A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes

Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, Ahmed E. Hassan

Abstract— The approach proposed by Śliwerski, Zimmermann, and Zeller (SZZ) for identifying bug-introducing changes is at the foundation of several research areas within the software engineering discipline. Despite the foundational role of SZZ, little effort has been made to evaluate its results. Such an evaluation is a challenging task because the ground truth is not readily available. By acknowledging such challenges, we propose a framework to evaluate the results of alternative SZZ implementations. The framework evaluates the following criteria: (1) the earliest bug appearance, (2) the future impact of changes, and (3) the realism of bug introduction. We use the proposed framework to evaluate five SZZ implementations using data from ten open source projects. We find that previously proposed improvements to SZZ tend to inflate the number of incorrectly identified bug-introducing changes. We also find that a single bug-introducing change may be blamed for introducing hundreds of future bugs. Furthermore, we find that SZZ implementations report that at least 46% of the bugs are caused by bug-introducing changes that are years apart from one another. Such results suggest that current SZZ implementations still lack mechanisms to accurately identify bug-introducing changes. Our proposed framework provides a systematic mean for evaluating the data that is generated by a given SZZ implementation.

Index Terms—SZZ, Evaluation framework, Bug detection, Software repository mining.

1 Introduction

Software bugs are costly to fix [1]. For instance, a recent study suggests that developers spend approximately half of their time fixing bugs [2]. Hence, reducing the required time and effort to fix bugs is an alluring research problem with plenty of potential for industrial impact.

After a bug has been reported, a key task is to identify the root cause of the bug such that a team can learn from its mistakes. Hence, researchers have developed several approaches to identify prior bug-introducing changes, and to use such knowledge to avoid future bugs [3–10].

A popular approach to identify bug-introducing changes was proposed by Śliwerski, Zimmermann, and Zeller ("SZZ" for short) [9, 11]. The SZZ approach first looks for bug-fixing changes by searching for the recorded bug ID in change logs. Once these bug-fixing changes are identified, SZZ analyzes the lines of code that were changed to fix the bug. Finally, SZZ traces back through the code history to find when the changed code was introduced (*i.e.*, the supposed bug-introducing change(s)).

- *D. da Costa, U. Kulesza, and R. Coelho are with the Department of Informatics and Applied Mathematics (DIMAp), Federal University of Rio Grande do Norte, Brazil.*
  E-mails: danielcosta@ppgsc.ufrn.br,{uira,roberta}@dimap.ufrn.br
- *Shane McIntosh is with the Department of Electrical and Computer Engineering, McGill University, Canada.*
  E-mail: shane.mcintosh@mcgill.ca
- *Weiyi Shang is with the Department of Computer Science and Software Engineering, Concordia University, Canada.*
  E-mail: shang@encs.concordia.ca
- *Ahmed E. Hassan is with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Canada.*
  E-mail: ahmed@cs.queensu.ca

Two lines of prior work highlight the foundational role of SZZ in software engineering (SE) research. The first line includes studies of how bugs are introduced [9, 10, 12–22]. For example, by studying the bug-introducing changes that are identified by SZZ, researchers are able to correlate characteristics of code changes (*e.g.*, time of day that a change is recorded [9]) with the introduction of bugs. The second line of prior work includes studies that leverage the knowledge of prior bug-introducing changes in order to avoid the introduction of such changes in the future. For example, one way to avoid the introduction of bugs is to perform *just-in-time* (JIT) quality assurance, *i.e.*, to build models that predict if a change is likely to be a bug-introducing change before integrating such a change into a project's code base. [6, 8, 23–25].

Despite the foundational role of SZZ, the current evaluations of SZZ-generated data (the indicated bug-introducing changes) are limited. When evaluating the results of SZZ implementations, prior work relies heavily on manual analysis [9, 11, 26, 27]. Since it is infeasible to analyze all of the SZZ results by hand, prior studies select a small sample for analysis. While the prior manual analyses yield valuable insights, the domain experts (*e.g.*, developers or testers) were not consulted. These experts can better judge if the bug-introducing changes that are identified by SZZ correspond to the true cause of the bugs.

Unfortunately, to conduct such an analysis is impractical. For instance, the experts would need to verify a large sample of bug-introducing changes, which is difficult to scale up to the size of modern defect datasets. Additionally, those changes may be weeks, months, or even years old, forcing experts to revisit an older state of the system that they

are unlikely to recall reliably. Making matters worse, the experts with the relevant system knowledge may no longer be accessible due to developer turnover in large software organizations.

Still, evaluating SZZ-generated data is important, since (a) SZZ is used as an experimental component in several studies [9, 10, 12–22, 28] and (b) SZZ is used to detect the origin of bugs in practice [29]. If the SZZ-generated data is not sound, it may taint the conclusions of the analyses that rely upon such data.

To address the challenges associated with evaluating SZZ, we propose an evaluation framework based on three criteria: (1) the *earliest bug appearance*, which factors in the estimation by the development team of when a bug was introduced, based on data that is recorded in the *Issue Tracking System* (ITS, *e.g.*, Bugzilla); (2) the *future impact of a change*, which analyzes the number of future bugs that a given bug-introducing change introduces; and (3) the *realism of bug introduction*, which analyzes if the bug-introducing changes found by SZZ realistically correspond to the actual bug introduction context.

The main goal of our framework is to provide practitioners and researchers with a means for exploring and navigating SZZ-generated data before performing further analyses. For instance, our framework may highlight subsets of SZZ-generated data that are very likely to be inaccurate. Such an assessment of SZZ may also guide future manual analyses that are to be performed upon SZZ-generated data.

While prior work has used SZZ to study the relationship between bug-introducing changes and code clones [18], code ownership [13], fault-prone bug-fixing changes [16], and other factors [12, 14, 15, 17, 19–22, 28], this study evaluates the underlying SZZ approach itself. We perform an analysis of five SZZ implementations. Through an empirical study of ten open source projects, we make the following observations:

1) **Earliest bug appearance:** We find that previously proposed SZZ implementations [27] increase the likelihood of incorrectly flagging a code change as bug-introducing with respect to the earliest bug appearance criteria. For instance, the SZZ implementation that selects only the most recent bug-introducing change has the highest ratio of disagreement (median of 0.43) with the opinion of team members.
2) **Future impact of a change:** We find that the most realistic result among SZZ implementations (*i.e.*, the result that produces the fewest outliers) is that 29% of the bug-introducing changes lead to multiple future bugs that span at least one year. These results suggest that SZZ still lacks mechanisms to accurately flag bug-introducing changes. Indeed, it is unlikely that all 29% of the bug-introducing changes in a project do in fact introduce bugs that took years to be discovered.
3) **Realism of bug introduction:** Another unrealistically result from the evaluated SZZ implementations is that 46% of the bugs are caused by bug-introducing changes that span at least one year. Again, it is unlikely that all 46% of the bugs were in fact caused by code changes that are years apart.

The remainder of this paper is organized as follows: Section 2 introduces the background concepts and discusses related work. Section 3 describes our proposed evaluation framework. Section 4 describes our study settings. Section 5 presents the results of our study, while we derive practical guidelines for using our framework in Section 6. The threats to the validity of our work are described in Section 7. Finally, Section 8 draws conclusions.

2 Background & Related Work

This section describes the concepts that are necessary to understand our study, and surveys the related work.

2.1 Implementations of SZZ

2.1.1 SZZ approach

The SZZ approach was first defined by Śliwerski *et al.* [9]. The main goal of SZZ is to identify the changes that introduce bugs. SZZ begins with a *bug-fixing change*, *i.e.*, a change that is known to have fixed a bug. Nowadays, many projects have adopted a policy of recording the ID of the bug that is being fixed in the change log. In the basic SZZ implementation [9], to ensure that a change is indeed a bug-fixing change, the bug ID that is found in the change-log is checked in the ITS to verify that the ID is truly a bug.

Step 1 in Figure 1 shows the bug-fixing change for bug OPENJPA-68. Step 2 shows the bug fix, which involves updating an `if` condition in the file `PCClassFileTransformer.java`. The `if` condition checks for an incorrect package name, *i.e.*, the string value should be `org/apache/openjpa/enhance/PersistenceCapable` rather than `openjpa/enhance/PersistenceCapable`.

For each identified bug-fixing change, SZZ analyzes the lines of code that were updated. For instance, Step 2 in Figure 1 shows the differences between changes #468455 and #468454 in the `PCClassFileTransformer.java`. In this case, in order to fix the bug, the `if` condition at line 201 was changed.

Thus, to identify the change that introduced bug OPENJPA-68, SZZ traces through the history of the *source configuration management* (SCM) system. The basic SZZ implementation [9] uses the *annotate* function that is provided by most SCM systems to identify the last change that a given line of code underwent prior to the bug-fixing change. Step 3 in Figure 1 shows that change #425473 is flagged as a potential bug-introducing change by SZZ.

Furthermore, if a potential bug-introducing change was recorded after the bug under analysis was reported, it is excluded from further consideration, since it cannot be the bug-introducing change. Henceforth, we refer to the basic SZZ implementation as B-SZZ.

2.1.2 Improvements to B-SZZ

B-SZZ [9] has several limitations. For instance, B-SZZ may flag style changes (*e.g.*, modifications to the code indentation, code comments, and blank lines), as bug-introducing changes. These style changes cannot be the cause of a bug because they do not impact the behaviour of the system.

Instead, Kim *et al.* [11] proposed an SZZ implementation that excludes style changes from the analyses. Furthermore, Kim *et al.* [11] propose the use of the *annotation graph* [30] rather than the *annotate* function because the *annotation*

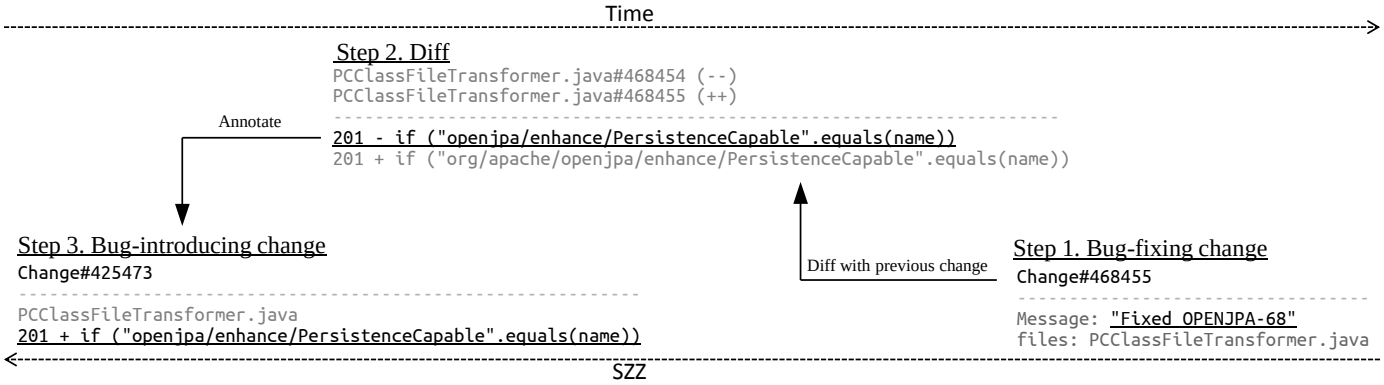


Fig. 1: **Overview of the SZZ approach.** SZZ first looks into a change log to find a bug-fixing change (Step 1). Then, it uses a diff algorithm to localize the exact fix (Step 2). Finally, it traces back to the origin of the modified code (Step 3). The origin of the modified code is a potential bug-introducing change.

