inducing commits for the fix commit `4722fe7`. Out of 218 studied bug fix commits, we found that 126 of them have three or more identified bug-inducing commits. It seems unlikely that so many commits can cause one specific bug in the system. Therefore, some of these commits may be false positive bug-inducing commits da Costa et al. (2017). Falsely identified positive bug-inducing commits may occur because the SZZ approach still has room for improvement da Costa et al. (2017); S. Kim et al. (2006). Prior studies show that it is unlikely that all of the modifications made in a bug-fixing change are actually related to the bug-fix (e.g., it may contain an opportunistic refactoring) da Costa et al. (2017); Davies et al. (2014); S. Kim et al. (2006). Furthermore, fixes to non-functional bugs, including performance bugs may be scattered across the source code and might be separate from their bug-inducing locations in the source code, making it impossible for the SZZ approach to locate bug-inducing commits by tracing back.

Because there is no verified ground truth for performance bug-inducing commits, we further verify the 899 commits identified through the SZZ approach. We do this manually to determine the reliability of the raw results obtained through the original dataset.

The steps performed in this thesis for the manual analysis of bug-inducing commits are as follows:

- **Step 1A**: The reviewers look at the description of a bug fix commit issue JIRA.
- **Step 1B**: The reviewers examine the code from the mentioned bug fix commit corresponding to the JIRA issue.
- **Step 1C**: For each of the commits identified as bug-inducing commits by the SZZ approach, the reviewers examine the code in the commit and determine if it induced the bug identified in *Step 1A* based on their knowledge after studying the code.

Two reviewers perform this task separately and in parallel. Following the manual analysis, the Cohen's Kappa scores of the agreement of the two reviewers is calculated, ensuring moderate levels of agreement. The reviewers later met together to discuss disagreements until consensus was reached for all disagreements on whether an identified commit is bug-inducing or not.

The above mentioned steps are individually completed by all reviewers for all bugs remaining in the dataset. All reviewers must have the same classification (i.e., bug-inducing or not bug-inducing) for a identified commit, otherwise this is marked as a disagreement. All agreements and disagreements are recorded and used to calculate the Cohen's Kappa score, a robust statistic useful for either interrater or intrarater reliability testing McHugh (2012). Cohen's Kappa score is a standardized score, where 0 represents the amount of agreement that can be expected from random chance, and 1 represents perfect agreement between raters McHugh (2012). Afterwards, the reviewers meet and discuss any of the disagreements to reach a consensus for all identified bug-inducing commits.

The disagreements are resolved as follows:

- **Step 2A**: The reviewers re-read the JIRA issue description, the bug fix, and the bug-inducing commit in question.

- **Step 2B**: Each reviewer states the reason why they think the identified commit is bug-inducing or not bug-inducing.

- **Step 2C**: The reviewers discuss until all two agree on a final decision.

4.2 Case Study Results

4.2.1 RQ1: How well can JIT models predict performance bug-inducing commits?

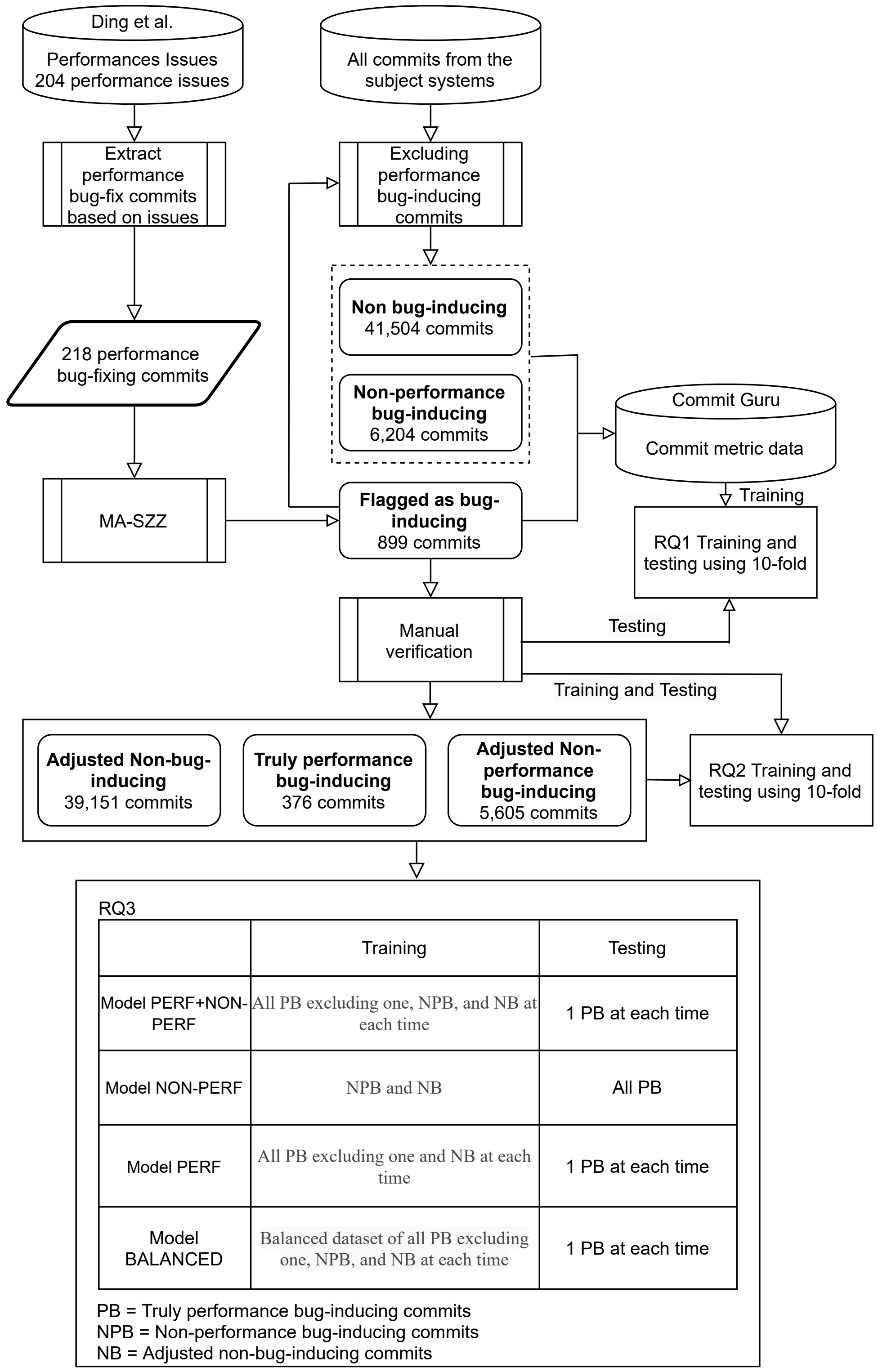
Motivation.

Because previous studies indiscriminately evaluate the SZZ approach on both functional and non-functional bugs without distinction, we seek a clear comparison between performance bugs and non-performance bugs. Zaman et al. found that developers spend more time fixing performance bugs rather than fixing non-performance bugs Zaman et al. (2011). Because performance bugs are a type of non-functional bug, an approach with a purpose such JIT models using the SZZ approach would be useful in helping developers locate where to fix a bug in the source code which can help fix future bugs that are similar to prior identified bugs. In this RQ, we evaluate the JIT models on their ability to predict bug-inducing commits identified by the SZZ approach.

Approach.

In order to evaluate the SZZ approach and the impact on JIT models with respect to performance bugs, we must first obtain a source of known performance bugs. Furthermore, we must have enough information for each bug to accurately determine the root causes for the bugs. We first find the corresponding bug fixing and bug-inducing changes for the bugs using the SZZ approach on our chosen datasets. Figure 4.2 provides an overview of our approach.

Using the JIRA issue IDs, we go through all commits in each subject repository, provided by Commit Guru Rosen et al. (2015). We do this to find linked commits that we assume are fix commits for those issues. We do this by creating a script where for each commit, if the **commit message** contains at least one of the JIRA performance issues from the list, we choose these commits as our bug fixing commits. In order to find the bug-inducing commits, we input the bug fixing commits into the SZZ approach, and we then identify the commits that the approach outputs as bug-inducing


Ding et al.
Performances Issues 204 performance issues
All commits from the subject systems
Extract performance bug-fix commits based on issues
Excluding performance bug-inducing commits
Non bug-inducing 41,504 commits
Non-performance bug-inducing 6,204 commits
218 performance bug-fixing commits
Commit Guru
Commit metric data
Training
MA-SZZ
Flagged as bug-inducing 899 commits
RQ1 Training and testing using 10-fold
Manual verification
Testing
Training and Testing
Adjusted Non-bug-inducing 39,151 commits
Truly performance bug-inducing 376 commits
Adjusted Non-performance bug-inducing 5,605 commits
RQ2 Training and testing using 10-fold
RQ3

	Training	Testing
Model PERF+NON-PERF	All PB excluding one, NPB, and NB at each time	1 PB at each time
Model NON-PERF	NPB and NB	All PB
Model PERF	All PB excluding one and NB at each time	1 PB at each time
Model BALANCED	Balanced dataset of all PB excluding one, NPB, and NB at each time	1 PB at each time

PB = Truly performance bug-inducing commits
NPB = Non-performance bug-inducing commits
NB = Adjusted non-bug-inducing commits

Figure 4.2: An overview of our approach to evaluate the impact of falsely identified performance bug inducing changes in JIT defect prediction models.

commits.

