Noname manuscript No.
(will be inserted by the editor)

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

Sophia Quach · Maxime Lamothe · Bram Adams · Yasutaka Kamei · Weiyi Shang

Received: date / Accepted: date

Abstract Performance bugs bear a heavy cost on both software developers and end-users. Tools to reduce the occurrence, impact, and repair time of performance bugs, can therefore provide key assistance for software developers racing to fix these bugs. 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. These commits can be reviewed while they are still fresh in developers' minds, reducing the costs of developing high-quality software. JIT models, however, leverage the SZZ approach to identify whether or not a change is bug-inducing. The fixes to performance bugs may be scattered across the source code, separated from their bug-inducing locations. The nature of performance bugs may make SZZ a sub-optimal approach for identifying their bug-inducing commits. Yet, prior studies that leverage or evaluate the SZZ approach do not distinguish performance bugs from other bugs, leading to potential bias in the results.

In this paper, we conduct an empirical study on the JIT defect prediction for performance bugs. We concentrate on SZZ's ability to identify the bug-inducing commits of performance bugs in two open-source projects, Cassandra, and Hadoop. We verify whether the bug-inducing commits found by SZZ are truly bug-inducing commits by manually examining these identified commits. Our manual examination includes cross referencing fix commits and JIRA bug reports. We evaluate

S.Quach · W.Shang
Department of Computer Science and Software Engineering, Concordia University, Montreal, QC, Canada
E-mail: {s_quach,max_lam,shang}@encs.concordia.ca

Y.Kamei
Faculty of Information Science and Electrical Engineering, Kyushu University, Japan
E-mail: kamei@ait.kyushu-u.ac.jp

B.Adams
Queen's School of Computing, Queen's University, Canada
E-mail: bram@cs.queensu.ca

M.Lamothe
Department of Computer and Software Engineering, Polytechnique, Montreal, QC, Canada
E-mail: maxime.lamothe@polymtl.ca

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, i.e., the SZZ approach does introduce errors when identifying bug-inducing commits. However, we find that manually correcting these 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 (i.e., truly performance bug-inducing commits, non-performance bug-inducing commits, and non-bug-inducing commits) yields the best classification results.

Keywords software engineering · defect prediction bugs · just-in-time · performance

1 Introduction

Software bugs, and the techniques that software engineers develop to fix and prevent them, are an important part of software engineering [1–3]. Bugs are costly to fix and increase maintenance effort [4]. In a world that is ever more reliant on software, 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-introducing changes and help development teams avoid future bugs by learning from their mistakes [1–3].

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 [3]. *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. However, JIT quality assurance relies on the SZZ approach, and must therefore grapple with the drawbacks of SZZ. 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.

The SZZ approach is a technique that attempts to find the source-code commit that first introduced a software bug [5, 6]. However, the SZZ approach, similarly to other bug localization techniques, is not perfect. Previous studies show that the SZZ approach can mislabel some changes as bug introducing, even if they are not [5]. Mislabels include semantically equivalent changes, directory or file renames, and initial code importing changes [3]. Although an evaluation framework exists to evaluate the various implementations of the SZZ approach that attempt to remedy these issues [7], 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, or more specifically performance bugs.

Developers often spend more time fixing performance bugs than fixing non-performance bugs [8]. 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 [8]. 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 [9]. Hamill and Goseva-Popstojanova [9] 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). For example, if a code change introduces a performance issue, this performance issue may be fixed and improved in a different part of the system, for example by changing configuration parameters. Meanwhile, functional bugs can have a single concrete introducing commit that can be tracked down through the SZZ approach [6]. Due to the scattered nature of non-functional bugs, we suspect that the SZZ approach might perform worse on performance bugs than on functional bugs.

Prior studies indiscriminately evaluate the SZZ approach on both functional and non-functional bugs without distinction, while seeking a clear comparison between performance bugs and non-performance bugs. We aim to know whether or not performance bugs are well predicted by JIT defect prediction. In this paper, 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 paper, 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.

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.

We formulate our study into three research questions:

– *RQ1: How well can JIT models predict performance bug-inducing commits?*
  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.

