The Impact of Refactoring Changes on the SZZ Algorithm: An Empirical Study

Edmilson Campos Neto*†, Daniel Alencar da Costa‡, Uirá Kulesza*
*Federal University of Rio Grande do Norte, Natal, Brazil
uira@dimap.ufrn.br
†Federal Institute of Education, Science and Technology of Rio Grande do Norte, Natal, Brazil
edmilson.campos@ifrn.edu.br
‡Queen's University, Kingston, Canada
daniel.alencar@queensu.ca

***Abstract*—SZZ is a widely used algorithm in the software engineering community to identify changes that are likely to introduce bugs (*i.e.*, bug-introducing changes). Despite its wide adoption, SZZ still has room for improvements. For example, current SZZ implementations may still flag refactoring changes as bug-introducing. Refactorings should be disregarded as bug-introducing because they do not change the system behaviour. In this paper, we empirically investigate how refactorings impact both the input (*bug-fix changes*) and the output (*bug-introducing changes*) of the SZZ algorithm. We analyse 31,518 issues of ten Apache projects with 20,298 *bug-introducing changes*. We use an existing tool that automatically detects refactorings in code changes. We observe that 6.5% of lines that are flagged as bug-introducing changes by SZZ are in fact refactoring changes. Regarding bug-fix changes, we observe that 19.9% of lines that are removed during a fix are related to refactorings and, therefore, their respective inducing changes are false positives. We then incorporate the refactoring-detection tool in our *Refactoring Aware SZZ Implementation* (RA-SZZ). Our results reveal that RA-SZZ reduces 20.8% of the lines that are flagged as bug-introducing changes compared to the state-of-the-art SZZ implementations. Finally, we perform a manual analysis to identify change patterns that are not captured by the refactoring identification tool used in our study. Our results reveal that 47.95% of the analyzed bug-introducing changes contain additional change patterns that RA-SZZ should not flag as bug-introducing.**

***Index Terms*—SZZ algorithm, refactoring, bug-introducing change, bug-fix change**

I. Introduction

Much research has been invested in bug prediction [1]–[6], [6]–[11]. For example, prior work predicts bugs using information from source code repositories. Examples of information used in the bug prediction are code churn metrics [1] [12] [2], object-oriented design metrics [3], complexity of code changes [5] [4], change meta-data [6] *etc*. Bug prediction is helpful to development teams to prioritize code regions with a greater probability of bugs occurrence [13].

Although bug prediction is an important tool, it was not possible to study the origin of bugs in large-scale scenarios until the introduction of the SZZ algorithm [14]. The SZZ algorithm traces back the code history to find changes that are likely to introduce bugs, *i.e.*, the so-called *bug-introducing changes*. SZZ was initially proposed by Śliwerski, Zimmermann and Zeller [14] – hence the acronym – and improved by Kim *et al*. [15]. However, SZZ is not without limitations [16]–[18], such as the recognition of *equivalent changes*, *i.e.*, changes that do not modify system behaviour.

Several studies [19]–[21] state that code refactoring is a frequently used technique by developers during bug-fix changes. SZZ may produce inaccurate data by not recognizing that bug-fix changes may contain interleaved refactorings, since code refactoring does not directly fix a bug [22]. Similarly, SZZ may erroneously flag refactoring changes as bug-introducing changes. For example, if SZZ flags a line as potentially bug-introducing, but such a line is the result of a method rename, SZZ should trace the history further using the previous method name.

Prior research has evaluated the SZZ algorithm [16]–[18]. For example, da Costa *et al.* [17] used an evaluation framework to appraise the results of SZZ. In addition, Prechelt *et al.* [16] also evaluated SZZ in an industrial setting. These studies help to identify possible enhancements to be implemented to SZZ and what are the hindrances to perform better evaluations of SZZ.