We perform 10-fold cross validation and evaluate the data of the JIT models for performance data and non-performance data on their recall, precision, F1, and AUC values. Since for this RQ (RQ1) as well as RQ2 we perform 10-fold cross validation, every single one of the commit instances are in the testing set exactly once. We evaluate all of the non-performance and performance classifications independently as a means to compare the models' power to predict performance and non-performance bug-inducing commits.

We use a default threshold of 0.5 to classify whether a commit is bug-inducing or not, where if the model-predicted probability of a bug-inducing commit is greater than 0.5, then the commit is classified as defect inducing; otherwise, it is classified as non-bug inducing Guo, Zimmermann, Nagappan, and Murphy (2010); Gyimothy et al. (2005); Kamei et al. (2013). Once all testing instances are evaluated, we then aggregate the results of all testing instances. We do this four times, experimenting with four sampling strategies: original (i.e., without any additional sampling strategy), downsampling Tantithamthavorn et al. (2020), upsampling Tantithamthavorn et al. (2020), and SMOTE *DMwR* (n.d.). We also explore alternative models that may be used other than a random forest model: logistic regression modelling, SVM, and decision trees in combination with SMOTE as summarized in Table 4.3.

Upon performing the manual analysis described in Chapter 4.1, we found that there are 899 identified performance bug-inducing commits: 327 from Cassandra and 572 from Hadoop shown in Figure 4.2. The 899 performance bug-inducing commits were identified through the SZZ approach, however, SZZ's assumptions about bug introduction (where removed line in a bug-fixing commit had introduced the bug) might not always be valid for performance bugs. These bug-inducing commits may or may not be truly bug-inducing commits, some of them may be false positives.

Since prior bug-inducing changes are the data that is fed into JIT defect prediction models to predict future bugs, it is important to determine whether or not the past changes are correctly labelled as bug-inducing or not bug-inducing. In this RQ we evaluate the performance of JIT models when using a manually verified dataset of performance bug-inducing commits as the training data. We first evaluate the JIT models on the performance related data output by the SZZ approach. However, because this data can contain errors we then re-evaluate the results after a manual analysis of the test dataset. We consider true positive detections in cases where our JIT models correctly identify non-performance and performance bug-inducing commits.

Results.

We find 120 and 83 JIRA issues related to performance issues for Cassandra and Hadoop respectively. For these issues, we find 179 and 80 commit fixes for Cassandra and Hadoop respectively. Using the SZZ approach on those fixes, we find that **there are 327 and 572 identified bug-inducing commits for Cassandra and Hadoop respectively, associated with the 148 out of 179 and 70 out 80 commit fixes identified by the SZZ approach** shown in Figure 4.2. For the remaining 41 bug fixes, there were no identified bug-inducing commits by the SZZ approach because the bug fixes only contained either changes to non-Java files or only additions.

Manual analysis: For some performance fix commits, there are several corresponding identified bug-inducing commits. For example, the commit: `4722fe7` in Cassandra linked to the JIRA issue `CASSANDRA-7245`, has 16 identified bug-inducing commits. Upon manual analysis, only eight of the 16 identified bug-inducing commits were truly bug-inducing changes. We show an example of a falsely identified bug-inducing commit: `84c0657` in Figure 4.3. The commit message is: *RefCount native frames from netty to avoid corruption bugs patch by tjake; reviewed by bes for CASSANDRA-7245* and the JIRA issue is *Out-of-Order keys with stress + CQL3*. The removed line in the fix commit flags commit `84c0657` as bug-inducing, as that line was introduced in a change in the commit history. The added and removed lines are snippets of code changes that are unrelated to the bug fix commit message and the JIRA issue description. Since the only code overlap that the fix commit `4722fe7` has with the identified inducing commit `84c0657` is the snippet removed shown in Figure 4.3 which is unrelated to the bug issue description, we rule it out as a false positive for introducing a bug in the context of `CASSANDRA-7245`.

For the manual examination of the identified bug-inducing commits, there were a total of 328 and 572 identified bug-inducing commits, and 141 and 92 disagreements from Cassandra and Hadoop, respectively. To quantitatively evaluate how often the reviewers agreed during manual evaluation, we use the Cohen's Kappa score McHugh (2012). The Cohen's Kappa score was 0.48 - a moderate level of agreement for Cassandra, and 0.43 - a moderate level of agreement for Hadoop McHugh (2012). Having a high Cohen's Kappa score reduces potential bias of a single

```
- List<AbstractWriteResponseHandler> responseHandlers =
-   new ArrayList<AbstractWriteResponseHandler>(mutations.size());
+ List<AbstractWriteResponseHandler> responseHandlers =
+   new ArrayList<>(mutations.size());
```

Figure 4.3: Simplified example of code changes in a fix commit that are unrelated to the bug description.

reviewer by hanging multiple reviewers, which is beneficial for producing reproducible results. The majority of the disagreements were due to either one of the reviewers misinterpreting the code, or not being sure whether the identified commit that contained a bug introduced the bug at the time or was a commit that retained a bug from a prior bug-inducing commit. Because the agreement score only ranked as moderate, each of the 233 disagreement was carefully re-examined and discussed until a consensus was reached between all reviewers.

Upon discussion between the two reviewers, it was found that 129 out of the 327 Cassandra commits and 244 out of the 572 Hadoop bug-inducing commits were manually verified to be truly bug-inducing commits. There are therefore 198 and 328 falsely identified bug-inducing commits from the SZZ approach for Cassandra and Hadoop, respectively. It should be noted that some commits were identified as bug-inducing commits for more than one bug. On the other hand, as mentioned earlier, for some performance fix commits, there are several corresponding identified bug-inducing commits. If the manually verified bug-inducing commit was a bug-inducing commit for more than one of the bugs (i.e., 1:N), we say it is a bug-inducing commit because it introduced a bug at the time of the commit In particular, 192 commits out of the 899 identified bug-inducing commits in our dataset (21.4%) were identified to induce more than one bug.

Our results upon performing 10-fold cross validation, on the JIT models are shown in Table 4.2 in the **Raw data** column.

The results of using raw SZZ data as JIT model input (i.e., the RQ1 *Raw data* column of Table 4.2) shows that the JIT models have worse prediction results when predicting non-performance bug inducing commits than the performance ones. However, this data is unvetted and may contain errors. Similar to RQ1, we find that the AUC scores and F1 scores for using SMOTE outperform the other three sampling strategies (i.e., when no sampling strategy is applied, downsampling, and upsampling) for both performance bugs and non-performance bugs.

After establishing a manually vetted ground truth, we actually find that the models are more accurate for non-performance commits rather than performance commits, shown by the F1 scores and AUC values of Table 4.2, column *RQ1 Verified data*. We correct the testing data used in RQ1, by using the ground truth of identified bug-inducing commits through manual analysis and re-build the models, as shown in Table 4.2 in the column *Verified data*. Note that both the *RQ1 Raw data* and *RQ1 Verified data* columns use the same model, which we will call *Unverified Data Model*. After establishing a ground truth through our manual analysis, rather than the initial one made up of the results directly provided by the SZZ approach, we re-evaluate our models on the correctly classified performance commits. Once again, we generally find that the AUC scores and F1 scores for using SMOTE outperforms the other three sampling strategies (i.e., no sampling strategy, downsampling, and upsampling), for performance and non-performance bugs.

The differing nature of performance bugs makes the SZZ approach a sub-optimal approach for identifying bug-inducing changes for performance bugs. Upon establishing a manually vetted ground truth, we find that the models are more accurate for non-performance commits rather than performance commits. Our training data contained raw data from the SZZ approach in terms of labelled bug-inducing commits that included falsely labelled performance bug-inducing commits. Since manual analysis is effort consuming, it will be valuable to determine whether manually validated data makes a significant difference for training models to classify performance bug-inducing commits, which we explore in RQ2.