- *RQ2: How does correcting falsely identified performance bug-inducing bugs impact JIT models?*
  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.
- *RQ3: Does only using the correct performance inducing bugs as training data improve the JIT models when predicting other performance inducing bugs?*
  Our findings show that only using correctly labelled performance bug-inducing commits in the training data does not result in an optimal model. Indeed, using more data appears to be the optimal solution. 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 this paper:

- 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.[1]

Paper organization. The rest of the paper is organized as follows: Section 2 introduces background concepts including the SZZ approach and Just-In-Time defect prediction. Section 3 describes the design of our study. Section 4 presents our case study results addressing our research questions. Section 5 presents the threats to the validity of our work. Section 6 presents prior work related to the work presented in this paper. Finally, Section 7 concludes the paper.

2 Background

The following section describes the concepts that are necessary to understand our study.

2.1 SZZ

The first SZZ approach B-SZZ (the basic SZZ implementation) was defined by Sliwerski et al. [6], 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 1 shows an example of the SZZ approach, with a bug fix

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

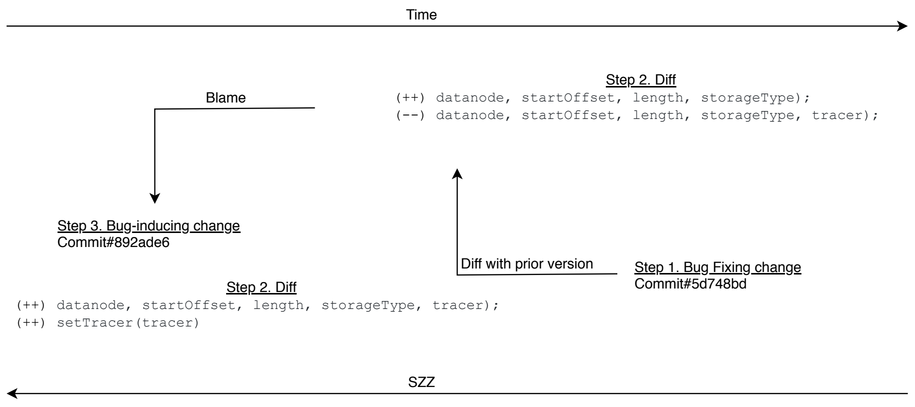

Time
Blame
Step 2. Diff
(++) datanode, startOffset, length, storageType);
(--) datanode, startOffset, length, storageType, tracer);
Step 3. Bug-inducing change
Commit#892ade6
Diff with prior version
Step 1. Bug Fixing change
Commit#5d748bd
Step 2. Diff
(++) datanode, startOffset, length, storageType, tracer);
(++) setTracer(tracer)
SZZ

Fig. 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.

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 [5]. Costa et al. [7] 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 [7].

Costa et al. [7] find that B-SZZ has the lowest disagreement ratio in general (0%-9%), followed by the MA-SZZ (0%-17%) [7]. 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 [7], as it is similar to B-SZZ with some improvements, such as excluding style changes and including usage of an annotation graph [5], and the removal of meta-changes [7]. MA-SZZ is also used in several studies including work on Just-In-Time defect prediction, to identify bug-inducing changes [3, 10] as a ground truth for building the prediction models. Additionally, prior work in refactoring changes, such as RA-SZZ [11], uses MA-SZZ by incorporating it with RefDiff to propose a refactoring aware SZZ implementation [11]. 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.

Because the SZZ approach does not provide perfect precision and recall of bug-inducing commits, it is important to understand the nature of its results. In an ideal scenario the SZZ approach can indeed identify bug-inducing changes. In this ideal case, we consider the results to be changes identified by the SZZ approach that are truly bug-inducing changes, or true positives. These results can directly be used by developers to find the root cause of a known bug. However, if the SZZ approach wrongly identifies changes that are not truly bug-inducing changes as bug-inducing, we consider those to be false positives. False positives can waste developer time by causing them to needlessly look at faultless code. Meanwhile, false negatives are truly bug-inducing changes that were missed by the SZZ approach and require different tools or manual investigation to find the root cause of a bug. As discussed above, various modifications of the SZZ approach such as MA-SZZ, attempt to improve the true positive rate and reduce the false positive and false negative rates.