Nevertheless, little is known about the impact of refactoring changes on the results of the SZZ algorithm. Studying the impact of refactoring changes is important because, differently from other SZZ limitations (*e.g.*, the identification of renames in directories and files [17]), such changes may impact the SZZ algorithm regardless the *Version Control System* (VCS) that is used to implement SZZ. The goal of our research is to quantify the refactoring changes that occur both in bug-fixing changes (*i.e.*, the input of the SZZ algorithm) and bug-introducing changes (*i.e.*, the SZZ-generated data). We use RefDiff [23], a recent proposed state-of-the-art tool to detect refactoring changes in our analyses. We address the following research questions (RQs):

- **RQ1. What is the impact of refactoring changes upon existing SZZ implementations?** The refactoring lines found by RefDiff represent 6.5% (30,562 lines) of the potential bug-introducing lines that are flagged by MA-SZZ (*meta change aware SZZ*)—a state-of-the-art SZZ implementation. These results should be interpreted as a lower bound of the total of refactoring lines

978-1-5386-4969-5/18 © 2018 IEEE

SANER 2018, Campobasso, Italy
Technical Research Papers

Accepted for publication by IEEE. © 2018 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

that SZZ may flag, since RefDiff can only detect 13 refactoring types [23]. Nevertheless, SZZ may also be tainted by refactoring changes that occur during bug-fixes. We identify that 19.9% (110,928) of the removed lines during fixes are related to refactoring changes—they should not be traced back by SZZ. Our results suggest that refactoring changes during bug-fixes may have a considerable impact on current SZZ implementations in terms of producing false positives.

- **RQ2. How many false bug-introducing changes can be removed from the SZZ-generated data?** We incorporate the RefDiff tool into an existing SZZ implementation and propose a *Refactoring Aware SZZ* implementation (RA-SZZ). RA-SZZ reduces 20.8% of the lines that were flagged as bug-introducing changes by MA-SZZ.
- **RQ3. Can we find other change patterns that are not supported by RefDiff?** After manually analyzing a statistical sample of the RA-SZZ-generated data (with 95% of confidence), we identify that 47.95% are related to equivalent changes that RA-SZZ should not flag as bug-introducing, such as undetected refactoring (8.49%); multiple refactoring per line (15.89%); addition/removal of unnecessary code (2.47%); swap iteration style (1.92%) *etc*. In future work, we plan to incorporate the detection of these changes into the RA-SZZ algorithm.

The remainder of this paper is organized as follows. In Section II, we present the background material of our paper. In Section III, we describe our methodology, while in Section IV we describe the results of our study. In Section V, we discuss the threats to validity of our study. In Section VI we draw our conclusions and outline venues for future work.

II. Background & Related Work

In this section, we describe the SZZ algorithm and explain why refactoring changes may distort the SZZ results. Finally, we also describe how refactoring may be interleaved within a bug-fix change.

A. Bug-fix & Bug-introducing Changes

Definition 1—Bug-fix change. *Bug-fix changes* refer to changes that are known to fix a bug that is reported on an *Issue Tracking System* (ITS, *e.g.*, JIRA and Bugzilla). There exist several heuristics that rely on ITS information to identify *bug-fix changes* [24]–[26]. For example, if a change log contains references to bug IDs that can also be found on the respective ITSs, such a change is deemed as bug-fixing. During a *bug-fix change* several lines are added, removed or modified. We call each one of these lines as *bug-fix lines*.

Definition 2—Bug-introducing change. *Bug-introducing changes* refer to code changes that eventually induce a bug fix change (Definition 1) in the future system [14], [15]. A bug-introducing change contains a set of lines of code that are added, removed or modified during the change. We refer to the code lines of a bug-introducing change as *bug-introducing lines*.

B. SZZ Algorithm

Identifying and preventing bugs in software systems is an overarching goal of the software engineering (SE) community [10], [12], [14], [15], [27]–[29]. In this matter, Śliwerski *et al.* [14] proposed the seminal SZZ algorithm to identify bug-introducing changes (see Definition 2).

In order to identify bug-introducing changes, the SZZ algorithm starts analyzing the bug-fix changes (see Definition 1). Figure 1 provides an illustrative example of how SZZ works. In V2, a loop is added to print an array (V2: lines `3`,`4`,`5`). This change introduces an *array index out of range* bug, since the loop has a number of iterations that is greater than the elements of the array. Eventually, this bug is reported on the ITS with an issue ID of `#123`. In this case, V2 is a bug-introducing change that is fixed by V3, which changes the loop terminating condition, *i.e.*, statement (V3: line `3`).