Upon manual analysis of the identified bug-inducing commits, we find that 61.6% of them do not contain bug-inducing changes, yet were used in the JIT models. Our findings show that when using an established ground truth, the JIT model performs better in classifying non-performance commits, rather than performance commits.

4.2.2 RQ2: How does correcting falsely identified performance bug-inducing changes impact JIT models?

Motivation.

Just-In-Time models are used to identify bug-inducing code changes, and are trained using techniques that assume past bug-inducing changes are similar to future bug-inducing changes. In RQ1, our training data contained raw data from the SZZ approach in terms of labelled bug-inducing commits. The trained model may be biased by the errors introduced by the SZZ approach. Since manual analysis is time-consuming, in this RQ, we want to determine whether verified data makes a significant difference in training a better model to classify performance bug-inducing commits. In RQ1, we train the models on unverified data, while in this RQ, we train the models on verified data, in terms of performance related commits.

Approach.

In this RQ, we evaluate the JIT models on the performance related data output by the SZZ approach, correctly labelled by reviewers through a manual analysis. Similar to RQ1, we perform 10-fold cross validation on our training dataset, which now contains a verified ground truth mix of non-performance related data as well as correctly identified performance related bug-inducing commit instances. We call this the Verified Data Model. We consider a true positive detection in cases where our JIT models correctly identify non-performance and performance bug-inducing commits. For each repository (i.e., Cassandra and Hadoop), we create one model and evaluate non-performance and performance classifications independently. Upon performing 10-fold cross validation after correcting the performance commits manually, we find the recall, precision, F1, and AUC values of performance commits and non-performance commits.

To obtain a fair comparison, we further evaluate the differences between the classifications made in RQ1's Unverified Data Model and RQ2's Verified Data Model. We can do this because both the Unverified Data Models and Verified Data Models use the same ground truth of correctly labelled bug-inducing commits through the manual analysis step performed in RQ1. We compare the results shown in Table 4.2 in the columns *Verified Data* of RQ1 and *Verified Data* of RQ2. We evaluate the AUC Fawcett (2006) for performance commits and non-performance commits for each project (i.e., Cassandra and Hadoop). Furthermore, we use Wilcoxon's signed-rank test McDonald (2014) and Cohen's d Sawilowsky (2009) to compare the differences in results using the model produced in RQ1 and the one produced in RQ2.

Similar to RQ1, we explore alternative models that may be used other than a random forest model: logistic regression modelling, SVM, and decision trees.

Results.

Similar to RQ1, we find that the AUC scores and F1 scores for using SMOTE outperform the other three sampling strategies (i.e., no sampling strategy, downsampling, and upsampling) for both performance and non-performance bugs, shown in Table 4.2. Therefore, we include SMOTE as a sampling strategy step for further model building.

The results are summarized in Table 4.3 in the *Verified Data Model data* column. Similarly to RQ1, our threshold to classify whether a commit is bug-inducing or not is 0.5. We base our results off the random forest model, as it outperforms the logistic regression modelling, SVM, and decision trees models.

Performance bug-inducing commits: **By using manually verified performance data in the training data to build the models, we find that there is no statistical difference on the models' classification power. We find that there are overlapping correctly identified bug-inducing commits from both the Unverified Data Model and Verified Data Model.** We find that for performance bug-inducing commits, the Unverified Data Model and Verified Data Model have the correct and same classification for 212 performance related commit instances in Cassandra and 110 of the same performance related commit instances in Hadoop. As shown in Figure 4.4, in Cassandra, the Unverified Data Model was able to correctly classify 11 more performance commit instances that the Verified Data Model was not able to classify, while the Verified Data Model correctly classified 13 more performance commit instances that the Unverified Data Model was not able to classify as summarized in Figure 4.4. Additionally, as shown in Figure 4.4, in Hadoop, the Unverified Data Model was able to correctly classify 5 more performance commit instances that the Verified Data Model was not able to classify, while the Verified Data Model correctly classified 9 more performance commit instances that the Unverified Data Model was not able to classify. Therefore, based on the various non-overlapping results of both models, we believe that having correctly identified performance bug-inducing commits in the training data of JIT models can help correctly classify other performance bug-inducing commits.

Non-performance bug-inducing commits: We find that for non-performance bug-inducing commits, the models in Unverified Data and Verified Data have the correct and same classification

Table 4.2: Comparison of results of bug-inducing commit classification based on verified and unverified SZZ input data using 10-fold cross-validation, using four different sampling strategies in combination with random forest.

Original		***RQ1***				***RQ2***	
Training data		Raw data		Raw data		Verified data	
Testing data		Raw data		Verified data		Verified data	
Project		Cassandra	Hadoop	Cassandra	Hadoop	Cassandra	Hadoop
Performance bugs	Recall	48.9%	62.5%	49.2%	71.9%	42.6%	71.1%
	Precision	100.0%	100.0%	43.0%	45.3%	43.2%	46.0%
	F1	65.7%	76.9%	45.9%	55.6%	42.9%	55.8%
	AUC	0.745	0.812	0.502	0.578	0.503	0.584
Non-performance bugs	Recall	29.7%	45.3%	29.7.0%	45.3%	23.1%	42.4%
	Precision	54.5%	64.3%	54.5%	64.3%	58.2%	65.6%
	F1	38.5%	53.1%	38.5%	53.1%	33.1%	51.5%
	AUC	0.631	0.682	0.631	0.682	0.604	0.673
Downsampling		***RQ1***				***RQ2***	
Training data		Raw data		Raw data		Verified data	
Testing data		Raw data		Verified data		Verified data	
Project		Cassandra	Hadoop	Cassandra	Hadoop	Cassandra	Hadoop
Performance bugs	Recall	48.8%	62.5%	48.4%	71.9%	42.2%	71.1%
	Precision	100.0%	100.0%	42.4%	45.3%	42.6%	46.0%
	F1	65.6%	76.9%	45.2%	55.6%	42.4%	55.8%
	AUC	0.744	0.812	0.496	0.578	0.498	0.854
Non-performance bugs	Recall	29.9%	45.5%	29.9%	45.5%	22.9%	42.6%
	Precision	54.7%	64.4%	54.7%	64.4%	58.1%	65.6%
	F1	38.7%	53.3%	38.7%	53.3%	32.9%	51.6%
	AUC	0.633	0.683	0.633	0.683	0.603	0.674
Upsampling		***RQ1***				***RQ2***	
Training data		Raw data		Raw data		Verified data	
Testing data		Raw data		Verified data		Verified data	
Project		Cassandra	Hadoop	Cassandra	Hadoop	Cassandra	Hadoop
Performance bugs	Recall	49.3%	62.8%	49.2%	72.7%	42.2%	71.1%
	Precision	100.0%	100.0%	42.7%	45.6%	42.6%	46.0%
	F1	66.0%	77.1%	45.7%	56.0%	42.4%	55.8%
	AUC	0.746	0.814	0.469	0.582	0.498	0.584
Non-performance bugs	Recall	30.0%	45.4%	30.0%	45.4%	22.9%	42.6%
	Precision	54.7%	64.4%	54.7%	64.4%	57.8%	65.6%
	F1	38.7%	53.2%	38.7%	53.2%	32.8%	51.6%
	AUC	0.633	0.683	0.633	0.683	0.603	0.673
SMOTE		***RQ1***				***RQ2***	
Training data		Raw data		Raw data		Verified data	
Testing data		Raw data		Verified data		Verified data	
Project		Cassandra	Hadoop	Cassandra	Hadoop	Cassandra	Hadoop
Performance bugs	Recall	87.7%	90.9%	91.4%	89.8%	90.6%	93.4%
	Precision	100.0%	100.0%	73.8%	56.9%	86.7%	82.1%
	F1	93.4%	95.2%	81.7%	69.7%	88.6%	87.4%
	AUC	0.939	0.955	0.518	0.486	0.647	0.787
Non-performance bugs	Recall	87.0%	86.2%	87.0%	86.2%	88.2%	86.7%
	Precision	79.6%	86.7%	79.6%	86.7%	81.2%	87.0%
	F1	83.1%	86.5%	83.1%	86.5%	84.6%	86.9%
	AUC	0.842	0.869	0.842	0.869	0.845	0.869

Table 4.3: Comparison of results of bug-inducing commit classification based on verified and unverified SZZ input data using 10-fold cross-validation, using four different models in combination with SMOTE.