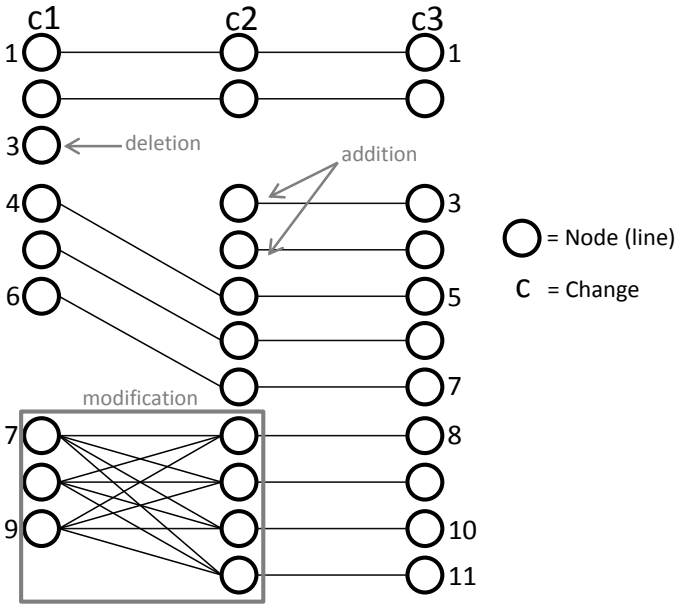


Fig. 2: Overview of the annotation graph.

graph provides traceability for lines that move from one location to another within a file. We refer to the SZZ implementation that is proposed by Kim *et al.* as AG-SZZ (*annotation graph* SZZ). Figure 2 shows an example of how the annotation-graph tracks the evolution of lines across three different changes ($c1$ to $c3$). Every line is a node. Each node has an edge connecting the line to its previous version. For instance, from $c1$ to $c2$, one line was deleted and two lines were added. Line 5 of $c2$ is mapped to line 4 in $c1$. If several lines are modified (*i.e.*, deleted and added as understood by the SCM), the annotation graph conservatively maps every added line in $c2$ as an evolution of every deleted line in $c1$. To find the bug-introducing changes, AG-SZZ performs a *depth-first search* on the annotation graph.

Furthermore, Williams and Spacco propose an enhancement to the line mapping algorithm of SZZ [26, 31]. The enhanced algorithm uses weights to map the evolution of a line. For example, if a given change modified an `if` condition to a different `if` condition and added a new line of code, the annotation graph would indicate that the previous `if` condition changed into these two lines. On the other hand, the enhanced algorithm would place a heavier weight on the edge between the previous and the updated `if` conditions to indicate that the previous `if` condition is more likely to have changed into the updated `if` condition rather than to be a new line of code. Moreover, Williams and Spacco use the DiffJ tool to identify changes at a semantic level rather than a textual one.[1]

2.2 Usage of SZZ

Before the advent of SZZ, researchers were limited to the information that is provided by bug fixes. In order to analyze bug fixes, techniques to link bug-fixing changes to bug reports were proposed [32–35]. This bug fix information enables useful analyses, such as: (1) counting the number of bugs per file [36], (2) bug prediction [37], and (3) finding risky modules within a software system [38].

However, solely with bug-fixing information, one cannot study how bugs are introduced. By detecting bug-introducing changes, SZZ has enabled a wealth of studies that can be broken down into: (1) *empirical research on how bugs are introduced*, and (2) *research that leverages such knowledge to avoid introducing new bugs*.

How bugs are introduced. Śliwerski *et al.* [9] used SZZ to study the size of bug-introducing changes, and to investigate which day of the week bug-introducing changes are more likely to appear. Eyolfson *et al.* [10] used SZZ to correlate bug-introducing changes with the time of the day and the developer experience.

Other studies used SZZ to relate bug-introducing changes with code clones [18], code ownership [13], fault-prone bug-fixing changes [16], and many other factors that may be related to bug introduction [12, 14, 15, 17, 19–22, 28]. Our study aims to evaluate the SZZ approach rather than studying the characteristics of bug-introducing changes that are detected by SZZ.

Just-in-time (JIT) quality assurance. Prior work used SZZ to label changes that are recorded within datasets as bug-introducing or not [6, 8, 23–25, 39]. Such datasets are then

1. http://www.incava.org/projects/java/diffj

used to train models that classify whether a future change will be bug-introducing one or not. For instance, Kim *et al.* [6] used the identifiers within added and deleted code, and the words within change logs to classify changes as bug-introducing or not. Our study evaluates the SZZ-generated data itself rather than using it to train classification models.

In short, a good understanding of the quality of the SZZ data is essential to ensure the validity of any findings that are derived from such data.

2.3 Evaluations of SZZ

The ideal way to evaluate if the bug-introducing changes that are flagged by SZZ are correct would be to compare them to a ground truth dataset that contains each line of code that contributes to the introduction of each bug of the studied projects. This ground truth information could be provided by domain experts, such as developers, since they have the domain knowledge necessary to understand the context of the bug-introducing changes.

Unfortunately, it is impractical to produce such dataset because the domain experts would need to analyze a large amount of historical data in order to pinpoint the changes that introduced bugs. Given the scale of the bug datasets that are commonly analyzed in SE research nowadays, such a task is unrealistic. Furthermore, the domain experts may not recall enough context to identify the cause of bugs that were addressed a long time ago. Moreover, the experts with relevant system knowledge may no longer be available (*e.g.*, s/he stopped contributing to the project).

Given the described challenges, prior research that evaluated SZZ has relied on manual analyses to label bug-introducing changes as true and false positives [11, 26, 27]. These manually labeled bug-introducing changes are then compared to the bug-introducing changes flagged by SZZ. While such manual analyses are important and produce valuable insight, they still have limitations.

First, the domain experts are not consulted due to the aforementioned challenges. Although the research team may thoroughly analyze the possible causes of a bug, their lack of domain and system knowledge introduces bias in the results. For example, the domain experts, when trying to fix a bug, may fail to eliminate the cause of the bug in the first attempt [40]. Therefore, finding the correct bug-introducing changes manually may be more challenging when the domain knowledge is not accessible. Second, due to the effort that is needed to scale up manual analyses, prior evaluations have been performed on small subsamples. For example, Davies *et al.* [27] performed an evaluation of two approaches to detect bug-introducing changes: (1) the *text approach* (*i.e.*, B-SZZ) and (2) the *dependence approach* [41]. The authors defined the ground truth by manually labeling the bug-introducing changes of 174 bugs from the Eclipse[2], Rachota[3], and JEdit[4] projects. Furthermore, to evaluate AG-SZZ, Kim *et al.* [11] compared two sets of bug-introducing changes: the set of bug-introducing changes detected by B-SZZ (S), and the set of bug-introducing changes detected by AG-SZZ (K). The authors assumed that AG-SZZ is more accurate than B-SZZ, and computed the *false positives* ($\frac{|S-K|}{|S|}$) and *false negatives* ($\frac{|K-S|}{|K|}$). Finally, Williams and Spacco [26] proposed a line mapping algorithm to improve the SZZ approach. They measured the improvement by performing a manual analysis of 25 bug-fixing changes.

2.4 Contributions of this paper

This paper addresses several of the limitations of prior evaluations of the SZZ approach. Table 1 provides an overview of the scope of prior evaluations of the SZZ approach. First, the ground truth for our study is composed of a large sample of 32,033 bug-fixing changes from 10 studied projects, whereas prior work analyzes samples of 25-543 bug-fixing changes from 1-3 studied projects. Second, our study compares five SZZ implementations, while prior studies have compared two. Third, our study proposes an evaluation framework that systematically assesses SZZ-generated data. Finally, our framework also factors in the estimates by the development team to perform the evaluation — our work is based on the estimates for 2,637 bug-fixing changes.

We do note that our proposed framework complements the valuable knowledge of domain experts. Since asking experts to produce entire datasets of thousands of bugs is impractical, our framework can be used to flag suspicious entries in automatically generated SZZ data. Moreover, when domain experts are not available to researchers and practitioners, the criteria of our evaluation framework is a practical alternative to spot suspicious entries in SZZ-generated data. For instance, research that uses SZZ to classify code changes as bug-introducing (*e.g.*, *JIT quality assurance* [6, 8, 23–25, 39]) may produce more reliable classification models once they are trained using data that is robust to the false positives that our framework can flag.

3 Evaluation Framework

This section describes our proposed framework to evaluate SZZ. The framework has three evaluation criteria: (1) earliest bug appearance, (2) future impact of changes, and (3) realism of bug introduction. We explain the rationale behind each proposed criterion, *i.e.*, why it may help in the detection of false positives in SZZ-generated data.

3.1 Earliest Bug Appearance

The *earliest bug appearance* criterion is concerned with how much SZZ disagrees with the estimates given by the development team. To quantify the disagreement between an SZZ implementation and such estimates, we compute the *disagreement ratio*, which relies on bug report data that we collect from the JIRA ITS.[5] The JIRA ITS is a system developed by Atlassian, which provides issue tracking and project management functionality.

What makes JIRA an important asset to our work is the *affected-version* field, which allows teams to record the versions of the system that a given bug impacts. For example, the HADOOP-1623 issue report states that versions 0.12.3 and 0.13.0 are impacted.[6]

2. https://www.eclipse.org/ (April 2016)
3. http://rachota.sourceforge.net/en/index.html (April 2016)
4. http://jedit.org/ (April 2016)
5. https://www.atlassian.com/software/jira (April 2016)
6. https://issues.apache.org/jira/browse/HADOOP-1623 (April 2016)

TABLE 1: Summary of the evaluations of the SZZ approach as performed by prior work.

	Year	#fixes	Subject projects	Accuracy Measures
Davies [27]	2014	543	Eclipse Rachota, and JEdit	The predictions of bug-introducing changes made by the simulation of B-SZZ were compared to the manually classified bug-introducing changes. In this way, the authors measured the true positives, false positives, and false negatives. Precision and recall were also computed.
Williams [26]	2008	25	Eclipse	No definition of accuracy measures. The research team judged if the bug-introducing changes flagged by SZZ were likely to be correct.
Kim [11]	2006	301	Columba and Eclipse	The authors considered the predictions of the AG-SZZ as the ground truth. By comparing the predictions of AG-SZZ and B-SZZ, the authors computed the false positives and false negatives that are produced by B-SZZ.
This paper	2016	2,637*-32,033	HBase, Hadoop Common, Derby, Geronimo, Camel, Tuscany, OpenJPA, ActiveMQ, Pig, and Mahout	We use the measures of our proposed framework (Section 3) to evaluate five SZZ implementations.

* Only 2,637 bug-fixes can be analyzed by the earliest bug appearance criterion. The other criteria can analyze all 32,033 bug-fixes.

To compute the disagreement ratio, we count the number of *disagreements* that SZZ has when compared to the earliest affected-version. To compute the disagreements, we first identify the earliest affected-version. For example, if a team member indicates that Bug-01 impacts versions 1.5 and 2.0, we select version 1.5 as the earliest version. We use the earliest affected-version because SZZ aims to find when a bug was actually introduced. Hence, SZZ's results have to be compared with the first known version in which a bug is known to appear.

We classify a potential bug-introducing change that is flagged by SZZ as *incorrect* (according to the earliest affected version) if the change was recorded after the release date of the earliest affected-version. On the other hand, if SZZ flags a bug-introducing change that was recorded before the release date of the earliest affected-version, such a change may or may not be the actual cause of the bug. As we do not have enough information about which version such bug-introducing changes were integrated into, we classify those bug-introducing changes as *unknown*.

We count a bug as a disagreement if all of the potential bug-introducing changes that are flagged by SZZ for that bug are classified as incorrect. Furthermore, a change could not lead to a bug if it was performed after the bug has been reported. Hence, if all of the potential bug-introducing changes for a given bug are recorded after the bug report date, we consider it to be a disagreement. Our proposed disagreement ratio is a lower-bound metric to verify how much a particular SZZ implementation disagrees with the estimates that are provided by team members.