2.2 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. [12] 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. [12] 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 [12]. To know whether or not a change introduces a defect, Kamei et al. [12] 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 [12]. 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. [10] 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.

```
- 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;
```

Fig. 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;
```

Fig. 3: Simplified example of code changes in a fix commit for a non-performance bug.

2.3 Performance Bugs

Jin et al [13] 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 and Figure 3, respectively.

Performance bug in Figure 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 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. [14] use keywords as the heuristics to identify performance issue reports. Ding et al. [14] start by using the keywords that are used in prior research [13, 15]. In order to avoid missing performance issues, the list of keywords was expanded by using word embedding. Ding et al. [14] 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. [14] only focus on the issue reports that have the type Bug and are labeled as *Resolved* or *Fixed*. In total, Ding et al. [14] find 121 performance-related issue reports in Cassandra and 83 in Hadoop. We use this vetted performance-related data in our paper.

3 Study Design

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

3.1 Dataset

In order to conduct our study, we need a dataset of performance bugs. However, existing datasets from prior studies on non-functional bugs [16] 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 [17], 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 [18] and Hadoop [19] performance bugs found by Ding et al. [14]. 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 [20]. 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 [14, 20–22]. Both repositories are open-source and also have JIRA issue tracking systems for identifying fix commits.

3.2 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 shows our approach to extract bug fix commits.

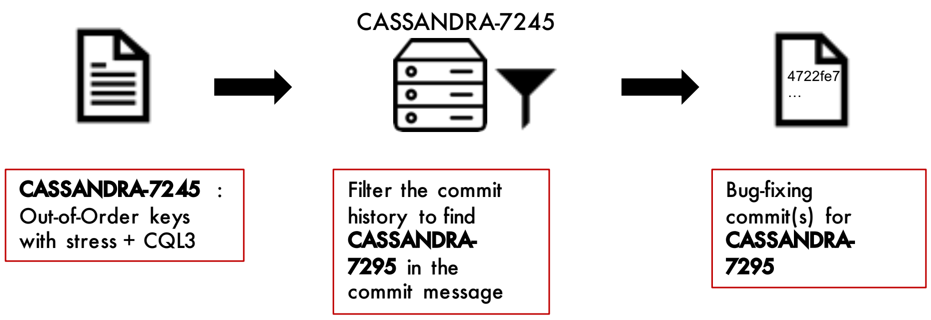

CASSANDRA-7245
4722fe7
...
CASSANDRA-7245 : Out-of-Order keys with stress + CQL3
Filter the commit history to find CASSANDRA-7295 in the commit message
Bug-fixing commit(s) for CASSANDRA-7295

Fig. 4: An overview to extract fix commits given the JIRA issue ID: CASSANDRA-7245.

3.3 Data preparation

Table 1: Summary of Change Measures from Kamei et al.'s work [3]

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

Similarly to Kamei et al. [3], we consider 13 factors grouped into four dimensions to filter our data, as presented in Table 1; these factors and dimensions are derived from the source control repository data of a project. The rationale and related work for each measure can be found in Kamei et al.'s [3] work. The studied change metrics in Table 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. [14] to identify the commits that are bug fix commits.

We use commit metric data provided by Commit Guru [23], a language-agnostic analytics and classification tool in line with our work. Commit Guru [23] 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 [24–28]. We use the tool provided by Commit Guru [23] to analyze the Cassandra and Hadoop repositories. Although Commit Guru [23] 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 [23] 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 [23]. We also only consider the identified bug-inducing commits dated prior to the bug report submission date to filter out some false positives [7]. The metrics and information provided by Commit Guru [23] is presented in Table 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 paper, we use the metrics in Table 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) [29]. A hierarchical overview of the correlation amongst the metrics is constructed using variable clustering analysis [10]. We remove metrics that are highly correlated where $\rho > 0.75$ [3].