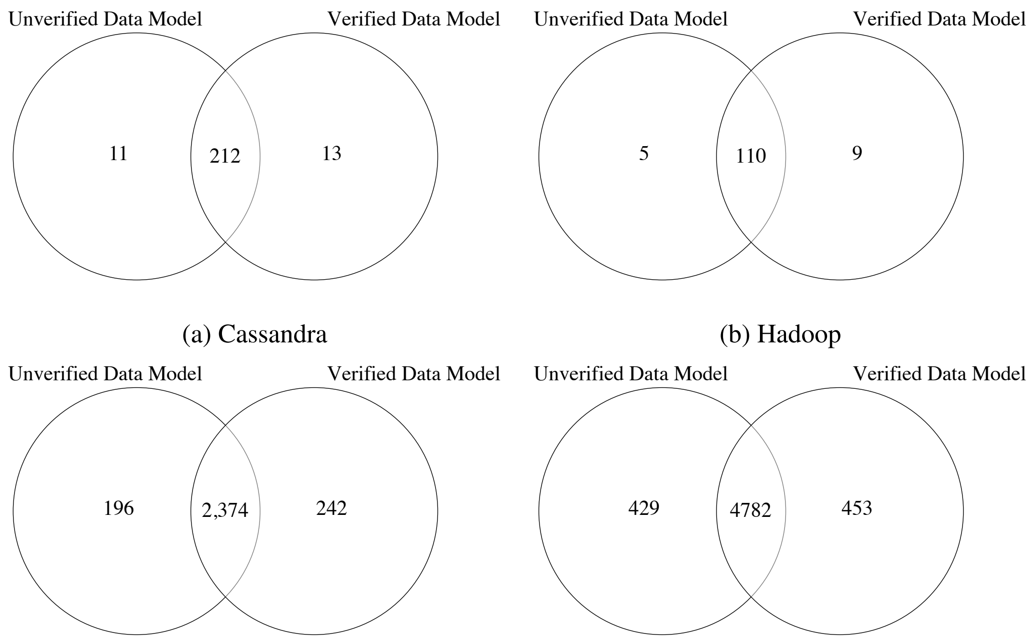
Random Forest		***RQ1***				***RQ2***	
Training data		Raw data		Raw data		Verified data	
Testing data		Raw data		Verified data		Verified data	
Project		Cassandra	Hadoop	Cassandra	Hadoop	Cassandra	Hadoop
Performance bugs	Recall	87.7%	90.9%	91.4%	89.8%	90.6%	93.4%
	Precision	100.0%	100.0%	73.8%	56.9%	86.7%	82.1%
	F1	93.4%	95.2%	81.7%	69.7%	88.6%	87.4%
	AUC	0.939	0.955	0.518	0.486	0.647	0.787
Non-performance bugs	Recall	87.0%	86.2%	87.0%	86.2%	88.2%	86.7%
	Precision	79.6%	86.7%	79.6%	86.7%	81.2%	87.0%
	F1	83.1%	86.5%	83.1%	86.5%	84.6%	86.9%
	AUC	0.842	0.869	0.842	0.869	0.845	0.869
Logistic Regression		***RQ1***				***RQ2***	
Training data		Raw data		Raw data		Verified data	
Testing data		Raw data		Verified data		Verified data	
Project		Cassandra	Hadoop	Cassandra	Hadoop	Cassandra	Hadoop
Performance bugs	Recall	86.4%	88.0%	88.1%	89.1%	88.1%	91.4%
	Precision	100.0%	100.0%	77.3%	57.3%	85.5%	69.0%
	F1	92.7%	93.6%	82.4%	69.7%	86.8%	78.7%
	AUC	0.932	0.940	0.503	0.507	0.490	0.547
Non-performance bugs	Recall	79.6%	77.2%	79.6%	77.2%	82.6%	77.6%
	Precision	85.4%	80.3%	85.4%	80.3%	72.9%	75.4%
	F1	82.4%	78.7%	82.4%	78.7%	77.4%	76.5%
	AUC	0.770	0.761	0.770	0.761	0.769	0.763
SVM		***RQ1***				***RQ2***	
Training data		Raw data		Raw data		Verified data	
Testing data		Raw data		Verified data		Verified data	
Project		Cassandra	Hadoop	Cassandra	Hadoop	Cassandra	Hadoop
Performance bugs	Recall	87.6%	89.7%	88.5%	91.4%	89.3%	93.4%
	Precision	100.0%	100.0%	74.5%	57.6%	83.7%	68.3%
	F1	93.4%	94.6%	80.9%	70.7%	86.4%	78.9%
	AUC	0.938	0.948	0.502	0.523	0.513	0.562
Non-performance bugs	Recall	83.5%	79.3%	83.5%	79.3%	84.3%	80.9%
	Precision	85.6%	80.7%	85.6%	80.7%	73.9%	76.3%
	F1	84.5%	80.0%	84.5%	80.0%	78.7%	78.5%
	AUC	0.777	0.772	0.777	0.772	0.781	0.780
Decision Trees		***RQ1***				***RQ2***	
Training data		Raw data		Raw data		Verified data	
Testing data		Raw data		Verified data		Verified data	
Project		Cassandra	Hadoop	Cassandra	Hadoop	Cassandra	Hadoop
Performance bugs	Recall	90.1%	87.8%	92.2%	90.6%	94.1%	94.5%
	Precision	100.0%	100.0%	75.3%	54.2%	83.5%	62.4%
	F1	94.8%	93.5%	82.9%	67.8%	88.4%	75.2%
	AUC	0.950	0.939	0.521	0.516	0.541	0.533
Non-performance bugs	Recall	86.2%	79.0%	86.2%	79.0%	88.3%	80.6%
	Precision	84.2%	78.1%	84.2%	79.1%	72.6%	72.0%
	F1	85.2%	78.5%	85.2%	78.5%	79.7%	76.0%
	AUC	0.777	0.749	0.777	0.749	0.785	0.747

Table 4.4: Comparison of results of bug-inducing commit classification based on verified SZZ input data using 10-fold cross-validation with Random Forest, using three different classification thresholds: 0.4, 0.5, and 0.6.

Threshold		**0.4**		**0.5**		**0.6**	
Project		Cassandra	Hadoop	Cassandra	Hadoop	Cassandra	Hadoop
Performance bugs	Recall	93.4%	96.1%	90.6%	93.4%	86.3%	89.8%
	Precision	89.1%	79.6%	86.7%	82.1%	90.5%	81.0%
	F1	91.2%	87.1%	88.6%	87.4%	88.4%	85.2%
	AUC	0.647	0.787	0.647	0.787	0.647	0.787
Non-performance bugs	Recall	90.7%	92.3%	88.2%	86.7%	81.8%	79.6%
	Precision	80.1%	83.3%	81.2%	87.0%	86.5%	89.9%
	F1	85.1%	87.6%	84.6%	86.9%	84.1%	84.4%
	AUC	0.845	0.869	0.845	0.869	0.845	0.869

for 2,374 non-performance bug-inducing commit instances in Cassandra and 4,782 same classification for non-performance bug-inducing commit instances in Hadoop. For Cassandra, the Unverified Data Model was able to correctly classify 196 more non-performance bug-inducing commit instances that the Verified Data Model was not able to classify, while the Verified Data Model correctly classified 242 more non-performance bug-inducing commit instances that the Unverified Data Model was not able to classify. For Hadoop, the Unverified Data Model was able to correctly classify 429 more non-performance bug-inducing commit instances that the Verified Data Model was not able to classify, while the Verified Data Model correctly classified 453 more non-performance bug-inducing commit instances that the Unverified Data Model was not able to classify. The results are summarized in Figure 4.4.

For performance commits, the results of RQ2's Verified Data Model are higher than those for RQ1's Unverified Data Model as shown by the AUC values in Table 4.2. **When comparing the results of both models for performance commits, although the results are *statistically different* for Hadoop (p-value = 0.00051), the effect size is classified as *small* (Cohen's d = 0.437)** Sawilowsky (2009)**, additionally the results are *not statistically different* for Cassandra (p-value = 0.068).** Similarly, as shown in Table 4.3, RQ2's Verified Data Model has higher AUC values than those for RQ1's Unverified Data Model for the SVM and Decision Trees models. The logistic regression model presents an AUC score for RQ1's Unverified Data Model that is statistically higher



(a) Cassandra (b) Hadoop

(c) Cassandra (d) Hadoop

Figure 4.4: Top: Comparison of correctly classified **performance bug-inducing commits**, before and after manual correction. Bottom: Comparison of correctly classified **non-performance bug-inducing commits**, before and after manual correction.

than RQ2's Verified Data Model. However, when comparing the results of both models for performance commits, we find that for both Hadoop and Cassandra the results of the Logistic Regression, SVM, and Decision Trees models are not statistically different.