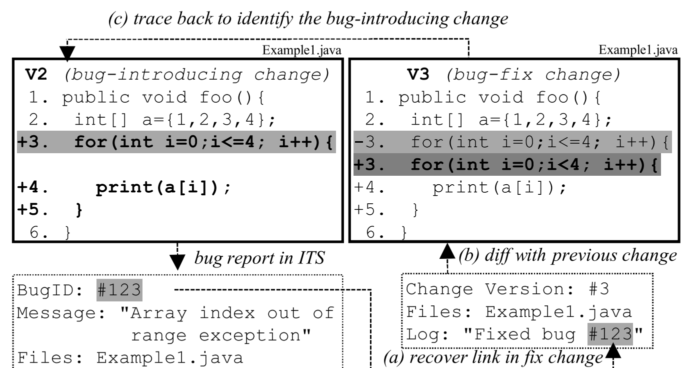

Fig. 1: Illustrative Example 1

By using the aforementioned heuristics (*e.g.*, references of bug IDs within change logs), the SZZ algorithm identifies a bug ID within the V3 log, which refers to the bug `#123` (Figure 1.a). Hence, V3 is found to be a bug-fix change. Next, SZZ performs a diff operation between the bug-fix change and the previous change (Figure 1.b) to identify how the bug was fixed. In our example, line 3 was modified to fix the bug `#123`. Finally, to locate the bug-introducing change, SZZ traces back in code history (*e.g.*, using the *git blame* function) to find the change that introduced the bug, *i.e.*, the bug-introducing change (Figure 1.c).

The initial SZZ implementation [14] uses the *annotate* function provided by particular VCSs (*e.g.*, Subversion) to identify the latest change that introduced each removed/modified line of a bug-fix change. Kim *et al.* [15] improved SZZ by using an annotation-graph that helps SZZ to avoid flagging comments, blank lines and cosmetic changes as potential bug-introducing changes. Later, da Costa *et al.* [17] noticed that SZZ erroneously flagged meta-changes, such as branch/merge and property changes, as potential bug-introducing changes. Then, they proposed the MA-SZZ (*meta change aware SZZ*) [17] implementation, which disregards meta-changes as bug-introducing. The authors also proposed a conceptual framework to evaluate SZZ implementations. The

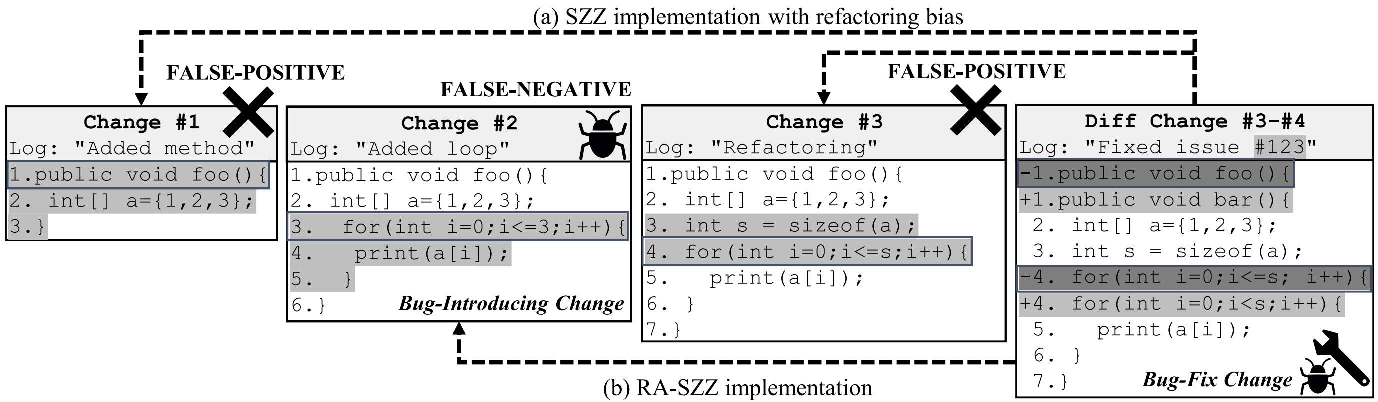

Fig. 2: Illustrative Example 2 showing the difference SZZ-generated data caused by RA-SZZ implementation

framework was used to compare the results of five previous SZZ implementations. Their results suggest that current state-of-the-art SZZ implementations may still be improved. For example, SZZ still needs to avoid flagging *equivalent changes* [17] as bug-introducing. An example of an *equivalent change* is the replacement of the old-fashioned for Java loop for the most recent syntax.[1]

C. The Impact of Refactoring Changes

Definition 3—Refactoring. Fowler [22] defines a refactoring as being a change performed to improve the design of a system without changing its external behaviour. He also presents an extensive catalogue of 63 different refactorings. Silva *et al.* [23] proposed RefDiff[2], a tool that is able to automatically identify 13 refactoring types out of the 63 that are mentioned by Fowler. They [23] demonstrate that RefDiff obtains a higher precision and recall when compared to existing approaches. The precision and recall of RefDiff are, respectively, 100% and 88%. Henceforth, we use the term *refactoring changes/lines* to refer to changes or lines that contain one or more of the 13 refactoring types that can be identified by RefDiff.