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. [10], 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. [3], 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 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) [30]. 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) [30].

A prior study [31] found through a literature review that an overwhelming majority of SE papers (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 [32]. 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 undersampled. In our case, non-bug inducing commits are under-sampled leading to a more balanced dataset [32]. 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 [32]. We also use SMOTE to balance the number of performance and non-performance related commits within our BALANCED model.

3.4 Model construction

In this paper, 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 [33]. Random forests are robust against overfitting and perform very well in terms of accuracy [33], 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 [34].

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.

3.5 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 [7]. Falsely identified positive bug-inducing commits may occur because the SZZ approach still has room for improvement [5, 7]. 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) [5, 7, 35]. 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 paper 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 [36]. 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 [36]. 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 Case Study Results

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 [8]. 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 5 provides an overview of our approach.

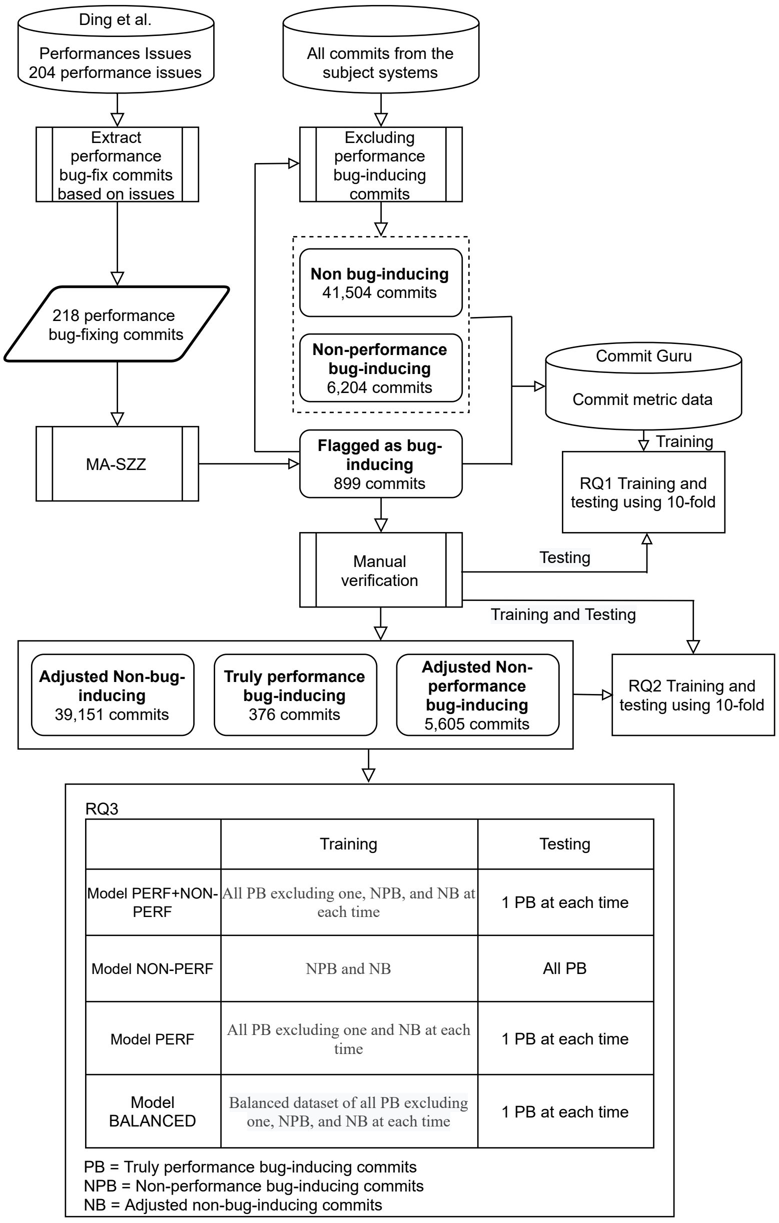
Using the JIRA issue IDs, we go through all commits in each subject repository, provided by Commit Guru [23]. We do this to find linked commits that we assume are fix commits for those issues. We do this by creating a script where for each commit, if the **commit message** contains at least one of the 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 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 [1, 3, 37]. 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 [30], upsampling [30], and SMOTE [32]. 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 3.

Upon performing the manual analysis described in Section 3, we found that there are 899 identified performance bug-inducing commits: 327 from Cassandra and 572 from Hadoop shown in Figure 5. 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


Ding et al.
Performances Issues
204 performance issues
Extract performance bug-fix commits based on issues
218 performance bug-fixing commits
MA-SZZ
All commits from the subject systems
Excluding performance bug-inducing commits
Non bug-inducing
41,504 commits
Non-performance bug-inducing
6,204 commits
Flagged as bug-inducing
899 commits
Commit Guru
Commit metric data
Training
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

Fig. 5: An overview of our approach to evaluate the impact of falsely identified performance bug inducing changes in JIT defect prediction models.

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 5. 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 6. 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 6 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 [36]. 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 [36]. Having a high Cohen's Kappa score reduces potential bias of a single 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

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

Fig. 6: Simplified example of code changes in a fix commit that are unrelated to the bug description.

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 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 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 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 rebuild the models, as shown in Table 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.

RQ2: How does correcting falsely identified performance bug-inducing bugs 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 2 in the columns *Verified Data* of RQ1 and *Verified Data* of RQ2. We evaluate the AUC [38] for performance commits and non-performance commits for each project (i.e., Cassandra and Hadoop). Furthermore, we use Wilcoxon's signed-rank test [39] and Cohen's d [40] 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