For non-performance commits, the results of the Verified Data Model are slightly higher than those for the Unverified Data Model as shown by the AUC values in Table 4.2. However, when comparing the results of Verified Data Model and Unverified Data Model for non-performance commits, the results are *not statistically different* for both Cassandra (p-value = 0.291) and Hadoop (p-value = 0.464). Similarly, as shown in Table 3, RQ2's Verified Data Model has higher AUC values than those for RQ1's Unverified Data Model for the Logistic Regression and SVM models. The AUC score for RQ1's Unverified Data Model is higher than that of RQ2's Verified Data Model, when comparing the Decision Tree models. We find that the results are significantly different for the Logistic Regression models for Cassandra (p-value = 0.001) and the Decision Trees models for Hadoop (p-value = 0.003). However, both effect sizes are classified as trivial: a Cohen's d value of

0.068 for Cassandra, and a Cohen's d value of 0.004 for Hadoop.

These results indicate that manually verifying the results of SZZ to produce JIT models generally does not impact the models' classification power, and when it does (i.e., in the case of Cassandra performance and Hadoop non-performance) the changes only have a negligible to small effect on the classification power of these models. However, the changes in the models brought about by manual verification of performance commits do add new information that was not present in the original, non-verified model as shown in Figure 4.4. This indicates that **including correct performance bug-inducing commits in Just-In-Time training data can help with identifying other previously unidentified bug-inducing commits as well as classifying non-bug inducing commits**. Further insight into the classification power could benefit future model building. We investigate this further in RQ3.

Upon comparing the four models, we choose to use random forest models in combination with SMOTE for building future models, as it ranks the highest in terms of F1 and AUC scores for performance and non-performance bugs in both Hadoop and Cassandra, depicted by Table 4.3. While we do select the random forest model due to its higher predictive power, the overall trends uncovered are mostly model agnostic. The models present similar patterns, where the AUC scores for non-performance bugs is always higher than those of performance bugs.

We find that there is valuable knowledge present in non-performance bug-inducing commits that was not present in our sample of truly performance bug-inducing commits. Unfortunately, our truly performance bug-inducing commits sample is simply too small to contain all of the project's knowledge. We believe that performance bug localization would strongly benefit from large datasets of performance bugs.

Including manually verified bug-inducing data in the training data has minimal impact on the models overall classification power. Despite the low impact, we find that Unverified and Verified Data Models contain exclusive knowledge, which they show through exclusively correctly classified commit instances for performance related commits as well as non-bug inducing commits.

4.2.3 RQ3: Does only using the correct performance inducing changes as training data improve the JIT models when predicting other performance inducing changes?

Motivation.

In RQ2, we find that including verified performance bug-inducing commits in JIT training data can properly identify other bug-inducing commits as well as classify non-bug inducing commits, indicating that including correctly performance bug-inducing commits in Just-In-Time training data can help with identifying other bug-inducing commits as well as classifying non-bug inducing commits. If performance and non-performance bug-inducing commits have different characteristics, a model only based on performance bugs can better label performance bug-inducing changes. In this RQ, we evaluate Just-In-Time models solely on the manually verified performance commits identified by the two reviewers in RQ1, by using four different combinations of training data shown in Figure 4.2. Of the model combinations, we include one where the training data is only comprised of manually labelled bug-inducing commits, to see whether we need a separate model for predicting performance bugs. With the results of this RQ, we seek to determine how different training data influences the models' power to predict performance bug-inducing commits.

Approach.

We evaluate four JIT models with manually verified bug-inducing commits for each subject system to find what combination of training data is best suited for predicting performance related bug-inducing commits. We use *'X'* to denote the variable number of bug-inducing commits, shown in detail in Figure 4.2, for each subject system to evaluate the models below:

- **Model PERF+NON-PERF**: The training data is comprised of X-1 performance bug-inducing commits, the non-performance bug-inducing commits, and both all non-bug-inducing commits. The testing data is solely comprised of the *one* verified performance bug-inducing commit excluded from the training data. This split is repeated X times, and once all instances are evaluated. We then aggregate the results of all repetitions.

- **Model NON-PERF**: The training data is comprised of non-performance bug-inducing commits, and all non-bug-inducing commits. The testing data is comprised of the X verified

Table 4.5: Evaluation results of four JIT models with manually verified performance bug-inducing commits per subject system. The amount of true positive performance bug-inducing commits classified by each of the models is shown.

	Cassandra	Hadoop
Model PERF+NON-PERF	210	110
Model NON-PERF	201	111
Model PERF	185	94
Model BALANCED	209	102
Total	244	128

performance bug-inducing commits.

- **Model PERF**: The training data is comprised of X-1 performance bug-inducing commits, and all non-bug-inducing commits. The testing data is solely comprised of the *one* verified performance bug-inducing commit excluded from the training data. Similarly to Model PERF+NON-PERF the split is repeated X times, and all evaluation data is aggregated.

- **Model BALANCED**: The training data is comprised of X-1 performance bug-inducing commits, the non-performance bug-inducing commits, and all non-bug-inducing commits. The testing data is solely comprised of the *one* verified performance bug-inducing commit excluded from the training data. This split is repeated X times, and once all instances are evaluated. We then aggregate the results of all repetitions. Our performance bug-inducing commits sample is simply too small to contain all of the project's knowledge. Therefore, we produce this model to test the effect of a more balanced dataset. Contrary to Model PERF+NON-PERF, an extra balancing step is done to account for the imbalance in performance and non-performance related commits. We use the balancing approach outlined in the data preparation chapter of this thesis (i.e., Chapter 4.1).

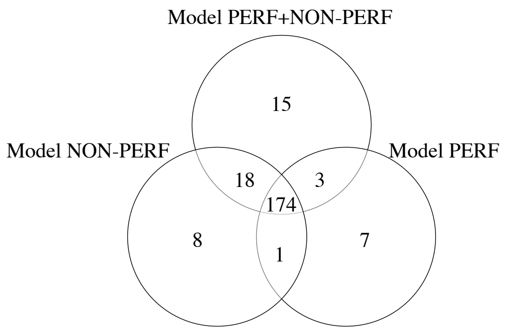
For each of the four models described above, each of the performance bug-inducing commits are tested on, exactly once.

Results.

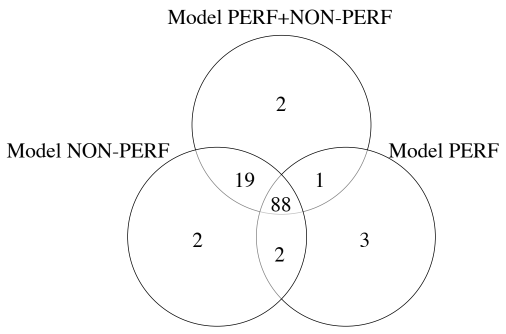
Model PERF performs the worst, while Model PERF+NON-PERF performs the best. Moreover, the difference between Model PERF+NON-PERF and Model PERF is that Model PERF+NON-PERF also trains on the non-performance bug inducing commits in addition to all commits except one performance bug-inducing commits and non-bug inducing commits. The single excluded performance bug-inducing commit is the one that the JIT model predicts on. Similarly to RQ1 and RQ2, our threshold to classify whether a commit is bug-inducing or not is 0.5. Including the non-performance bug-inducing commits in the training data increased the classification of truly performance bug-inducing commits by 10.2% in Cassandra and 12.5% in Hadoop, as shown in Table 4.5. This indicates that it is better for JIT model classification to include all commit types, possibly due to the small size of the performance bug data compared to non-performance bug data. The small performance bug data sample is not providing sufficient predictive power. This difference in internal knowledge is what allows Model PERF+NON-PERF to use information from non-performance bug-inducing data, which seems to have overlapping knowledge in terms of similar characteristics to performance bugs, helping with the classification of performance bugs.

The effect of truly performance bug-inducing commits (PB in Figure 4.2) Although the performance bug data alone may not be enough to accurately detect other performance bug-inducing commits, as shown by Model PERF performing worse than Model PERF+NON-PERF, it is still better to include them in the training data, as it still gives new information, show in Figure 4.5. Excluding the non-performance commits has a larger effect than excluding performance bug-inducing commits as shown in Table 4.5. This is likely because there are many more non-performance bug-inducing commits than performance bug-inducing commits. However, **performance bug-inducing commits do contain unique knowledge that is not contained within non-performance bug-inducing commits** as shown in Figure 4.5. By the same logic, this explains why Model PERF+NON-PERF performs the best out of the four models, since it contains both performance bug-inducing commits and non-performance bug-inducing commits. More data included in the training data has positive results on the models' classification power. As a future work, we propose enlarging the amount of training instances of performance bug-inducing data to determine if this can further improve the JIT models.