After counting disagreements, the disagreement ratio R for a studied project S is: $R(S) = \frac{D(S)}{B(S)}$, where D is the number of disagreements in S, and B is the total number of bugs in S. The disagreement ratio R ranges from 0 to 1.

PIG-204 is an example of how the number of disagreements can indicate problems in the studied SZZ implementations. PIG-204 has the highest number of disagreements in the SZZ-generated data for the Pig project. MA-SZZ flags changes 617338 and 644033 as potentially bug-introducing for PIG-204. These changes refer to file/directory renaming changes, *i.e.*, they are not the original changes that introduced the buggy code. Such problems happen because Subversion does not track the history of renamed files, so SZZ cannot trace back further in the history to find the original bug-introducing change.

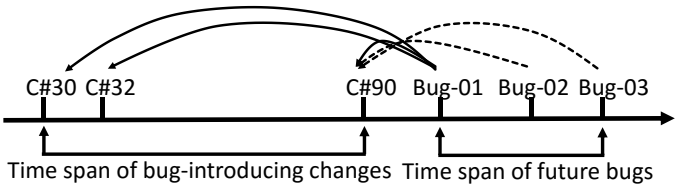


Fig. 3: **The *time-span of bug-introducing changes* and *time-span of future bugs* measures**. The dashed lines indicate relationships that are only used in the *time-span of future bugs* computation.

3.2 Future Impact of Changes

The *future impact of changes* criterion is concerned with the impact that a bug-introducing change has upon future bugs. This criterion has two associated metrics: (1) *count of future bugs* and (2) *time-span of future bugs*.

Figure 3 shows how the count of future bugs is computed. For example, when SZZ analyzes the fix(es) of Bug-01, it flags change #90 as a potential bug-introducing change. However, it also happens that change #90 is flagged as a bug-introducing change for Bug-02 and Bug-03, which means that the count of future bugs for change #90 is three.

We suspect that a change with a high count of future bugs (*e.g.*, hundreds of future bugs) indicates either that (a) the change was highly problematic or (b) the SZZ-generated data is not correct. For instance, let us suppose that change #90 is flagged as a potential bug-introducing change in the fixes of more than 100 future bugs. It is unlikely that a single change could have truly introduced so many bugs. Indeed, manually verified SZZ-generated data from prior work (Williams and Spacco [26]) reveals that 93% of the changes have a relatively low count of 1 to 3 future bugs. On the other hand, potential bug-introducing changes that are flagged by SZZ in the fixes of many bugs could be foundational changes that introduce core functionality, which has to be changed frequently while performing bug

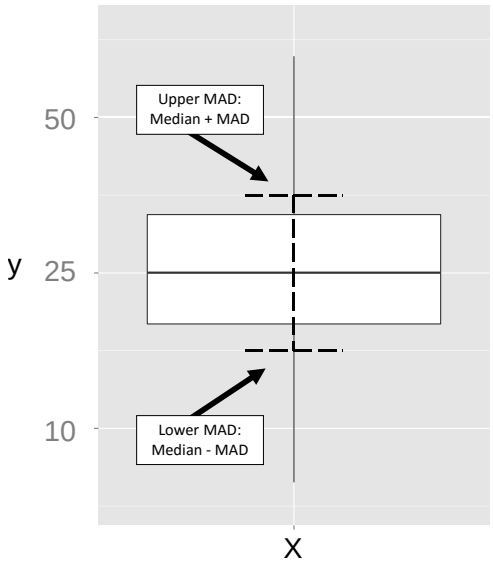


Fig. 4: **The upper and lower MADs.** The *upper MAD* is the sum of the median value and the MAD, while *lower MAD* is the difference between the median value and the MAD

fixing activities [42, 43]. Nevertheless, we expect the number of such bug-introducing changes to be rather low.

The second metric that we use to quantify the future impact of changes is the *time-span of future bugs*. This metric counts the number of days between the first and last future bugs of a potential bug-introducing change. For example, the time-span of future bugs for change #90 in Figure 3 is the number of days between Bug-01 and Bug-03.

In case that a bug-introducing change leads to several bugs in the future, we suspect that the other future bugs are unlikely to be discovered *several years* after the first discovered bug [44] (*e.g.*, a time-span above the upper *Median Absolute Deviation* of the SZZ-generated data). On the other hand, prior research has acknowledged the existence of dormant bugs, *i.e.*, bugs that are reported much later after they were actually introduced [45]. Nevertheless, although dormant bugs may inflate the time-span of future bugs (since their report date is much later after they were introduced), they may account for 33% of the bug data in prior research [45]. Therefore, if a change leads to future bugs with a time-span of *several years* it can either indicate that (a) the last bug could be dormant for a long time or (b) the SZZ-generated data is not correct.

We consider bug-introducing changes that have a time-span of future bugs above the *upper Median Absolute Deviation (MAD)* to be suspicious. MAD is a robust statistic to analyze deviations with respect to the median [46]. We use the MAD statistic instead of other measures of dispersion (*e.g.*, standard deviation), since prior work finds that MAD is more effective at identifying outliers [47], especially in data that does not follow a normal distribution [46]. Figure 4 shows how we use MAD to analyze our results. The *upper MAD* is the sum of the median value of the data and the MAD, while the *lower MAD* is the difference between the median value and the MAD.

Change #355543 has one of the highest counts of future bugs (74 future bugs) in the SZZ-generated data for the ActiveMQ project. Change #355543 refers to the initial code import of the ActiveMQ 4.x code-base in the Subversion SCM. However, ActiveMQ was originally hosted using CVS before, which means that SZZ could trace back further in the history if it had access to the CVS data.[7] The high count of future bugs may be indicating that change #355543 is a roadblock for many bugs that should be traced back further in the history.

3.3 Realism of bug introduction

Previous research has recognized 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) [11, 27]. The aim of the *realism of bug introduction* criterion is to evaluate the likelihood that all of the bug-introducing changes that are flagged by SZZ are indeed the actual cause of a given bug.

We use the *time-span of bug-introducing changes* to quantify the realism of bug introduction. For a given bug, this metric counts the number of days between the first and last potential bug-introducing changes that are flagged by SZZ. Figure 3 shows an example of how the time-span of bug-introducing changes metric is calculated. SZZ flags changes #30, #32, and #90 as the potential bug-introducing changes of Bug-01. To compute the time-span of bug-introducing changes, we count the days between change #30 (the first change), and change #90 (the last change).

We suspect that the time-span between bug-introducing changes should be short (*e.g.*, below the upper MAD). Indeed, prior work suggests that bugs are unlikely to be noticed years after their introduction [44]. Therefore, we suspect that a very long time-span (*e.g.*, several years) is unlikely to represent the true sequence of changes that introduced a particular bug. For example, the potentially bug-introducing changes that are flagged by SZZ for GERONIMO-6370 span 2,684 days (*i.e.*, the time-span between changes #153289 and #1350831). However, the most recent change (#1350831) is actually a file/directory renaming change. If SZZ was able to trace back further to find the original bug-introducing change, the time-span between the potential bug-introducing changes would likely be more realistic (*e.g.*, below the upper MAD).

The proposed framework is stronger when the three criteria are considered in tandem with each other. While a naïve SZZ implementation may perform well in terms of one criterion, it will likely suffer in terms of the other two criteria.

4 Study Settings

This section describes our study settings: (1) the different SZZ implementations that we evaluate, (2) the studied projects, and (3) the study procedures.

4.1 Evaluated SZZ implementations

In this work, we evaluate five SZZ implementations. Table 2 shows the *mapping* and *selection* mechanisms of each SZZ implementation. The *mapping* mechanism is the strategy used to link bug-fixing changes to potential bug-introducing

7. http://activemq.apache.org/cvs.html (April 2016)

TABLE 2: **The mapping and selection mechanisms of the studied SZZ implementations.** In addition to the selection mechanisms described directly in each row, the selection mechanisms of the prior rows that have the † symbol are also inherited. For example, L-SZZ inherits all of the previous selection mechanisms except the one from R-SZZ. Finally, the ‡ symbol indicates that all of the potential bug-introducing changes are returned by that SZZ implementation

	Description	Mapping Mechanism	Selection Mechanism
B-SZZ	First SZZ implementation proposed by Śliwerski *et al.* [9].	The *annotate* function is used to prepend the last change that modified each line of code within a file in a given change. Next, each line of code is scanned in order to identify the last change that modified the lines that were involved in the bug-fixing change. Such changes are potential bug-introducing changes.	††Potential bug-introducing changes that are dated after the bug report date are removed.
AG-SZZ	B-SZZ improvement proposed by Kim *et al.* [11].	The annotation-graph is used to represent evolution of each line of code within source files. A depth-first search of the annotation-graph is used to find the potential bug-introducing changes.	††Changes such as comments, format changes, blank lines, and code movement are not flagged as potential bug-introducing changes.
MA-SZZ	It is built on top of the AG-SZZ, but it is aware of meta-changes. This implementation is proposed in this paper.		††Potential bug-introducing changes that are meta-changes are removed.
R-SZZ	B-SZZ improvement proposed by Davies *et al.* [27]. We build R-SZZ on top of MA-SZZ in this paper.		The latest potential bug-introducing change is indicated as bug-introducing.
L-SZZ	B-SZZ improvement proposed by Davies *et al.* [27]. We build L-SZZ on top of MA-SZZ in this paper.		The largest potential bug-introducing change is indicated as bug-introducing.

changes. The *selection* mechanism filters away changes that are unlikely to be bug-introducing changes. For example, AG-SZZ uses the annotation graph to implement the mapping mechanism.

After prototyping B-SZZ and AG-SZZ, we noticed that meta-changes (*i.e.*, changes that are not related to source code modification) were being reported as bug-introducing. We observe three types of potential bug-introducing meta-changes: (1) branch-changes, (2) merge-changes, and (3) property-changes. A branch change is a meta-change that copies the project state from one SCM branch to another (*e.g.*, from the trunk to a feature branch). A merge-change is a meta-change that applies change activity from one branch to another (*e.g.*, from a feature branch to the trunk). Finally, a property-change is a meta-change that only impacts file properties that are stored in the SCM (*e.g.*, end of line property). Ideally, meta-changes should not be flagged as bug-introducing, since they do not change system behaviour, and hence, cannot introduce bugs.

In order to address the meta-change problem, we implement an enhancement to the previous annotation graph that is used by AG-SZZ [48]. The original annotation graph relies on *hunks* to build the nodes and edges. A *hunk* denotes a contiguous set of changed lines. The differences between two versions of a file may contain more than one *hunk*. For instance, when two changes are compared by the *svn diff* command, the output is a set of *hunks* that shows the differences between two changes.

In the original annotation graph algorithm [48], if there are no hunks between the changes, the edges between nodes are not built. However, when comparing a meta-change with its previous change (*i.e.*, origin of the meta-change), no hunks are produced, and no edges are built. For example, let us consider that $c3$ in Figure 2 is a copy of $c2$ on a different branch. The original definition of the annotation graph algorithm [48] would not map the nodes from $c3$ to $c2$, *i.e.*, the `diff` command would produce no hunks, so no edges would be built. Hence, AG-SZZ could not find the actual bug-introducing change (which is $c2$) when performing the depth-first search of the annotation graph.