Table 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

in Table 2. Therefore, we include SMOTE as a sampling strategy step for further model building.

The results are summarized in Table 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 7, 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 7. Additionally, as shown in Figure 7, 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 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 7.

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 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) [40], additionally the results are *not statistically different* for Cassandra (p-value = 0.068).** 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 SVM and Decision Trees models. The logistic regression model presents an AUC score for RQ1's

Table 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

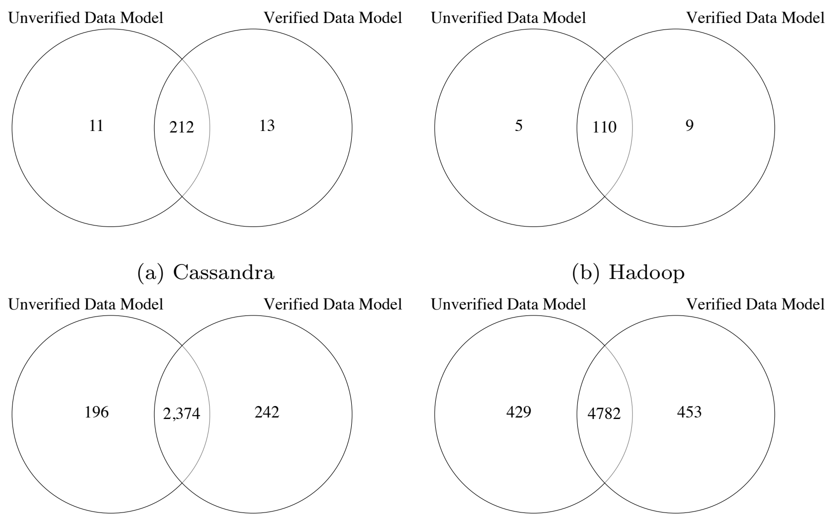
Logistic Regression		***RQ1***				***RQ2***	
Training data		Raw data		Raw data		Verified data	
Testing data		Raw data		Verified data		Verified data	
Project		Cassandra	Hadoop	Cassandra	Hadoop	Cassandra	Hadoop
Performance bugs	Recall	86.4%	88.0%	88.1%	89.1%	88.1%	91.4%
	Precision	100.0%	100.0%	77.3%	57.3%	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: 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



(a) Cassandra (b) Hadoop

(c) Cassandra (d) Hadoop

Fig. 7: 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.

Unverified Data Model that is statistically higher 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 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 7 and Figure 7. 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 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.

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

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 5. 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 5, 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 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 section of this paper (i.e., Section 3.3).

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

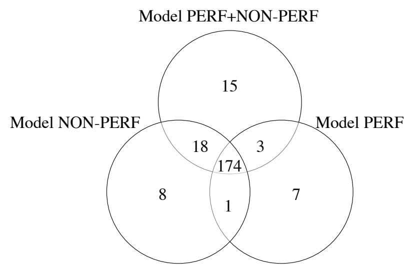
Table 5: Evaluation results of four JIT models with manually verified performance bug-inducing commits per subject system. The amount of true positive performance bug-inducing commits classified by each of the models is shown.

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

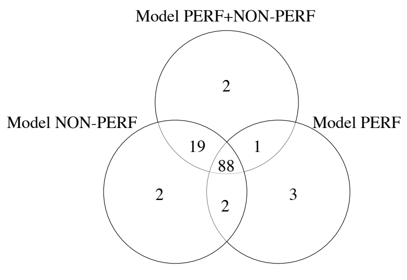
and Model PERF is that Model PERF+NON-PERF also trains on the non-performance bug inducing commits in addition to all commits except one performance bug-inducing commits and non-bug inducing commits. The single excluded performance bug-inducing commit is the one that the JIT model predicts on. Similarly 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 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 5) 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 8. Excluding the non-performance commits has a larger effect than excluding performance bug-inducing commits as shown in Table 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 8. 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 8. 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 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. In-



(a) Cassandra



(b) Hadoop

Fig. 8: 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.

cluding 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 5 where we use the R SMOTE function [32] as described in Section 3, to balance the non-performance commits and the performance commits. As shown in Figure 9, 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 9, 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 [41], 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.

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

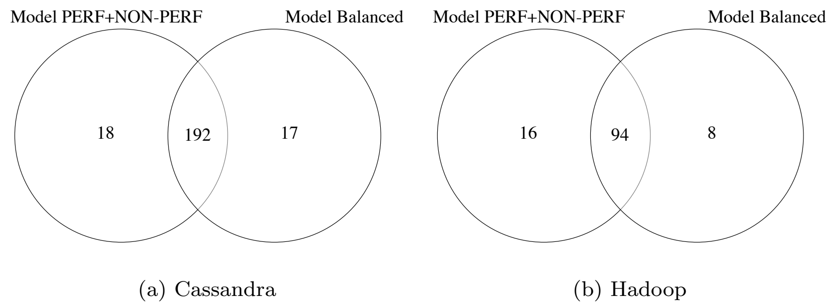


Fig. 9: Comparison of Model PERF+NON-PERF and BALANCED of Cassandra (left) and Hadoop (right) of correctly classified performance bug-inducing commits.

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 [13, 14, 17, 42]. 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 [25]. 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.