The results are summarized in Figure 4.5. The data shows that there is knowledge present in non-performance bug-inducing commits that was not present in our sample of truly performance bug-inducing commits. This is likely due to the sheer size difference between our non-performance



(a) Cassandra



(b) Hadoop

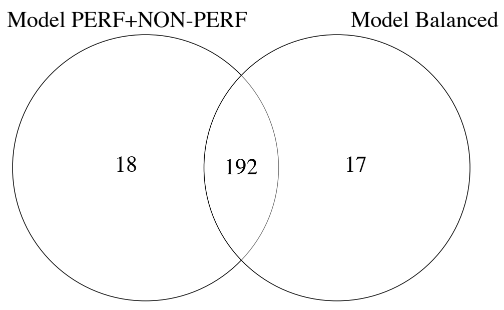
Figure 4.5: Comparison of Model PERF+NON-PERF, Model NON-PERF, and Model PERF of Cassandra (left) and Hadoop (right) of correctly classified performance bug-inducing commits.

bug-inducing commits and truly performance bug-inducing commits samples. Our truly performance bug-inducing commits sample is simply too small to contain all of the project's knowledge. Including correct performance bug-inducing data is therefore valuable to predict on other performance bug-inducing commits.

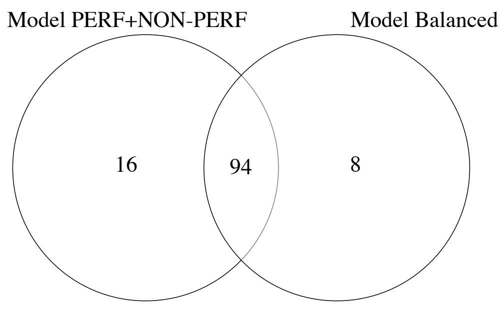
We therefore create a model called BALANCED, as shown in Table 4.5 where we use the R SMOTE function *DMwR* (n.d.) as described in Chapter 4.1, to balance the non-performance commits and the performance commits. As shown in Figure 4.6, we find that for performance related commits, Model PERF+NON-PERF and Model BALANCED have the correct and same classification for 192 performance related commit instances in Cassandra and 94 of the same performance related commit instances in Hadoop. As shown in Figure 4.6, balanced datasets (BALANCED) and large datasets (PERF+NON-PERF) each yield good, and slightly different results. Therefore, we suggest that JIT models aim for large and balanced training datasets.

When limited to not having enough performance commit related data, we find that using all commit data, i.e., truly performance bug-inducing commits, non-performance bug-inducing commits, and non-bug-inducing commits in the training data, still gives the best results. The rationale behind this result stems from the limited amount of performance bug-inducing commit data in our case study. It appears that more data, no matter the bug-type, is still superior to less data of a specific bug type (i.e., performance). Hence, in the absence of sufficient performance regression data, a "blind" model with all the bug-inducing commits still performs reasonably well.

Additionally, prior work shows that cross-prediction models Fukushima, Kamei, McIntosh, Yamashita, and Ubayashi (2014), where combining the data of several other projects to produce a larger pool of training data, works well in JIT defect prediction, which can be explored in future work for performance-related bugs. Leveraging other projects' labelled performance data would be able to enlarge the amount of training instances of performance bug-inducing data.



(a) Cassandra



(b) Hadoop

Figure 4.6: Comparison of Model PERF+NON-PERF and BALANCED of Cassandra (left) and Hadoop (right) of correctly classified performance bug-inducing commits.

Our findings show that it is better for JIT model classification to include all commit types, due to the small size of the performance bug data compared to non-performance bug data. Moreover, our results for Model BALANCED, indicate that larger samples of correct performance bug-inducing data are more valuable for predicting other performance bug-inducing commits. For future work, we propose enlarging the amount of training instances of performance bug-inducing data to determine if this can further improve the JIT models.

Compared to the amount of functional bugs, it is typical that the amount of performance bugs are relatively small in software projects Ding et al. (2020); Jin et al. (2012); Nistor et al. (2013); Radu and Nadi (2019). Therefore, the lack of data to build JIT bug prediction models for performance bugs may become a common challenge in practice. On the other hand, research has shown that transfer learning techniques can be used to leverage data from other projects in order to enlarge the training data for modeling and prediction Catolino et al. (2019). Therefore, future research may consider enlarging the amount of training instances of performance bug-inducing data by transfer learning techniques to further improve the JIT models.

Using correctly labelled performance bug-inducing commits in the training data doesn't result in an optimal model. We find that using all commit data: truly performance bug-inducing commits, non-performance bug-inducing commits, and non-bug-inducing commits in the training data results in better predicting performance bug-inducing commits.

4.3 Threats to Validity

In this section we discuss the threats to the validity of our research.

External validity.

Threats to external validity are concerned with the extent to which we can generalize our results. Although our study only focuses on 121 performance issues, the scale of our study is comparable to prior research on performance issues Zaman et al. (2012). We attempt to mitigate these issues by establishing our benchmark of performance issues based on an existing, manually verified dataset used in prior research Ding et al. (2020). Our findings might not be generalizable to other systems, therefore, for our future work, we propose increasing the quantity of training instances of performance bug-inducing data.

Construct validity.

Threats to construct validity are concerned with the validity of our conclusions within the constraints of the dataset we used. Very few of the projects have an issue tracking system, and so for many, looking for bug reports for that have keywords relating to performance in the system was inapplicable. In order for us to mitigate the constraints of drawing conclusions, the dataset we employed from prior work Ding et al. (2020) contains two Java projects: Cassandra and Hadoop, which are both highly concerned with performance and have been studied in prior research in mining performance data Chen and Shang (2017); Chen et al. (2014); Ding et al. (2020); Syer et al. (2017).

Internal validity.

Threats to internal validity are concerned with how our experiments were designed. Our manual analysis of the candidate bug-inducing commits for known bug fixing commits were subject to our own opinion and could therefore be biased by the opinion of the experimenter. In order to mitigate the risk of bias, we included two other reviewers in parallel, following with Cohen's Kappa to measure agreement between the reviewers. After reviewing separately, the reviewers later met together to discuss disagreements. These measures taken allow for us to mitigate and measure the internal bias of our manual study.

While the existence of non-performance bug-inducing commits in the training data may have an effect on classifying performance bug-inducing commits, it is also possible that performance bug-inducing commits in the training data may have an effect on classifying non-performance bug-inducing commits. For the purpose of the thesis, we focus on the impact of having non-performance related data on classifying performance bugs. We may focus on the effect that performance bug-inducing commits have on classifying non-performance commits as a future work.

We use 0.5 as a threshold of probability to classify whether a commit is bug-inducing or not. Since the choice of a threshold may bias our findings, we experiment with threshold values of 0.4 and 0.6 in Table 4.4. We notice that from the threshold values of 0.5 to 0.6, the precision and recall values cross each other, except for Hadoop performance bugs. Since we value both precision and recall equally, we choose 0.5 by default. We find that when changing the threshold, the recall and precision values change slightly, however the conclusions still hold, indicating that JIT performance bug prediction is not impacted substantially by the threshold.

4.4 Chapter Summary

In this thesis we present a study to highlight the nature of performance bugs, and its impact on Just-In-Time defect prediction models. Due to their nature, these bugs may be scattered across the source code and might be separate from their bug-inducing locations in the source code. This scattering may cause unreliable data to be fed into Just-In-Time defect prediction models. We conduct an empirical study on the results of the SZZ approach used for JIT defect prediction, concentrating on the use of JIT defect prediction to identify the inducing changes of the performance related bugs in Cassandra and Hadoop.

We find that in the data fed into the Just-In-Time models, for more than 57% of fix commits, there are several commits identified as bug-inducing. As shown in prior studies, it is unlikely that all bug fixing-changes are related to the bug-fix so we suspect some of the identified bug-inducing changes are unlikely to be correct. This is likely due to nature of performance bugs, which makes the SZZ approach a sub-optimal approach for identifying bug-inducing changes for performance bugs. Through manual analysis of 899 identified bug-inducing commits, we find that 372 of them are correctly identified, while the remaining 528 do not contain bug-inducing changes accounting for 61.6%, which are fed into the JIT models.

Although the SZZ approach does give incorrect results, due to the small number of performance bugs in the population of total bugs, this has little impact on the overall predictive power of the models. Our findings show that there is knowledge present in non-performance bug-inducing commits which was not present in our sample of truly performance bug-inducing commits. Our truly performance bug-inducing commits sample is simply too small to contain all of the projects' knowledge.