Definition 4—Refactoring line. *Refactoring lines* refer to a set of lines of code that are deemed as refactoring for each change (*i.e.*, either bug-fix or bug-introducing changes). For example, let us consider a bug-fix (or bug-introducing change) b that has a set of lines of code $L_b = \{l_1, l_2, ..., l_n\}$. After applying a refactoring-detection tool to L_b, we obtain a subset $R_b \subset L_b$ that contains all the *refactoring lines*.

Definition 5—Equivalent changes. We use the term *equivalent changes*, as defined by da Costa *et al.* [17], to refer to code changes (usually related to a specific programming language, *e.g.*, Java) that do not change software behaviour. For example, the swap of Java loop styles—from `for(int i=0 ; i < papers.size() ; i++)` to `for(Paper p : papers)`—or the addition/deletion of equivalent syntax code—removal of an unnecessary `this` keyword. These changes are equivalent within the Java programming language and by no means should modify the behaviour of a software system. Therefore SZZ should not flag such changes as bug-introducing changes. Henceforth, we use the term *equivalent changes* to refer to the instances of aforementioned examples. It is not the goal of our work to dive into the conceptual similarities/differences between a *refactoring change* and an *equivalent change*. Thus, we advise the reader to be attained to our provided definitions when reading these terms in the rest of this paper.

[1]https://coderanch.com/t/408756/java/loop-loop
[2]https://github.com/aserg-ufmg/RefDiff

D. Refactoring within Bug-fixes

A bug-fix change consists of multiple lines. However, not all of these lines are responsible for the bug-fix. For example, developers may decide to refactor the code while performing a bug-fix [16]. We present another illustrative example in Figure 2 to explain this scenario. Change `#4` fixes a problem in the conditional statement of the loop at line `#4`, *i.e.*, instead of $\leq$ the statement should be $<$.

Change `#4` also renames the method `foo()` to `bar()` (*i.e.*, `#3`-`#4`, line `1`). In addition, change `#3` stores the size of the array in the variable `s` (*i.e.*, line `#3`), which is later used in the loop at line `#4`. Both the method rename and the use of a temporarily variable are considered as *refactoring changes* (see Definition 3).

In our example, SZZ would erroneously flag changes `#1` and `#3` (*i.e.*, false positives) when tracing the history of lines `#1` and `#4` of change `#4` (Figure 2.a). On the other hand, an eventual refactoring aware SZZ implementation could automatically identify and handle those refactoring changes. In fact, change `#2` is the one responsible for introducing the bug illustrated in Figure 2.b.

III. Study Settings

In this section, we present details about our studied systems (Section III-A), studied SZZ implementations (Section III-B), and investigated RQs (Section III-C). We describe the motivation and the approach to address each RQ.

TABLE I: Subject Systems Overview

System	System Domain	Bug issues	Bug-Fix
ActiveMQ	an open source messaging broker	2,180	3,581
Camel	an open source integration framework	4,747	10,568
Derby	an open source relational database management system (DBMS)	3,925	9,556
Geronimo	an open source application server for Java Enterprise Edition (J2EE)	3,561	7,702
Hadoop	a software library (framework) for distributed processing	4,824	8,595
HBase	a non-relational database for large data	5,779	11,329
Mahout	a framework for scalable performant machine learning applications	864	1,414
OpenJPA	an object-relational mapping (ORM) solution for Java Persistence API	1,486	4,516
Pig	a high-level platform for creating programs that run on Apache Hadoop	1,904	3,164
Tuscany	an implementation for service-oriented architecture (SOA)	2,248	4,430
		31,518	**64,855**