5 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 [15]. We attempt to mitigate these issues by establishing our benchmark of performance issues based on an existing, manually verified dataset used in prior research [14]. 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 [14] 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 [14, 20–22].

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 paper, 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. 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.

6 Related Work

6.1 Evaluation and improvement of SZZ

Kim et al. present algorithms to automatically and accurately identify bug-introducing changes, they compare their algorithms to SZZ [5]. They reduced the incidence of false positives and false negatives by using a combination of annotation graphs and ignoring 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 our paper, we evaluate our Cassandra and Hadoop datasets with the bug inducing changes found by SZZ. Furthermore, we concentrate on the SZZ approach [5], as it is used in JIT defect prediction.

Williams et al. revisit SZZ by outlining several improvements to the approach [43]. 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 [44]. Furthermore, they verify how often bug-inducing changes identified by the SZZ approach are truly bug-inducing changes. In our paper, 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. [7] introduced a framework to evaluate the results of SZZ 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 [7]. The framework evaluates the approach with three criteria: the earliest bug appearance, the future impact of changes, and the realism of bug introduction [7]. 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-introducing changes. A single bug-introducing change may be blamed for introducing hundreds of future bugs and SZZ implementations report at least 46% of the bugs are caused by bug-inducing changes that are years apart from one another [7]. Our study builds 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.

6.2 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 [8]. Tools can help reduce the cost overhead caused by these bugs [13]. In this study we focus on the applicability of the SZZ approach to determine the root cause of these non-functional bugs.

Jin et al. [13] 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 [13]. 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. [8] 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 [8]. The study shows the importance of identifying root causes for performance issues and evaluating the impact of changes on performance issues. Our paper 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. [42] 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 [45,46]. 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 [47,48]. 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 [49]. Extrinsic bug introducing-changes cannot be ident ified by the version control system of the software. Rodriguez-Perez et al. [49] study the impact of extrinsic bugs in JIT models, by replicating Kamei et al.'s recent paper in which they analyze the performance of JIT models [3]. Rodriguez-Perez et al. [49] remove the extrinsic bugs from their data, as all bugs were previously considered to be intrinsic. Findings from Rodriguez-Perez et al. [49] 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. [50] 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. [50] 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 [51]. 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. [3], which employ the SZZ approach. Our study investigates the predictive power of supervised models in effort-aware JIT defect prediction [3] 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. [52] 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 [52]. 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.

7 Conclusions

In this paper 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.

Acknowledgements

This research was partially supported by JSPS KAKENHI Japan (Grant Numbers: 21H04877, JP18H03222) and JSPS International Joint Research Program with SNSF (Project "SENSOR").

References