Our findings show that it is better for JIT model classification to include all commit types, possibly due to the small size of the performance bug data compared to non-performance bug data. Moreover, including performance bug-inducing commits in the Just-In-Time models' training data increases the percentages of correctly labelled performance bug-inducing commits, indicating that correct performance bug-inducing data is valuable for predicting on other performance bug-inducing commits. For future work, we propose enlarging the amount of training instances of performance bug-inducing data by transfer learning techniques in order to further improve the JIT models.

Chapter 5

Conclusion

5.1 Major topics addressed

Chapter 3 focuses on the impact of the SZZ approach on the non-functional bugs in the NFBugs dataset, and the performance bugs in Cassandra, and Hadoop. We manually examine whether each identified bug-inducing change is indeed the correct bug-inducing change. Our manual study shows that a large portion of non-functional bugs cannot be properly identified by the SZZ approach. We uncover root causes for false detection that have not been previously found, and evaluate the identified bug-inducing changes based on criteria from prior research. Our results may be used to assist in future research on non-functional bugs, and highlight the need to complement SZZ to accommodate the unique characteristics of non-functional bugs.

Chapter 4 performs an empirical study to evaluate model performance for JIT models by using them to identify bug-inducing code commits for performance related bugs. Our findings show that JIT defect prediction classifies non-performance bug-inducing commits better than performance bug-inducing commits. However, we find that manually correcting errors in the training data only slightly improves the models. In the absence of a large number of correctly labelled performance bug-inducing commits, our findings show that combining all available training data yields the best classification results.

5.2 Thesis contributions

The contributions of this thesis are as follows:

(1) To the best of our knowledge this is the first study to focus exclusively on the use of the SZZ approach to identify the inducing commits of non-functional bugs.

(2) We manually verified the validity of the SZZ approach on non-functional bugs, and determine potential problems with the approach in dealing with them, and make the data readily available.

(3) We augmented the NFBugs dataset by including bug fix descriptions that contain the commits where the true bug-inducing changes reside.

(4) To the best of our knowledge this is the first study to focus exclusively on performance bug-inducing changes in the context of Just-In-Time defect prediction models.

(5) We manually verified the validity of the SZZ approach on performance bug-inducing changes and make the data readily available.

(6) We evaluated the results of the Just-In-Time model bug prediction before and after our modifications after manually checking whether a commit that is labelled as bug-inducing is truly bug-inducing.

5.3 Future research

For the purpose of our work, we focus on the impact of having non-performance related data on classifying performance bugs. We may focus on the effect that performance bug-inducing commits have on classifying non-performance commits as a future work.

Our findings show that it is better for JIT model classification to include all commit types, possibly due to the small size of the performance bug data compared to non-performance bug data. Moreover, including performance bug-inducing commits in the Just-In-Time models' training data increases the percentages of correctly labelled performance bug-inducing commits, indicating that correct performance bug-inducing data is valuable for predicting on other performance bug-inducing commits. For future work, we propose enlarging the amount of training instances of performance bug-inducing data by transfer learning techniques in order to further improve the JIT models.

Additionally, prior work shows that cross-prediction models, where combining the data of several other projects to produce a larger pool of training data, works well in JIT defect prediction, which can be explored in future work for performance-related bugs. Leveraging other projects' labelled performance data would be able to enlarge the amount of training instances of performance bug-inducing data.

References

Agrawal, A., & Menzies, T. (2018). Is "better data" better than "better data miners"? on the benefits of tuning smote for defect prediction. In *Proceedings of the 40th international conference on software engineering* (p. 1050–1061). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3180155.3180197 doi: 10.1145/3180155.3180197

Apache. (2019, Dec). *apache/cassandra.* Retrieved from https://github.com/apache/cassandra

Apache hadoop. (n.d.). Retrieved from https://hadoop.apache.org/

Borg, M., Svensson, O., Berg, K., & Hansson, D. (2019). Szz unleashed: An open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project. In *Proceedings of the 3rd acm sigsoft international workshop on machine learning techniques for software quality evaluation* (p. 7–12). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3340482.3342742 doi: 10.1145/3340482.3342742

Bryant, R. E., & O'Hallaron, D. R. (2015). *Computer systems: A programmer's perspective* (3rd ed.). Pearson.

Catolino, G. (2017, 05). Just-in-time bug prediction in mobile applications: The domain matters!. doi: 10.1109/MOBILESoft.2017.58

Catolino, G., Di Nucci, D., & Ferrucci, F. (2019). Cross-project just-in-time bug prediction for mobile apps: An empirical assessment. In *2019 ieee/acm 6th international conference on mobile software engineering and systems (mobilesoft)* (p. 99-110). doi: 10.1109/

MOBILESoft.2019.00023

Chen, J., & Shang, W. (2017). An exploratory study of performance regression introducing code changes. In *2017 ieee international conference on software maintenance and evolution (icsme)* (p. 341-352). doi: 10.1109/ICSME.2017.13

Chen, J., Shang, W., & Shihab, E. (2020). Perfjit: Test-level just-in-time prediction for performance regression introducing commits. *IEEE Transactions on Software Engineering*, 1-1. doi: 10.1109/TSE.2020.3023955

Chen, T.-H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., & Flora, P. (2014). Detecting performance anti-patterns for applications developed using object-relational mapping. In *Proceedings of the 36th international conference on software engineering* (p. 1001–1012). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2568225.2568259 doi: 10.1145/2568225.2568259

Correlation (pearson, kendall, spearman). (n.d.). Retrieved from https://www.statisticssolutions.com/correlation-pearson-kendall-spearman/

da Costa, D. A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., & Hassan, A. E. (2017). A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, *43*(7), 641-657. doi: 10.1109/TSE.2016.2616306

Davies, S., Roper, M., & Wood, M. (2014, 01). Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process*, *26*. doi: 10.1002/smr.1619

Ding, Z., Chen, J., & Shang, W. (2020). Towards the use of the readily available tests from the release pipeline as performance tests. are we there yet? In *42nd international conference on software engineering, seoul, south korea.* doi: 10.1145/3377811.3380351

Dmwr. (n.d.). Retrieved from https://www.rdocumentation.org/packages/DMwR/versions/0.4.1/topics/SMOTE

Fan, Y., Xia, X., Costa, D., Lo, D., Hassan, A. E., & Li, S. (2019, 07). The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering*, *PP*, 1-1. doi: 10.1109/TSE.2019.2929761

Fawcett, T. (2006). An introduction to roc analysis. *Pattern Recognition Letters*, *27*(8), 861-874. doi: 10.1016/j.patrec.2005.10.010

Fukushima, T., Kamei, Y., McIntosh, S., Yamashita, K., & Ubayashi, N. (2014, 05). Studying just-in-time defect prediction using cross-project models. *Empirical Software Engineering*, *21*. doi: 10.1145/2597073.2597075

Ghotra, B., McIntosh, S., & Hassan, A. E. (2015). Revisiting the impact of classification techniques on the performance of defect prediction models. In *2015 ieee/acm 37th ieee international conference on software engineering* (Vol. 1, p. 789-800). doi: 10.1109/ICSE.2015.91

Glinz, M. (2007, Oct). On non-functional requirements. In *15th ieee international requirements engineering conference (re 2007)* (p. 21-26). doi: 10.1109/RE.2007.45

Grubb, P., & Takang, A. (2003). *Software maintenance - concepts and practice (2. ed.).*

Guindon, C. (n.d.). *Swt: The standard widget toolkit.* Retrieved from https://www.eclipse.org/swt/

Guo, P. J., Zimmermann, T., Nagappan, N., & Murphy, B. (2010). Characterizing and predicting which bugs get fixed: An empirical study of microsoft windows. In *Proceedings of the 32nd acm/ieee international conference on software engineering - volume 1* (p. 495–504). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1806799.1806871 doi: 10.1145/1806799.1806871

Gyimothy, T., Ferenc, R., & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, *31*(10), 897-910. doi: 10.1109/TSE.2005.112

Hamill, M., & Goseva-Popstojanova, K. (2014, September). Exploring the missing link: An empirical study of software fixes. *Softw. Test. Verif. Reliab.*, *24*(8), 684–705. Retrieved from https://doi.org/10.1002/stvr.1518 doi: 10.1002/stvr.1518

Hassan, A. E. (2009). Predicting faults using the complexity of code changes. In *Proceedings of the 31st international conference on software engineering* (p. 78–88). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1109/ICSE.2009.5070510 doi: 10.1109/ICSE.2009.5070510

Jin, G., Song, L., Shi, X., Scherpelz, J., & Lu, S. (2012, June). Understanding and detecting real-world performance bugs. *SIGPLAN Not.*, *47*(6), 77–88. Retrieved from https://doi.org/10.1145/2345156.2254075 doi: 10.1145/2345156.2254075