Our enhancement consists of linking all of the nodes of a given meta-change to its previous version (*i.e.*, edges from $c3$ to $c2$ are built). Our enhancement helps us avoid reporting meta-changes as bug-introducing changes. Other approaches could also be used to avoid the flagging of meta-changes by SZZ. For example, the *history slice* proposed by Servant *et al.* [49] would also allow SZZ implementations to avoid meta-changes. The history slice approach uses a *history graph* rather than an annotation graph. The major difference between a history graph and an annotation graph is that the annotation graph uses a *hunk granularity* when linking nodes while the history graph uses a line granularity (*i.e.*, a line can only be linked to one line). We refer to the enhanced version of AG-SZZ as MA-SZZ (*meta-change aware SZZ*). After implementing MA-SZZ, we further improve it using two selection mechanisms that are proposed by Davies *et al.* [27] (*i.e.*, R-SZZ and L-SZZ).

4.2 Illustrative Example of the Studied SZZ implementations

Figure 5 provides an example that we use to illustrate the differences among the five studied SZZ implementations. The example consists of code that checks if a particular customer is old enough to watch a given movie and if the customer qualifies for a discount. A bug-fix is performed by change #4, where the conditional that checks the age of the customer and the discount price are fixed.

In this example, B-SZZ traces the history of all of the lines that are removed by change #4, including the "`// check loyalty card`" comment. B-SZZ would generate a false positive by flagging line 2 of change #2 as potentially bug-introducing. On the other hand, AG-SZZ would ignore the Java comment. However, AG-SZZ would incorrectly flag change #3 (*i.e.*, a meta-change) as potentially

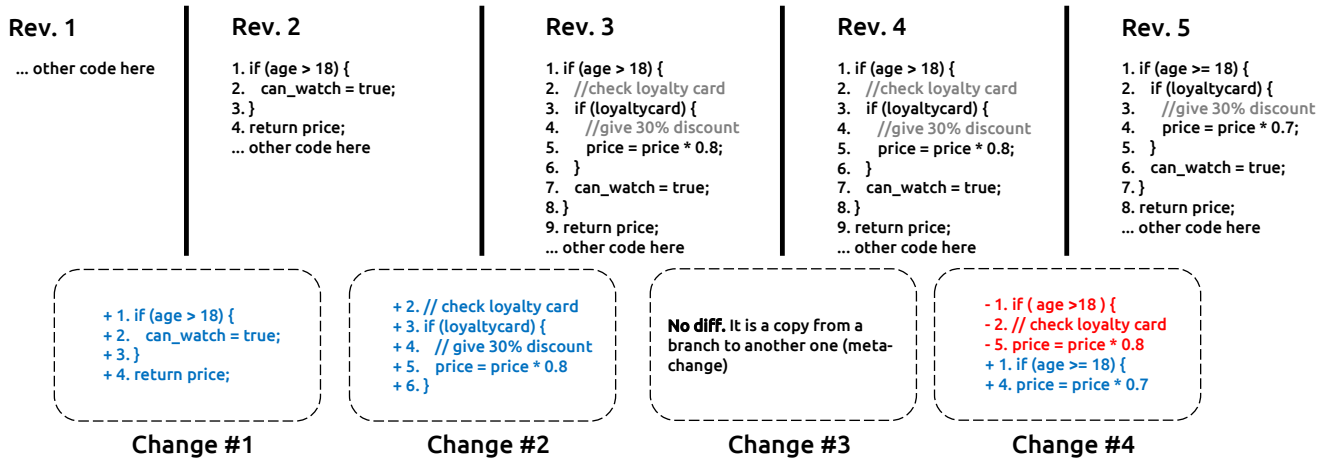


Fig. 5: **An overview of the studied SZZ implementations.** The boxes contain the differences between revisions.

bug-introducing, generating another false positive. MA-SZZ would ignore Java comments and meta-changes. Thus, MA-SZZ would correctly flag line 1 of change #1 and line 5 of change #2 as potentially bug-introducing. L-SZZ would only select change #1 from the potential bug-introducing changes, since it is the largest code change (comments are ignored). On the other hand, R-SZZ would select change #2, since it is the most recent change of the potential bug-introducing changes.

4.3 Studied Projects

In this paper, we study 10 Apache projects that are managed using the JIRA ITS. We considered the following criteria to select our studied projects: (1) the number of linked changes, *i.e.*, bug-fixing changes that can be linked to bug reports and (2) the proportion of bugs with the *affected-version* field filled in.

Table 3 provides an overview of the studied projects. For each studied projects, we evaluate ten years of historical data from August 2003 to September 2013.

4.4 Study Procedures

Figure 6 provides an overview of the steps that are involved in our study. In Step 1, we collect raw data from each studied project. This raw data includes the bug reports from the JIRA ITS and the source code changes from the Subversion SCM. Next, in Step 2, we link the bug reports to the bug-fixing changes based on the approaches that are proposed by Śliwerski *et al.* [9] and Eyolfson *et al.* [10]. These approaches parse potential bug IDs within change logs and verify whether such bug IDs really exist in the JIRA ITS. After obtaining the linked bug data, in Step 3, we select the subset of linked bugs that have the *affected-version* field filled in (as shown in Table 3). In Step 4, we execute the different SZZ implementations and obtain the SZZ-generated data (see Table 2). In Step 5, we compute metrics using the SZZ-generated data. Finally, we perform a manual analysis of the obtained results. The goal of our manual analysis is to investigate if the proposed metrics help identify subsets of the SZZ-generated data that are suspicious.

Our manual analysis consists of two steps. First, we analyze items (*i.e.*, bugs and bug-introducing changes), of the upper extremes of the SZZ-generated data produced by each SZZ implementation. For instance, we analyze the data with the highest *number of disagreements*, largest *time-span of bug-introducing changes*, and highest *count of future bugs*. Second, we analyze the lower extremes of the SZZ-generated data produced by each SZZ implementation. Finally, we compare the obtained results for the upper and lower extremes. We suspect that the upper extremes should contain most of the suspicious SZZ-generated data (see Section 3), while the lower extremes should contain sound data (*i.e.*, they are unlikely to contain false positives). The upper extremes may vary among SZZ implementations. For example, as B-SZZ is prone to flag cosmetic changes as bug-introducing, it may have different items in the upper extreme of *time-span of bug-introducing changes* than another SZZ implementation that ignores cosmetic changes.

We exclude R-SZZ and L-SZZ from our manual analysis for the following reasons: (i) they do not generate data for the *time-span of bug-introducing changes* metric (*i.e.*, they select only one bug-introducing change) and (ii) we already gain perspective about the R-SZZ and L-SZZ generated data as their *selection mechanisms* are applied to MA-SZZ, which we manually analyze. In total, we analyze 240 items in our manual analyses: 60 items for each analyzed SZZ implementation (20 items for each of the three analyzed metrics) as well as an additional 60 items for the lower extreme of the MA-SZZ generated data (which we discuss in more details in Section 5.4).

Additionally, we analyze the data from a prior study by Williams and Spacco [26] (who gratefully shared their data). We compute the *count of future bugs* metric using their data. Unfortunately, since affected-versions information is not available, we cannot compute the *disagreement ratio* metric. We also cannot compute the *time-span of bug-introducing changes* metric because the dates of the changes were not available in the shared data. Finally, we study the number of false positives that our criteria can spot within the shared data.

5 Study Results

This section presents the results of our analysis of the five studied SZZ implementations (see Table 2) using the ten

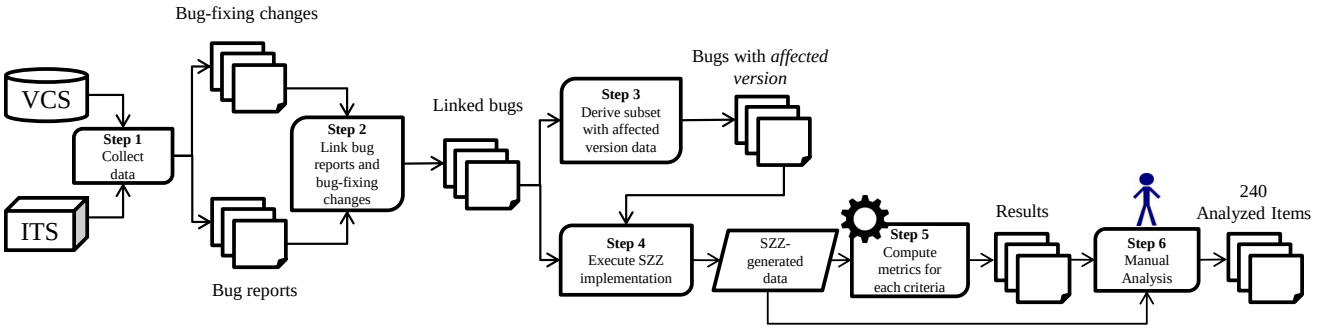


Fig. 6: Overview of the steps involved in our study.

TABLE 3: Overview of the studied Apache projects.

Project	Description	Changes that can be linked to bug reports	Bugs with affected-version
HBase	HBase is the Hadoop database — a distributed, scalable, big data store.	6,507	168
Hadoop Common	Hadoop processes large data sets using clusters of computers.	5,337	201
Derby	Derby is an open source relational database implemented in Java.	2,228	220
Geronimo	Geronimo is an open source application server compatible with the Java Enterprise Edition (Java EE) 6 specification.	4,404	123
Camel	Camel is a tool to build routing and mediation rules in a variety of domain-specific languages.	3,715	141
Tuscany	Tuscany provides the infrastructure for easily developing and running applications using a service oriented approach.	2,838	94
OpenJPA	OpenJPA is an implementation of the Java Persistence API.	2,454	72
ActiveMQ	ActiveMQ is a message broker with a full Java Message Service (JMS) client.	2,229	142
Pig	Pig is a high level platform for creating MapReduce programs with Hadoop.	1,953	81
Mahout	Mahout provides implementations of distributed or scalable machine learning algorithms.	368	26
Totals		32,033	1,268

studied projects. The results are presented for each criterion of the proposed framework. Table 4 provides an overview of the obtained results for each SZZ implementation with respect to each criterion.

5.1 Earliest Bug Appearance

Unfortunately, the *affected-version* field is not mandatory, and is left empty for some bugs in the studied projects. Therefore, we consider only the linked bugs with a filled in *affected-version* field when computing the *disagreement ratio*.

Table 3 shows that we analyze 2,637 bug-fixing changes that are linked to 1,268 unique bugs with the affected-version field filled in. Despite analyzing only a subset of the bugs of the studied projects, the number of bug-fixing changes that we analyze in this paper still exceeds the manual-analyzed changes of prior work (see Table 1).

The B-SZZ implementation has the lowest disagreement ratio (0%-9%). Table 5 shows the results of the disagreement ratio for each SZZ implementation. We find that B-SZZ has the lowest disagreement ratio in general (0%-9%), followed by the MA-SZZ (0%-17%). The R-SZZ implementation has the highest disagreement ratio (24%-50%) followed by the L-SZZ (6%-29%).

Interestingly, for the Mahout project, both the B-SZZ and MA-SZZ implementations have a disagreement ratio of 0%. This indicates that these SZZ implementations did not identify any bug-introducing changes after the earliest affected-versions. Still, the AG-SZZ has a disagreement ratio of 14% for the Mahout project. The only difference between MA-SZZ and AG-SZZ is that MA-SZZ is aware of meta-changes. Indeed, our manual analysis of the 20 bugs with the largest disagreements reveals that AG-SZZ flagged meta-changes as potential bug-introducing in all of the 20 bugs.

Although B-SZZ has the best result in the earliest bug appearance criterion, it is not enough to convey that the B-SZZ-generated data is better than the others. In fact, the 20 manually analyzed bugs related to B-SZZ flag cosmetic changes, blank lines and comments as bug-introducing changes.