1. T. Gyimothy, R. Ferenc, and I. Siket, “Empirical validation of object-oriented metrics on open source software for fault prediction,” *IEEE Transactions on Software Engineering*, vol. 31, no. 10, pp. 897–910, Oct 2005.
2. A. E. Hassan, “Predicting faults using the complexity of code changes,” in *2009 IEEE 31st International Conference on Software Engineering*, May 2009, pp. 78–88.
3. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 757–773, June 2013.
4. T. D. LaToza, G. Venolia, and R. DeLine, “Maintaining mental models: A study of developer work habits,” in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE ’06. New York, NY, USA: ACM, 2006, pp. 492–501.
5. S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead, “Automatic identification of bug-introducing changes,” in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*, Sep. 2006, pp. 81–90.
6. J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–5, May 2005.
7. D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, “A framework for evaluating the results of the szz approach for identifying bug-introducing changes,” *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, July 2017.
8. S. Zaman, B. Adams, and A. E. Hassan, “Security versus performance bugs: A case study on firefox,” in *Proceedings of the 8th Working Conference on Mining Software Repositories*, ser. MSR ’11. New York, NY, USA: ACM, 2011, pp. 93–102.
9. M. Hamill and K. Goseva-Popstojanova, “Exploring the missing link: An empirical study of software fixes,” *Softw. Test. Verif. Reliab.*, vol. 24, no. 8, p. 684–705, Sep. 2014. [Online]. Available: https://doi.org/10.1002/stvr.1518
10. S. McIntosh and Y. Kamei, “Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction,” *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, May 2018.
11. E. Neto, D. Costa, and U. Kulesza, “The impact of refactoring changes on the szz algorithm: An empirical study,” 03 2018.
12. Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, june 2013.
13. G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, “Understanding and detecting real-world performance bugs,” *SIGPLAN Not.*, vol. 47, no. 6, pp. 77–88, Jun. 2012.
14. Z. Ding, J. Chen, and W. Shang, “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*, 2020.
15. S. Zaman, B. Adams, and A. E. Hassan, “A qualitative study on performance bugs,” *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*, pp. 199–208, 2012.
16. M. Ohira, Y. Kashiwa, Y. Yamatani, H. Yoshiyuki, Y. Maeda, N. Limsettho, K. Fujino, H. Hata, A. Ihara, and K. Matsumoto, “A dataset of high impact bugs: Manually-classified issue reports,” in *Proceedings of the 12th Working Conference on Mining Software Repositories*, ser. MSR ’15. Piscataway, NJ, USA: IEEE Press, 2015, pp. 518–521.
17. A. Radu and S. Nadi, “A dataset of non-functional bugs,” in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR ’19. Piscataway, NJ, USA: IEEE Press, 2019, pp. 399–403.
18. Apache, “apache/cassandra,” Dec 2019. [Online]. Available: https://github.com/apache/cassandra
19. “Apache hadoop.” [Online]. Available: https://hadoop.apache.org/
20. M. D. Syer, W. Shang, Z. M. Jiang, and A. E. Hassan, “Continuous validation of performance test workloads,” *Automated Software Engg.*, vol. 24, no. 1, p. 189–231, Mar. 2017. [Online]. Available: https://doi.org/10.1007/s10515-016-0196-8
21. J. Chen and W. Shang, “An exploratory study of performance regression introducing code changes,” in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 341–352.

22. T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, “Detecting performance anti-patterns for applications developed using object-relational mapping,” in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 1001–1012. [Online]. Available: https://doi.org/10.1145/2568225.2568259
23. C. Rosen, B. Grawi, and E. Shihab, “Commit guru: Analytics and risk prediction of software commits,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 966–969. [Online]. Available: http://doi.acm.org/10.1145/2786805.2803183
24. G. Catolino, “Just-in-time bug prediction in mobile applications: The domain matters!” 05 2017.
25. G. Catolino, D. Di Nucci, and F. Ferrucci, “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)*, 2019, pp. 99–110.
26. S. Tabassum, “An investigation of cross-project learning in online just-in-time software defect prediction,” 06 2020.