Jpace. (n.d.). *jpace/diffj*. Retrieved from https://github.com/jpace/diffj

Kamei, Y., Shihab, E., Adams, B., Hassan, A. E., Mockus, A., Sinha, A., & Ubayashi, N. (2013). A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, *39*(6), 757-773. doi: 10.1109/TSE.2012.70

Kim, M., & Lee, E. (2018). Are information retrieval-based bug localization techniques trustworthy? In *Proceedings of the 40th international conference on software engineering: Companion proceeedings* (p. 248–249). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3183440.3194954 doi: 10.1145/3183440.3194954

Kim, S., & Whitehead, E. J. (2006). How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on mining software repositories* (p. 173–174). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1137983.1138027 doi: 10.1145/1137983.1138027

Kim, S., Zimmermann, T., Pan, K., & Whitehead, E. J. J. (2006). Automatic identification of bug-introducing changes. In *Proceedings of the 21st ieee/acm international conference on automated software engineering* (p. 81–90). USA: IEEE Computer Society. Retrieved from https://doi.org/10.1109/ASE.2006.23 doi: 10.1109/ASE.2006.23

Kondo, M., German, D., Mizuno, O., & Choi, E. (2020, 01). The impact of context metrics on just-in-time defect prediction. *Empirical Software Engineering*, *25*. doi: 10.1007/s10664-019-09736-3

Kotonya, G., & Sommerville, I. (1998). *Requirements engineering: Processes and techniques* (1st ed.). Wiley Publishing.

LaToza, T. D., Venolia, G., & DeLine, R. (2006). Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th international conference on software engineering* (p. 492–501). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1134285.1134355 doi: 10.1145/1134285.1134355

Li, H., Shang, W., Zou, Y., & E. Hassan, A. (2017, August). Towards just-in-time suggestions for log changes. *Empirical Softw. Engg.*, *22*(4), 1831–1865. Retrieved from https://doi.org/10.1007/s10664-016-9467-z doi: 10.1007/s10664-016-9467-z

Liu Ping, Su Jin, & Yang Xinfeng. (2011, Dec). Research on software security vulnerability detection technology. In *Proceedings of 2011 international conference on computer science and network technology* (Vol. 3, p. 1873-1876). doi: 10.1109/ICCSNT.2011.6182335

Mahrous, H., & Malhotra, B. (2018, Aug). Managing publicly known security vulnerabilities in software systems. In *2018 16th annual conference on privacy, security and trust (pst)* (p. 1-10). doi: 10.1109/PST.2018.8514187

McDonald, J. H. (2014). Handbook of biological statistics. In (Vol. 3, p. 186-189). sparky house publishing Baltimore, MD.

McHugh, M. (2012, 10). Interrater reliability: The kappa statistic. *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB*, *22*, 276-82. doi: 10.11613/BM.2012.031

McIntosh, S., & Kamei, Y. (2018, May). Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering*, *44*(5), 412-428. doi: 10.1109/TSE.2017.2693980

Molyneaux, I. (2009). *The art of application performance testing: Help for programmers and quality assurance* (1st ed.). O'Reilly Media, Inc.

Nayrolles, M., & Hamou-Lhadj, A. (2018). Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *Proceedings of the 15th international conference on mining software repositories* (p. 153–164). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3196398.3196438 doi: 10.1145/3196398.3196438

Neto, E., Costa, D., & Kulesza, U. (2018, 03). The impact of refactoring changes on the szz algorithm: An empirical study.. doi: 10.1109/SANER.2018.8330225

Nistor, A., Jiang, T., & Tan, L. (2013). Discovering, reporting, and fixing performance bugs. In *2013 10th working conference on mining software repositories (msr)* (p. 237-246). doi: 10.1109/MSR.2013.6624035

Nugroho, Y. S., Hata, H., & Matsumoto, K. (2019, Sep). How different are different diff algorithms in git? *Empirical Software Engineering*, *25*(1), 790–823. Retrieved from http://dx.doi.org/10.1007/s10664-019-09772-z doi: 10.1007/s10664-019-09772-z

Ohira, M., Kashiwa, Y., Yamatani, Y., Yoshiyuki, H., Maeda, Y., Limsettho, N., … Matsumoto, K. (2015). A dataset of high impact bugs: Manually-classified issue reports. In *2015 ieee/acm 12th working conference on mining software repositories* (p. 518-521). doi: 10.1109/MSR.2015.78

Pan, K., Kim, S., & Whitehead, E. J. (2009, June). Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, *14*(3), 286–315. Retrieved from https://doi.org/10.1007/s10664-008-9077-5 doi: 10.1007/s10664-008-9077-5

Radu, A., & Nadi, S. (2019). A dataset of non-functional bugs. In *Proceedings of the 16th international conference on mining software repositories* (p. 399–403). IEEE Press. Retrieved from https://doi.org/10.1109/MSR.2019.00066 doi: 10.1109/MSR.2019.00066

Rodriguezperez, G., Nagappan, M., & Robles, G. (2020). Watch out for extrinsic bugs! a case study of their impact in just-in-time bug prediction models on the openstack project. *IEEE Transactions on Software Engineering*, 1-1. doi: 10.1109/TSE.2020.3021380

Rosen, C., Grawi, B., & Shihab, E. (2015). Commit guru: Analytics and risk prediction of software commits. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering* (p. 966–969). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2786805.2803183 doi: 10.1145/2786805.2803183

Sawilowsky, S. (2009, 11). New effect size rules of thumb. *Journal of Modern Applied Statistical Methods*, *8*, 597-599. doi: 10.22237/jmasm/1257035100

Sliwerski, J., Zimmermann, T., & Zeller, A. (2005, May). When do changes induce fixes? *SIGSOFT Softw. Eng. Notes*, *30*(4), 1–5. Retrieved from https://doi.org/10.1145/1082983.1083147 doi: 10.1145/1082983.1083147

Steidl, D., Hummel, B., & Juergens, E. (2014). Incremental origin analysis of source code files. In *Proceedings of the 11th working conference on mining software repositories* (p. 42–51). New York, NY, USA: Association for Computing Machinery. Retrieved from https://

doi.org/10.1145/2597073.2597111 doi: 10.1145/2597073.2597111

Syer, M. D., Shang, W., Jiang, Z. M., & Hassan, A. E. (2017, March). Continuous validation of performance test workloads. *Automated Software Engg.*, *24*(1), 189–231. Retrieved from https://doi.org/10.1007/s10515-016-0196-8 doi: 10.1007/s10515-016-0196-8

Tabassum, S., Minku, L. L., Feng, D., Cabral, G. G., & Song, L. (2020). An investigation of cross-project learning in online just-in-time software defect prediction. In *Proceedings of the acm/ieee 42nd international conference on software engineering* (p. 554–565). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3377811.3380403 doi: 10.1145/3377811.3380403

Tantithamthavorn, C., Hassan, A. E., & Matsumoto, K. (2020). The impact of class rebalancing techniques on the performance and interpretation of defect prediction models. *IEEE Transactions on Software Engineering*, *46*(11), 1200-1219. doi: 10.1109/TSE.2018.2876537

team, J. (n.d.). *Eclipse java development tools (jdt).* Retrieved from https://www.eclipse.org/jdt/

Tsakiltsidis, S., Miranskyy, A., & Mazzawi, E. (2016). On automatic detection of performance bugs. In *2016 ieee international symposium on software reliability engineering workshops (issrew)* (p. 132-139). doi: 10.1109/ISSREW.2016.43

Williams, C., & Spacco, J. (2008). Szz revisited: Verifying when changes induce fixes. In *Proceedings of the 2008 workshop on defects in large software systems* (p. 32–36). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1390817.1390826 doi: 10.1145/1390817.1390826

Williams, L., McGraw, G., & Migues, S. (2018, Sep.). Engineering security vulnerability prevention, detection, and response. *IEEE Software*, *35*(5), 76-80. doi: 10.1109/MS.2018.290110854

Yang, Y., Zhou, Y., Liu, J., Zhao, Y., Lu, H., Xu, L., … Leung, H. (2016). Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering* (p. 157–168). New York, NY, USA: Association for Computing

Machinery. Retrieved from https://doi.org/10.1145/2950290.2950353 doi: 10.1145/2950290.2950353

Zaman, S., Adams, B., & Hassan, A. E. (2011). Security versus performance bugs: A case study on firefox. In *Proceedings of the 8th working conference on mining software repositories* (p. 93–102). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1985441.1985457 doi: 10.1145/1985441.1985457

Zaman, S., Adams, B., & Hassan, A. E. (2012). A qualitative study on performance bugs. In *2012 9th ieee working conference on mining software repositories (msr)* (p. 199-208). doi: 10.1109/MSR.2012.6224281