Furthermore, B-SZZ is more likely to flag changes before the earliest affected-version because it flags more potentially bug-introducing changes per bug than the other studied SZZ implementations. We perform a Kruskall Wallis test to compare the distributions of bug-introducing changes per bug that are flagged by B-SZZ, AG-SZZ, and MA-SZZ. We omit L-SZZ and R-SZZ, since they select only one bug-introducing change per bug. We obtain a p-value of $2.2e^{-16}$, which indicates that the distributions are indeed statistically different. We also compute the Cliff's delta effect-size measure [50] to quantify the magnitude of the difference in bug-introducing change counts of each studied SZZ implementation. In fact, we obtain a $small$ effect-size ($delta = 0.17$) when comparing B-SZZ and AG-SZZ, indicating that B-SZZ is more likely to have a higher number of bug-introducing changes.

When considering only the SZZ implementations that ignore cosmetic changes, blank lines, and code comments (*i.e.*, all the evaluated implementations except B-SZZ), MA-SZZ is the one that has the lowest disagreement ratio. Moreover, it is not surprising that L-SZZ and R-SZZ have

TABLE 4: **Summary of the obtained results.** The obtained results are shown for each SZZ implementation with respect to each criterion of our proposed framework.

	Earliest Bug Appearance	**Future Impact of a Change**		**Realism of Bug Introduction**	**Meta-changes**
Summary	Evaluates the extent to which an SZZ implementation disagrees with the team members of a project	Evaluates the number of future bugs that a given bug-introducing change leads to		Evaluates the likelihood that the identified bug-introducing changes reflect the actual cause of the bug	Changes that are not the actual cause of bugs (*i.e.*, branch-changes, merge-changes, and property-changes)
Metric	Disagreement Ratio	Count of Future Bugs (% of multiple future bugs)	Timespan of Future Bugs (median of days)	Timespan of Bug-introducing Changes (median of days)	Does it flag meta-changes as potential bug-introducing changes?
B-SZZ	0%-9%	45%	451	429	Yes
MA-SZZ	0%-17%	38%	368	316	No
L-SZZ	6%-29%	26%	273	Not applied	No
R-SZZ	24%-50%	20%	162	Not applied	No

TABLE 5: The *disagreement ratio* for the SZZ implementations.

	B-SZZ	AG-SZZ	MA-SZZ	R-SZZ	L-SZZ
ActiveMQ	0.07	0.16	0.11	0.51	0.24
Camel	0.01	0.09	0.05	0.42	0.16
Derby	0.09	0.17	0.08	0.47	0.15
Geronimo	0.02	0.07	0.04	0.28	0.13
Hadoop Common	0.07	0.26	0.17	0.46	0.28
HBase	0.02	0.12	0.07	0.50	0.24
Mahout	0	0.14	0	0.36	0.09
OpenJPA	0.09	0.22	0.16	0.44	0.28
Pig	0.02	0.14	0.04	0.32	0.06
Tuscany	0.06	0.13	0.04	0.31	0.13

the highest disagreement ratios, since they are less likely to have a bug-introducing change before the earliest affected-version date — these approaches select one bug-introducing change per bug. Additionally, R-SZZ is more likely to have the highest disagreement ratios of both, since it selects the latest potential bug-introducing change. Previous research that assessed L-SZZ and R-SZZ selection mechanisms reported an improvement of precision [27], whereas in our work, we find that L-SZZ and R-SZZ inflate the number of incorrectly flagged bug-introducing changes according to the earliest affected-versions.

AG-SZZ flags at least one meta-change as bug-introducing for 90%-98% of the bugs. On the other hand, B-SZZ flags at least one meta-change as bug-introducing for 0%-48% of the bugs. For example, for the OpenJPA and Derby projects, B-SZZ flags 19% and 48% meta-changes as bug-introducing, respectively. The only type of meta-change that B-SZZ incorrectly flagged as bug-introducing was the *property-change* type, since the built-in `annotate` function of SZZ recognizes changes in the SCM that are copies of others (*e.g.*, a copy from one branch to another or the synchronization of a merge).

Moreover, manual analysis of the 20 bugs that AG-SZZ disagreed with earliest affected-versions reveals that AG-SZZ flagged meta-changes as bug-introducing for all of these 20 bugs.

SZZ still needs improvements to accurately identify bug-introducing changes. AG-SZZ flags meta-changes as bug-introducing for 90%-98% of the bugs. Moreover, L-SZZ and R-SZZ, which are more precise according to previous research, have a median disagreement ratio of 17% and 38%, respectively.

TABLE 6: Measures of the *future impact of changes* criteria. The highest and lowest values are in bold.

	B-SZZ	MA-SZZ	L-SZZ	R-SZZ
%MFB	**45**%	38%	26%	**20**%
MD	**451**	368	273	**162**
UMAD	**981**	809	611	371
%AUMAD	**25**%	27%	26%	**29**%

Legend

%MFB	Percentage of multiple future bug
MD	Median in days of the time-span of future bugs
UMAD	Upper median absolute deviation (days)
%AUMAD	Percentage of samples above UMAD

5.2 Future Impact of Changes

This section presents the results of the two measures that are associated with the *future impact of changes* criterion. Since AG-SZZ is prone to flagging meta-changes as bug-introducing (see Section 5.1), we do not consider it for further analysis in the remaining two criteria.

R-SZZ has the lowest proportion of bug-introducing changes that lead to multiple future bugs (20%). Table 6 shows the proportions of bug-introducing changes that lead to multiple future bugs for each SZZ implementation. B-SZZ has the highest proportion of multiple future bugs (45%) followed by MA-SZZ (38%). On the other hand R-SZZ and L-SZZ have the lowest proportions (20% and 26%, respectively).

B-SZZ is more likely to have a large proportion of changes that lead to multiple bugs because it identifies more bug-introducing changes per bug (see Section5.1). Such behaviour may happen because B-SZZ flags cosmetic modifications, comments, and blank lines as bug-introducing. Therefore, these bug-introducing changes are more likely to overlap when B-SZZ is applied to find the cause of future bugs. Indeed, manual analysis of 20 bug-introducing changes with the highest future bugs counts for B-SZZ reveals that seven changes are cosmetic modifications only (*e.g.*, checkstyle and indentation changes).

Moreover, L-SZZ and R-SZZ are more likely to have the lowest proportions of multiple future bugs because they select only one bug-introducing change per bug. As L-SZZ selects the bug-introducing changes with the most lines of code, it is more likely that one of these lines is changed or deleted in a bug fix in the future. Such a change, if selected by L-SZZ, is more likely to lead to multiple future bugs when compared to R-SZZ.

R-SZZ has the shortest time-span of future bugs (162 days). Figure 7 shows the time-span of future bugs in days. The

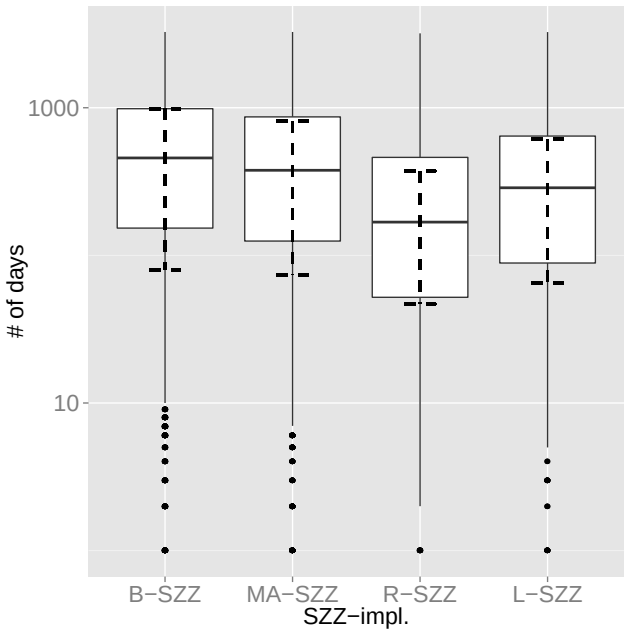
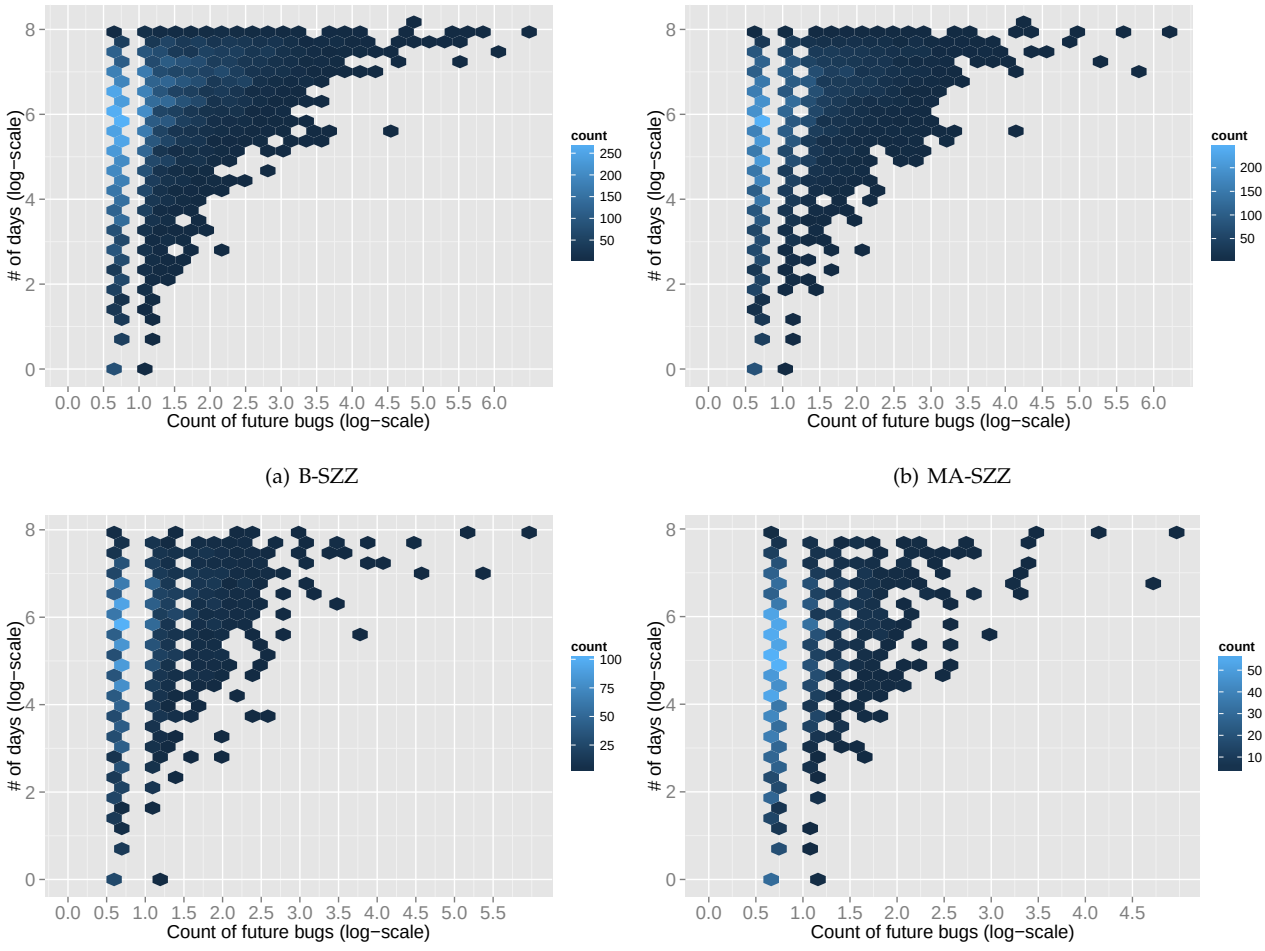


Fig. 7: The time-span of future bugs for each SZZ implementation with upper and lower Absolute Median Deviation (MAD).