27. M. Kondo, D. German, O. Mizuno, and E. Choi, “The impact of context metrics on just-in-time defect prediction,” *Empirical Software Engineering*, vol. 25, 01 2020.
28. M. Nayrolles and A. Hamou-Lhadj, “Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects,” 03 2018.
29. “Correlation (pearson, kendall, spearman).” [Online]. Available: https://www.statisticssolutions.com/correlation-pearson-kendall-spearman/
30. C. Tantithamthavorn, A. E. Hassan, and K. Matsumoto, “The impact of class rebalancing techniques on the performance and interpretation of defect prediction models,” *IEEE Transactions on Software Engineering*, vol. 46, no. 11, pp. 1200–1219, 2020.
31. A. Agrawal and T. Menzies, “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*, ser. ICSE ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 1050–1061. [Online]. Available: https://doi.org/10.1145/3180155.3180197
32. “Dmwr.” [Online]. Available: https://www.rdocumentation.org/packages/DMwR/versions/0.4.1/topics/SMOTE
33. H. Li, W. Shang, Y. Zou, and A. E. Hassan, “Towards just-in-time suggestions for log changes (journal-first abstract),” in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2018, pp. 467–467.
34. B. Ghotra, S. McIntosh, and A. E. Hassan, “Revisiting the impact of classification techniques on the performance of defect prediction models,” in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE ’15. IEEE Press, 2015, p. 789–800.
35. S. Davies, M. Roper, and M. Wood, “Comparing text-based and dependence-based approaches for determining the origins of bugs,” *Journal of Software: Evolution and Process*, vol. 26, 01 2014.
36. M. McHugh, “Interrater reliability: The kappa statistic,” *Biochemia medica : časopis Hrvatskoga društva medicinskih biokemičara / HDMB*, vol. 22, pp. 276–82, 10 2012.
37. P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, *Characterizing and Predicting Which Bugs Get Fixed: An Empirical Study of Microsoft Windows.* New York, NY, USA: Association for Computing Machinery, 2010, p. 495–504. [Online]. Available: https://doi.org/10.1145/1806799.1806871
38. T. Fawcett, “An introduction to roc analysis,” *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.
39. J. H. McDonald, *Handbook of biological statistics.* sparky house publishing Baltimore, MD, 2014, vol. 3, pp. 186–189.
40. S. S. Sawilowsky, “New effect size rules of thumb,” *Journal of Modern Applied Statistical Methods*, vol. 8, no. 2, p. 26, 2009.
41. T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, “Studying just-in-time defect prediction using cross-project models,” *Empirical Software Engineering*, vol. 21, 05 2014.
42. A. Nistor, T. Jiang, and L. Tan, “Discovering, reporting, and fixing performance bugs,” in *2013 10th Working Conference on Mining Software Repositories (MSR)*, May 2013, pp. 237–246.

43. C. Williams and J. Spacco, “Szz revisited: Verifying when changes induce fixes,” in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, ser. DEFECTS ’08. New York, NY, USA: ACM, 2008, pp. 32–36.
44. Jpace, “jpace/diffj.” [Online]. Available: https://github.com/jpace/diffj
45. I. Molyneaux, *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*, 1st ed. O’Reilly Media, Inc., 2009.
46. R. E. Bryant and D. R. O’Hallaron, *Computer Systems: A Programmer’s Perspective*, 3rd ed. Pearson, 2015.
47. J. team, “Eclipse java development tools (jdt).” [Online]. Available: https://www.eclipse.org/jdt/
48. C. Guindon, “Swt: The standard widget toolkit.” [Online]. Available: https://www.eclipse.org/swt/
49. G. Rodrıguez-Perez, M. Nagappan, and G. Robles, “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*, May 2020.
50. S. Tsakiltsidis, A. Miranskyy, and E. Mazzawi, “On automatic detection of performance bugs,” in *2016 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2016, pp. 132–139.
51. Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung, “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*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 157–168. [Online]. Available: https://doi.org/10.1145/2950290.2950353
52. J. Chen, W. Shang, and E. Shihab, “Perfjit: Test-level just-in-time prediction for performance regression introducing commits,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.