(a) B-SZZ

(b) MA-SZZ

(c) L-SZZ

(d) R-SZZ

Fig. 8: The relationship between the count of future bugs (x-axis) and the time-span of future bugs (y-axis) measures

lines in the middle of the boxes are the medians, and the upper and lower dashed lines are the upper and lower MADs, respectively. B-SZZ has the largest time-span of future bugs (451 days in the median) with an upper MAD of 981 days. On the other hand, R-SZZ has the shortest time-span of future bugs (162 days in the median).

Although it is not surprising that R-SZZ has the best results, the results are still relatively poor. For instance, 29% of the bug-introducing changes that are flagged by R-SZZ have a time-span of future bugs above the upper MAD (427 days as shown in Table 6). Furthermore, if we consider MA-SZZ, the time-spans become even higher (27% above 745 days). Such results suggest that the evaluated SZZ implementations still lack mechanisms to accurately flag bug-introducing changes. For example, considering R-SZZ, it is higly unlikely that all 29% of the bug-introducing changes indeed lead to future bugs that span over 427 days.

Finally, in Figure 8 we use hexbin plots to show the relationship between the two measures of the *future impact of changes* criteria. We present the data for bug-introducing changes that have multiple future bugs, *i.e.*, that lead to more than one future bug. The paler the shade of a hexagon, the more instances that fall within that hexagon. We observe that the time-span of future bugs does not always tend to increase as the count of future bugs increases (median spearman correlation of 0.39). The majority of the analyzed changes have two future bugs.

The best performing SZZ implementation still flags 29% of bug-introducing changes with a time-span of future bugs of over 427 days. Such results suggest that current state-of-the-art SZZ implementations still need improvements to accurately flag bug-introducing changes.

5.3 Realism of Bug Introduction

This section describes the results of the *realism of bug introduction* criterion. As discussed in Section 5.2, we do not consider AG-SZZ for this analysis because it is prone to flagging meta-changes as bug-introducing changes. Furthermore, since we analyze the time-span of bug-introducing changes, we can only analyze bugs with more than one bug-introducing change. Hence, we also omit the results for L-SZZ and R-SZZ, because these two implementations select only one bug-introducing change per bug fix.

The bugs that are analyzed by MA-SZZ have the shortest time-span of bug-introducing changes (316 days). Figure 9 shows the time-span of bug-introducing changes in days (y axis). B-SZZ has the longest time-span of bug-introducing changes (429 days in the median). On the other hand, the median for MA-SZZ is 316 days. Furthermore, the upper MAD for MA-SZZ and B-SZZ are 710 and 916 days, respectively. Even for the best result among the studied SZZ implementations (316 days median), such a result is still relatively poor. For example, it is unlikely that bugs take almost a year (316 days) to be detected on average (median).

We also analyze how many bug-introducing changes span at least one year. MA-SZZ and B-SZZ have 46% and 65% of bug-introducing changes that span at least one year. Such results suggest that the evaluated SZZ implementations still need improvements. Indeed, it is unlikely that 46% of the bugs are introduced by changes that span at least

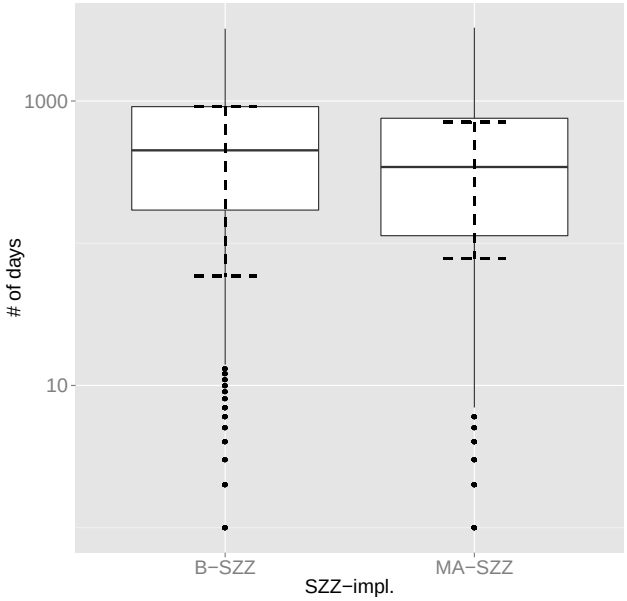


Fig. 9: Time-span of bug-introducing changes that are identified by SZZ with the upper and lower MAD.

one year. Furthermore, for 5 out of the 20 manually analyzed bugs with the highest time-span of bug-introducing changes, MA-SZZ flags changes that are the initial import commits of code into the repository. Since these initial commits occur early in the project history, flagging them as potential bug-introducing changes tends to inflate the *time-span of bug-introducing changes*.

We highlight that although B-SZZ performs better in terms of the *earliest bug appearance* criterion, it has a higher ratio of multiple future bugs (and future bug time-spans) and a larger time-span of bug-introducing changes. These results highlight the importance of considering the criteria (and associated metrics) of our framework in tandem: while a more naïve SZZ implementation may have strong performance along one criterion, it may perform poorly according to the other criteria.

We find that for 46% and 65% of bugs, MA-SZZ and B-SZZ flag potential bug-introducing changes that span at least one year. These results suggest that state of the art SZZ implementations still need improvement to accurately flag bug-introducing changes.

5.4 Manual Analysis

We perform a manual analysis of 240 items (160 bugs as a whole and 80 bug-introducing changes) as explained in Section 4.4. We follow *open coding* from Grounded Theory [51] to group the manually analyzed items into categories of changes that are unlikely to be truly bug-introducing. The categories are the following: (i) missing bug-introducing changes, (ii) initial code import, (iii) directory/file renaming changes, (iv) semantically equivalent changes, (v) backout changes, and (vi) general changes with low likelihood of being bug-introducing. The changes within categories (i)-(v) are easier to identify, since they do not require a deep understanding of the code of the studied projects. On the other hand, category (vi) requires a careful code reading

TABLE 7: Changes found during our manual analysis that highlight possible improvement directions for SZZ implementations.

	Missed	Initial code importing	Directory/File renaming	Semantically equivalent	Backout	Low likelihood	True positive
Disagreement ratio	5	0	3	1	0	2	9
Count of future bugs	0	5	4	5	0	0	6
Time-span of bug-introducing changes	0	5	3	0	1	4	7
Total	5	10	10	6	1	6	22

to be judged upon. The data from this manual analysis is available online to the interested reader.[8]

We report the amount of changes within each category in Table 7. We report the results for MA-SZZ, since it inherits the selection mechanisms of B-SZZ and AG-SZZ and it is not prone to flagging meta-changes as potential bug-introducing changes. With limited domain knowledge about the studied projects, we are able to find 38 out of 60 changes from the upper extreme of our data that may taint MA-SZZ results. Conversely, a manual analysis of the lower extreme of the MA-SZZ generated data only finds 1 out of 60 changes which is included in the categories of changes that are unlikely to introduce bugs (initial code import). Such result suggests focusing analysis effort on the upper extreme is a cost effective mechanism for spotting suspicious entries in SZZ-generated data. In the following, we describe each of the categories that are presented in Table 7.

Missing bug-introducing changes. This category refers to the bugs to which all of the bug-introducing changes are dated after the bug-report date (see Section 3). For instance, MA-SZZ flags changes 832300 and 832306 as bug-introducing for bug OPENJPA-1369. However, bug OPENJPA-1369 was created at October 30th of 2009, while both potential bug-introducing changes were recorded in November 2009. Hence, such bug-introducing changes could not have introduced that bug.

Directory/File renaming. MA-SZZ flags potential bug-introducing changes that are actually directory/file renaming changes. For example, when analyzing the PIG-204 bug, MA-SZZ flags change #617338 as a potential bug-introducing change within `MapReducePlanCompile.java` file. However, change #617338 performs a deletion of file `/physicalLayer/-MapreducePlanCompiler.java` and an addition of file `/executionengine/MapreducePlanCompiler.java`. SVN does not track renamed files. Therefore, MA-SZZ cannot connect the code changes that are performed on `/executionengine/MapreducePlanCompiler.java` to the changes that are performed on `/physicalLayer/-MapreducePlanCompiler.java`. The same sort of change happens when considering HBASE-5967 on change #1342856, in which several files are moved to different packages in order to convert the source tree into Maven-compliant modules. These broken historical links could be heuristically recovered using repository mining techniques like those proposed by Steidl *et al.* [52].

Semantically equivalent changes. It is known that AG-SZZ and MA-SZZ are aware of comments, blank lines, indentation, and whitespace changes. However, SZZ still has problems with other sorts of format changes. In this regard, Williams and Spacco *et al.* [26] used the DiffJ tool to make SZZ aware of other types of format changes (*e.g.*, reordering and renaming parameters).

Nevertheless, we find other types of changes of which current SZZ implementations are not aware. For example, when analyzing the TUSCANY-1867 bug, MA-SZZ selects change #641567 as a bug-introducing change within file `SCANNodeManagerService.java`, which is a java interface. Change #641567 only removes the "`public`" access modifier from the methods defined in this interface. The use of the "`public`" modifier is optional in Java interfaces since methods defined in interfaces are "`public`" by default.[9] For instance, using DiffJ, such a change would be mapped to the `accessRemoved` change type. Even though DiffJ makes SZZ aware that the access modifier was removed, it is not obvious that such a change is not bug-introducing. Only by making SZZ aware that `public` and `blank` interface methods have the same level of accessibility would allow SZZ to avoid flagging such a change as bug-introducing.

Another type of semantically equivalent changes is the renaming of variables. For instance, change #563607 has the highest count of future bugs in project Camel (MA-SZZ). This change renames the variable `log` to `LOG` within file `MyInterceptorProcessor.java`. DiffJ classifies such a change as `variableAdded` (at the variable definition) and `codeChanged` (when the renamed variable is used within methods). Again, it is not obvious that `variableAdded` and `codeChanged` should be interpreted as a change that cannot introduce a bug.

Other example of a semantically equivalent change is a change to the Java iteration style for loop constructs. For example, `for (Object obj : objects)` and `for (int i = 0; i < objects.size(); i++)`, are semantically equivalent. DiffJ flags these as the `codeChanged` type. It is again non-trivial to detect that these are semantics preserving changes.

Naturally, we would suggest that SZZ could be improved by making it more language-aware (*e.g.*, to be aware that the default access within Java interfaces is `public`). This would allow SZZ to avoid flagging changes that do not change system behaviour as bug-introducing.

Initial code importing changes. Our manual analysis reveals that MA-SZZ flags changes that are initial code importing changes. Initial code import changes may still represent true positives. However, in case that the analyzed project by SZZ had been migrated from one SCM system (*e.g.*, CVS) to another (*e.g.*, SVN), a good practice would be to enable SZZ to trace back further in the old data, since the initial code importing of an SCM might not be the starting point of the project development, but an import of several changes that have been performed in the past. Indeed, the initial code import changes that we analyze refer

8. http://sailhome.cs.queensu.ca/replication/szz_evaluation/

9. https://docs.oracle.com/javase/tutorial/java/IandI/interfaceDef.html (April 2016)

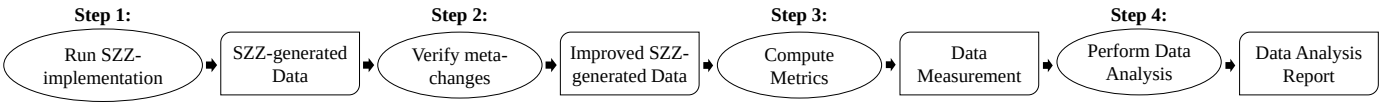


Fig. 10: **The proposed methodology to be used along with our framework.** The elipses show the steps of the methodology, while the rectangles show the output of each step.

to hundreds of classes being imported (300 to 800), which might suggest that the work that is contained in those initial code imports had evolved progressively on another SCM. For instance, MA-SZZ flags change #355543 for the AMQ-4563 bug. Change #355543 refers to the initial code import of the ActiveMQ 4.x code base within the Subversion SCM. However, ActiveMQ had been hosted in CVS before. It is likely that MA-SZZ could trace back further if it had access to the CVS data.[10]

Backout changes. A backout change is a change that reverts another change [53]. Backout changes may also mislead SZZ to flag bug introducing changes. For instance, MA-SZZ flags a backout change (an `if` statement re-added by change #1243702) for bug OPENJPA-2131 as bug-introducing. However, change #1243702 re-adds that same `if` statement that was introduced in change #1222818 for the purpose of reverting to the previous version of the studied project. SZZ should flag change #1222818 (which is the actual origin of the `if` statement) as bug-introducing rather than change #1243702.

We suggest the use of techniques to identify backout changes [53]. SZZ can skip such changes and trace back further to find the actual bug-introducing changes.

General changes with low likelihood of bug-introduction. Changes with low likelihood of being bug-introducing are changes where we perform a deeper analysis of the code and identify that they are unlikely to introduce the bug. For instance, the most recent potential bug-introducing change of bug MAHOUT-1017 (change #1336424) adds a new message when throwing an exception — from `throw new IllegalStateException("Clusters is empty!");` to `throw new IllegalStateException("No input clusters found. Check your -c argument.")`. However, such a change is very unlikely to introduce bug MAHOUT-1017, since this bug is about searching data in wrong directories.

SZZ implementations still have room for improvement. SZZ implementations have to handle changes such as directory/file renames, semantically equivalent changes, and initial code importing changes in order to more accurately flag bug-introducing changes.

5.5 Analysis with prior work data

We analyze the Eclipse SZZ-generated data from prior work of Williams and Spacco [26], which includes 3,282 transactions (*i.e.*, grouped CVS commits) of 2,341 bug reports.[11] From this data, 27 transactions (containing 43 lines of code) were manually tagged as 8 false positives and 19 true positives.

10. http://activemq.apache.org/cvs.html (April 2016)
11. https://eclipse.org/eclipse/ (April 2016)

We compute the *count of future bugs* metric for the entire dataset. Unfortunately, we could not compute the *disagreement ratio* metric nor the *timespan of bug-introducing changes* because such data was not available.

The upper MAD of the count of future bugs is one, since a great majority of transactions lead to one future bug only (69%). 7 out of the 8 false positives (87%) have a count of future bugs that is higher than the upper MAD. This result suggests that our framework holds promise in identifying false positives in SZZ-generated data. If we were able to compute all three criteria, they could have worked in tandem with each other to spot suspicious entries. If more data from related work were available (*i.e.*, [27]), we could have performed deeper analysis.

6 Practical Guidelines

Prior work enhanced the SZZ approach by improving the way that SZZ considers that a line of code has changed (*e.g.*, ignoring format changes) [11], improving the way that SZZ links the bug-fixing changes to bug-introducing changes (*e.g.*, annotation graph) [11, 26], and changing the selection mechanism of SZZ [27].

Our work proposes a framework to evaluate the data that is generated by SZZ implementations by using three criteria: (i) earliest bug appearance, (ii) future impact of changes, and (iii) realism of bug introduction. In Figure 10, we show an overview of a methodology that we propose to be used along with our framework. We explain each step in the methodology below.

After running an SZZ implementation (Step 1), it is important to verify if the SZZ implementation works in tandem with the SCM that is being used (Step 2). For instance, in this work, we observe that B-SZZ and AG-SZZ may flag meta-changes as potential bug-introducing changes when they are applied to the Apache Subversion SCM — which makes heavy use of branches and merges to manage the software development process of Apache projects. The verification that is performed in Step 2 may lead to fixes on how the SZZ-generated data is produced (*e.g.*, we fix the annotation graph of SZZ which leads to the MA-SZZ implementation). Next (Step 3), the criteria that are proposed in our framework can be computed using the SZZ-generated data. We suggest that the criteria of our framework should be used in tandem with each other. For instance, while an SZZ implementation may obtain a lower disagreement ratio, it may just reveal that such an implementation is flagging much more cosmetic changes as bug-introducing. In such a case, another criterion such as the count of future bugs could highlight a problem with the implementation.

Finally, in Step 4, the obtained measurements can then be used as an input for data analyses. In this paper, we

propose the use of the upper MAD (Section 3) to spot extremes within the SZZ-generated data that are likely to be suspicious. We also perform a manual analysis of the extremes of our SZZ-generated data and highlight opportunities for improvement of current state of the art SZZ implementations.

Nevertheless, the data analysis step (Step 4) should be performed based on the practitioner's experience and availability (*e.g.*, the choice of certain thresholds and outlier analyses). Moreover, we highlight below some practical guidelines based on our observations.

1) **Use affected-versions as a selection criterion:** We factor in the estimates of the development teams in our framework by using the *affected-version* field that is available in the JIRA ITS. We suggest that the earliest affected-version date can be used to filter out bug-introducing changes that are likely to be false positives in the SZZ-generated data.
2) **Investigate suspicious entries:** Our proposed criteria such as *future impact of changes* and *realism of bug introduction* can be used to expose the extremes of the SZZ-generated data at hand (*e.g.*, bug-introducing changes with hundreds of future bugs). We suggest that practitioners can use such criteria to flag suspicious entries within the SZZ-generated data and act on it accordingly. For instance, we perform a manual analysis of the SZZ-generated data at hand in Section 5.4 in which we highlight opportunities to improve the current state of the art of SZZ implementations.

7 Threats to Validity

This section describes the threats to the validity of our work.

7.1 External Validity

External threats to validity are concerned with the extent to which we can generalize our results. We studied 10 open source projects from the Apache Software Foundation. Even though our results may not generalize to other projects, we carefully chose projects of different sizes and domains to combat bias.

In addition, our *earliest bug appearance* criterion depends on the availability of the affected-version data. Nevertheless, the main goal of this paper is not to provide generalizable SZZ results, but rather to provide a framework for researchers and practitioners to evaluate SZZ-generated data before using it.

7.2 Internal Validity

Internal threats to validity are concerned with the extent to which our conclusions are valid when considering the particular dataset that we used. In this regard, the main internal threat is related to the representativeness of the affected-versions data that we used. In addition, we are aware that the data that is provided by the development team may not be complete. For instance, a bug may be present in versions that were released prior to the earliest version that is indicated by team members. Hence, the disagreement ratio should be interpreted as a lower-bound, rather than a concrete value.

The scarcity of affected-versions data limits the scope of our *earliest bug appearance* analysis. However, our analysis is valuable because: (1) to the best of our knowledge, no prior work has considered the affected-versions when evaluating SZZ, and (2) although only a subset of the full dataset could be analyzed, the subset surpasses the amount of manually verified bug-fix changes in prior research [26, 27].

Moreover, we implement five variations of SZZ based on the specifications that is provided by related work [9, 11, 27]. This introduces the risk that our implementations may not be fully aligned with what was proposed by prior work.

7.3 Construct Validity

Threats to construct validity are concerned with the degree to which a metric is measuring what it claims to be measuring. In our work, we factor in the knowledge of team members by using the affected-version field from the JIRA ITS. The affected-version enables us to identify entries of the SZZ-generated data that are suspicious, *i.e.*, the potential bug-introducing changes that are dated after the release of the earliest affected-version. Nevertheless, we do not identify the *true* bug-introducing changes. For instance, even if a bug-introducing change is dated before the release of the earliest affected-version (*i.e.*, the *unknown* bug-introducing change according to our classification), such a bug-introducing change can still be incorrectly flagged by SZZ. The ground truth for such a problem is not available, since one would have to have access to domain experts who would have to specify which exact code changes led to a bug. Moreover, we do not observe any case in which the date of the bug-introducing change is the same as the release date of the earliest affected-version in our SZZ-generated data.

Our manual analysis of the suspicious bug-introducing and bug-fixing changes is subject to our own opinion. If the data were analyzed by another group of researchers, they may arrive at different results. Nonetheless, we limited the manual analysis categories to those that could be understood with limited domain knowledge. Hence, we suspect that although other researchers may highlight other characteristics, the conclusions will remain stable. Moreover, the data that we use in our manual analysis is publicly available and researchers are encouraged to replicate and extend our study.[12]

SZZ has some core limitations. For example, SZZ cannot analyze bugs that are due to missing a change (in contrast to bugs that are due to performing incorrect changes). Our proposed criteria help researchers and practitioners to find false positives in SZZ-generated data. However, the identification of false negatives (*i.e.*, bug-introducing changes that are not flagged as such by SZZ) remains an open challenge.

There are more sophisticated techniques than SZZ for studying the future impact of a code change [42, 43, 54]. For instance, Herzig and Zeller [54] used change genealogies to mine cause-effect-chains of code changes. This approach may be used to study what sequence of events cause a bug, for example. Nevertheless, we did not intend to propose an exhaustive list of criteria. Instead, we propose three

12. http://sailhome.cs.queensu.ca/replication/szz_evaluation/

simple criteria that can spot suspicious entries in the SZZ-generated data and that can be computed when using the commonly used SZZ implementations [9–11]. Finally, our proposed framework can be used to guide non-experts in their exploration of large sets of SZZ-generated data.

8 Conclusion and Future Work

In this paper, we propose a framework to evaluate the implementations of the SZZ approach. The proposed framework is comprised of three criteria: (1) earliest bug appearance, (2) future impact of changes, and (3) realism of bug introduction. Through analysis of ten projects from the Apache Software Foundation, we evaluate five implementations of the SZZ approach using our framework. We find that:

- The suggested SZZ improvements by previous research (*i.e.*, R-SZZ, L-SZZ) have a lower rate of agreement with respect to the earliest affected-versions criterion. When R-SZZ was applied to the ActiveMQ project, up to 50% of the bugs are linked to bug-introducing changes that are not in agreement with the earliest affected-version of those bugs.
- B-SZZ has the worst results regarding the *future impact of a change* criterion. For example, 45% of the identified bug-introducing changes by B-SZZ lead to multiple future bugs.
- B-SZZ also has the worst results for the *realism of bug introduction criteria* criterion. When using B-SZZ, we identified bug-introducing changes that span over one year for 65% of the bugs. Even for MA-SZZ (an enhancement to B-SZZ), 46% of bugs have bug-introducing changes that span over one year.
- We report several opportunities to improve SZZ implementations. For example, one has to carefully consider meta-changes, directory/file renaming changes, syntax equivalent changes, and backout changes.

We recognize that SZZ still has core problems that are not currently addressed. For instance, SZZ cannot find the true location of bugs that are fixed by only adding code. In addition, SZZ may flag potential bug-introducing changes that were correct changes at the time of their writing, but start leading to bugs because of external changes (*e.g.*, user requirement changes or project structure changes). Nevertheless, the main goal of our proposed framework is to help practitioners and researchers to evaluate the SZZ-generated data at hand. For instance, practitioners and researchers can analyze the upper extremes that are spotted by our proposed criteria to find suspicious entries in the SZZ-generated data.

Our results suggest that SZZ implementations that are currently available still need improvements in order to accurately detect bug-introducing changes. Additionally, researchers and practitioners should carefully scrutinize potential bug-introducing changes that are suggested by current state of the art SZZ implementations. The evaluation framework that we propose in this paper is a first step towards improving the quality of SZZ-generated data and SZZ implementations themselves.

References

[1] M. Lerner, "Software maintenance crisis resolution: The new ieee standard," in *Software Development Journal*, vol. 2, 1994, pp. 65–72.

[2] 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*, 2006, pp. 492–501.

[3] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," in *IEEE Transactions on Software Engineering Journal*, 2005, pp. 897–910.

[4] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 78–88.

[5] P. L. Li, J. Herbsleb, M. Shaw, and B. Robinson, "Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc." in *Proceedings of the 28th International Conference on Software Engineering*, 2006, pp. 413–422.

[6] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" in *IEEE Transactions on Software Engineering Journal*, vol. 34, 2008, pp. 181–196.

[7] A. Mockus and D. M. Weiss, "Predicting risk of software changes," in *Bell Labs Technical Journal*, vol. 5, 2000, pp. 169–180.

[8] 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," in *IEEE Transactions on Software Engineering Journal*, vol. 39, 2013, pp. 757–773.

[9] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM SIGSOFT Software Engineering Notes*, vol. 30, 2005, pp. 1–5.

[10] J. Eyolfson, L. Tan, and P. Lam, "Do time of day and developer experience affect commit bugginess?" in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 153–162.

[11] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, 2006, pp. 81–90.

[12] M. Asaduzzaman, M. C. Bullock, C. K. Roy, and K. A. Schneider, "Bug introducing changes: A case study with android," in *Proceedings of the 9th Working Conference of Mining Software Repositories*, 2012, pp. 116–119.

[13] F. Rahman and P. Devanbu, "Ownership, experience and defects: a fine-grained study of authorship," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 491–500.

[14] K. Pan, S. Kim, and E. J. Whitehead Jr, "Toward an understanding of bug fix patterns," in *Empirical Software Engineering Journal*, vol. 14, 2009, pp. 286–315.

[15] S. Kim and E. J. Whitehead, Jr., "How long did it take to fix bugs?" in *Proceedings of the 3rd International Workshop on Mining Software Repositories*, 2006, pp. 173–174.

[16] H. Yang, C. Wang, Q. Shi, Y. Feng, and Z. Chen, "Bug inducing analysis to prevent fault prone bug fixes,"

in *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering*, 2014, pp. 620–625.

[17] M. L. Bernardi, G. Canfora, G. A. Di Lucca, M. Di Penta, and D. Distante, "Do developers introduce bugs when they do not communicate? the case of eclipse and mozilla," in *Proceedings of 16th European Conference on Software Maintenance and Reengineering*, 2012, pp. 139–148.

[18] F. Rahman, C. Bird, and P. Devanbu, "Clones: What is that smell?" in *Empirical Software Engineering Journal*, vol. 17, 2012, pp. 503–530.

[19] G. Canfora, M. Ceccarelli, L. Cerulo, and M. Di Penta, "How long does a bug survive? an empirical study," in *Proceedings of the 18th Working Conference on Reverse Engineering*, 2011, pp. 191–200.

[20] J. Ell, "Identifying failure inducing developer pairs within developer networks," in *Proceedings of the 35th International Conference on Software Engineering*, 2013, pp. 1471–1473.

[21] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proceedings of the 29th International Conference on Software Engineering*, 2007, pp. 489–498.

[22] D. A. da Costa, U. Kulesza, E. Aranha, and R. Coelho, "Unveiling developers contributions behind code commits: an exploratory study," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, 2014, pp. 1152–1157.

[23] Y. Kamei, S. Matsumoto, A. Monden, K.-i. Matsumoto, B. Adams, and A. E. Hassan, "Revisiting common bug prediction findings using effort-aware models," in *Proceedings of the 26th IEEE International Conference on Software Maintenance*, 2010, pp. 1–10.

[24] T. Fukushima, Y. Kamei, S. McIntosh, K. Yamashita, and N. Ubayashi, "An empirical study of just-in-time defect prediction using cross-project models," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 172–181.

[25] O. Mizuno and H. Hata, "Prediction of fault-prone modules using a text filtering based metric," in *International Journal of Software Engineering and Its Applications*, vol. 4, 2010, pp. 43–52.

[26] C. Williams and J. Spacco, "Szz revisited: verifying when changes induce fixes," in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, 2008, pp. 32–36.

[27] S. Davies, M. Roper, and M. Wood, "Comparing text-based and dependence-based approaches for determining the origins of bugs," in *Software: Evolution and Process Journal*, vol. 26, 2014, pp. 107–139.

[28] J. Śliwerski, T. Zimmermann, and A. Zeller, "Hatari: raising risk awareness," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, 2005, pp. 107–110.

[29] L. Prechelt and A. Pepper, "Why software repositories are not used for defect-insertion circumstance analysis more often: A case study," in *Information and Software Technology Journal*, vol. 56, 2014, pp. 1377–1389.

[30] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead Jr, "Mining version archives for co-changed lines," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, 2006, pp. 72–75.

[31] C. C. Williams and J. W. Spacco, "Branching and merging in the repository," in *Proceedings of the 5th International Working Conference on Mining Software Repositories*, 2008, pp. 19–22.

[32] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proceedings of the 16th International Conference on Software Maintenance*, 2000, pp. 120–130.

[33] D. Čubranić and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proceedings of the 25th International Conference on Software Engineering*, 2003, pp. 408–418.

[34] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Proceedings of the 19th International Conference on Software Maintenance*, 2003, pp. 23–32.

[35] M. W. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," in *IEEE Transactions on Software Engineering Journal*, vol. 31, 2005, pp. 166–181.

[36] A. E. Hassan and R. C. Holt, "The top ten list: Dynamic fault prediction," in *Proceedings of the 21st International Conference on Software Maintenance*. IEEE, 2005, pp. 263–272.

[37] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings of the 27th International Conference on Software Engineering*, 2005, pp. 284–292.

[38] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the bugs are," in *ACM SIGSOFT Software Engineering Notes*, vol. 29, 2004, pp. 86–96.

[39] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," in *IEEE Transactions on Software Engineering Journal*, vol. 39, 2013, pp. 552–569.

[40] M. Kim, S. Sinha, C. Gorg, H. Shah, M. J. Harrold, and M. G. Nanda, "Automated bug neighborhood analysis for identifying incomplete bug fixes," in *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, 2010, pp. 383–392.

[41] V. S. Sinha, S. Sinha, and S. Rao, "Buginnings: identifying the origins of a bug," in *Proceedings of the 3rd Indian Software Engineering Conference*, 2010, pp. 3–12.

[42] O. Alam, B. Adams, and A. E. Hassan, "Preserving knowledge in software projects," in *Systems and Software Journal*, vol. 85, 2012, pp. 2318–2330.

[43] O. Alam, A. Bram, and A. E. Hassan, "Measuring the progress of projects using the time dependence of code changes," in *Proceedings of the 25th IEEE International Conference on Software Maintenance*, 2009, pp. 329–338.

[44] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting fault incidence using software change history," in *IEEE Transactions on Software Engineering Journal*, vol. 26, 2000, pp. 653–661.

[45] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, "An empirical study of dormant bugs," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 82–91.

[46] D. C. Howell, "Median absolute deviation," in *Encyclopedia of Statistics in Behavioral Science*, 2005, available at

Wiley Online Library.

[47] C. Leys, C. Ley, O. Klein, P. Bernard, and L. Licata, "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median," in *Experimental Social Psychology Journal*, vol. 49, 2013, pp. 764–766.

[48] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead Jr, "Mining version archives for co-changed lines. technical report". Available at: http://www.st.cs.uni-sb.de/softevo/, 2006, Accessed at: 30-04-2016.

[49] F. Servant and J. A. Jones, "History slicing: assisting code-evolution tasks," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 43.

[50] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." in *Psychological Bulletin Journal*, vol. 114, 1993, p. 494.

[51] K. Charmaz, *Constructing grounded theory*. Sage, 2014.

[52] D. Steidl, B. Hummel, and E. Juergens, "Incremental origin analysis of source code files," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 42–51.

[53] R. Souza, C. Chavez, and R. Bittencourt, "Rapid releases and patch backouts: A software analytics approach," in *IEEE Software Journal*, vol. 32, 2015, pp. 89–96.

[54] K. Herzig and A. Zeller, "Mining cause-effect-chains from version histories," in *Proceedings of the 22nd IEEE International Symposium on Software Reliability Engineering*, 2011, pp. 60–69.

Daniel Alencar da Costa is a PhD student at the Federal University of Rio Grande do Norte (UFRN), Brazil. He received his Bachelor's degree from the University Centre of The Pará State (CESUPA) and his MSc degree from UFRN. In his research, Daniel has studied the delay that happens during the delivery of new software functionalities to end users. His work has been published at software engineering venues, such as the International Conference on Mining Software Repositories (MSR) and the IEEE International Conference on Software Maintenance and Evolution (ICSME). His interests also include general mining software repositories and empirical software engineering research. More information at http://danielcalencar.github.io.

Shane McIntosh is an Assistant Professor in the Department of Electrical and Computer Engineering at McGill University, where he leads the Software Repository Excavation and Build Engineering Labs (Software REBELs). He received his Bachelor's degree from the University of Guelph and his MSc and PhD degrees from Queen's University, where he held an NSERC Vanier Scholarship and was awarded the Governor General's Academic Gold Medal. In his research, Shane uses empirical software engineering techniques to study software build systems, release engineering, and software quality. His research has been published at several top-tier software engineering venues, such as the International Conference on Software Engineering (ICSE), the International Symposium on the Foundations of Software Engineering (FSE), and the Springer Journal of Empirical Software Engineering (EMSE). Shane actively collaborates with academics in Canada, the Netherlands, Singapore, Brazil, and Japan, as well as industrial practitioners in Germany and the USA. More about Shane and his work is available online at http://rebels.ece.mcgill.ca/.

Weiyi Shang is an assistant professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal. His research interests include big-data software engineering, software engineering for ultra-large-scale systems, software log mining, empirical software engineering, and software performance engineering. Shang received his PhD in computing from Queen's University, Canada. Contact him at shang@encs.concordia.ca.

Uirá Kulesza is an associate professor in the Department of Informatics and Applied Mathematics (DIMAp) at Federal University of Rio Grande do Norte (UFRN), Brazil. He received his Bachelor degree in Computer Science from the Federal University of Campina Grande, his MSc degree in Computer Science from University of So Paulo, and PhD in Computer Science from Pontifical Catholic University of Rio de Janeiro (PUC-Rio). His main research interests include software evolution, software architecture, and software analytics. He has published at several top-tier software engineering venues, such as the International Conference on Software Engineering (ICSE), the International Symposium on the Foundations of Software Engineering (FSE), and the European Conference on Object-Oriented Programming (ECOOP). More about Uirá and his work is available online at http://www.dimap.ufrn.br/~uira.

Roberta Coelho is a professor in the Department of Informatics and Applied Mathematics at Federal University of Rio Grande do Norte, Brazil. She received an MSc dregree from Federal University of Pernambuco, Brazil, and a PhD degree in computer science from PUC-Rio in cooperation with Lancaster University. Her research interests include software testing, software dependability and exception handling.

Ahmed E. Hassan is the Canada Research Chair (CRC) in Software Analytics, and the NSERC/BlackBerry Software Engineering Chair at the School of Computing at Queens University, Canada. His research interests include mining software repositories, empirical software engineering, load testing, and log mining. Hassan received a PhD in Computer Science from the University of Waterloo. He spearheaded the creation of the Mining Software Repositories (MSR) conference and its research community. Hassan also serves on the editorial boards of IEEE Transactions on Software Engineering, Springer Journal of Empirical Software Engineering, Springer Journal of Computing, and PeerJ Computer Science. Contact ahmed@cs.queensu.ca. More information at: http://sail.cs.queensu.ca/.