A. Studied Systems

In this study, we leverage the dataset that was used by da Costa *et al*. [17] and study 10 software systems. Table I shows our studied systems, their bug reports, and bug-fix changes. All of our studied systems use the JIRA ITS to manage their bug reports. Also, the source code changes of our studied systems are hosted in a Subversion VCS. In total, this dataset contains 31,518 bug reports and 64,855 bug-fix changes. The full details of our refactoring dataset is available online to the interested reader.[3]

B. Studied SZZ Implementation

Since we leverage the dataset that was used by da Costa *et al.* [17], our SZZ-generated data (*i.e.*, the bug-introducing changes) was produced by the MA-SZZ algorithm (Section II-B), which is an SZZ implementation that was proposed and evaluated by da Costa *et al.* [17]. In addition, we also enhance MA-SZZ to identify refactoring changes and propose the *Refactoring Aware SZZ* (RA-SZZ). We provide further details in Section III-C.

C. Research Questions

We investigate the following research questions to study the impact of refactoring changes on the SZZ algorithm:

RQ1. What is the impact of refactoring changes upon existing SZZ implementations?

Motivation: Since *refactoring changes* (see Definition 3) rarely are the cause of bugs, SZZ must be aware of these changes to not flag them as bug-introducing. Instead, SZZ should ignore *refactoring changes* when they occur in a bug-fix change, or trace back further in history if SZZ finds a potential bug-introducing change that is a *refactoring change*. This investigation is important because SZZ plays a foundational role in many software engineering (SE) studies [8], [30]–[48]. In case the impact of *refactoring changes* on SZZ is considerable, the SE community may consider to rethink or revisit the use of the SZZ algorithm in existing research.

RQ2. How many false bug-introducing changes can be removed from the SZZ-generated data?

Motivation: It is important to enhance the SZZ algorithm in order to prevent it from flagging *refactoring changes* as bug-introducing changes. If SZZ can properly handle *refactoring changes*, the results will become more credible.

RQ3. Can we find other change patterns that are not supported by RefDiff?

Motivation: Since the RefDiff tool can only identify 13 types of *refactoring changes*, which does not include *equivalent changes* (see Definition 5), we also investigate whether SZZ is producing a considerable number of false positives due to *equivalent changes*.

IV. Study Results

In this section, we present our obtained results for each RQ.

A. RQ1. What is the impact of refactoring changes upon existing SZZ implementations?

Approach: We use RefDiff to automatically identify refactoring changes in both bug-fix changes and bug-introducing changes of our dataset. RefDiff is our tool of choice because it outperformed existing tools that automatically detect refactoring in source code. Silva *et al.* [23] evaluated RefDiff and reported that it obtained the highest precision and recall compared to other existing tools.

After applying RefDiff to the bug-fix and bug-introducing changes, we obtain the *refactoring lines* (see Definition 4) for each change (*i.e.*, either bug-fix or bug-introducing changes).

Result: ***RefDiff indicates that 6.5% (30,562 lines) of the potential bug-introducing lines that are flagged by MA-SZZ are, in fact, refactoring changes.*** Table II shows how many refactoring lines (*refac-lines*) are found within the bug-introducing lines of each of our studied projects.

The *Camel* system has the highest ratio of refactoring to lines (11.2%). Additionally, the *OpenJPA*, *Hadoop Common*, and *Geronimo* systems have similar ratios (10.9%, 10.3% and 9.9%, respectively). Nonetheless, we observe that the overall mean of *refac-lines* is 6.5%. Although the small overall mean, our result should be interpreted as a lower bound observation, since RefDiff can only identify 13 types of refactoring changes. Moreover, we also investigate the number of refactoring changes that are performed during bug-fixes.

We identify that 19.9% (110,928) of the modified lines within bug-fixes should not be traced by SZZ, since such lines are deemed as refactoring changes by RefDiff. Table III shows the proportion of refactoring changes that are found in bug-fixes (*refac-proportion*) for each studied system (*i.e.*, $\frac{\text{refactoring lines}}{\text{bug introducing lines}}$). We observe that *Hadoop Common* contains the highest *refac-proportion* among our studied systems

[3]https://sites.google.com/view/refactoringszz/

(28.1%), followed by *HBase* (21.9%), and *Geronimo* (19.4%). *Derby* has the lowest proportion (12%). On the overall mean, the observed *refac-proportion* is of 19.9%.

Our results suggest that existing SZZ implementations might be generating several false positives by not considering refactoring changes during bug-fixes.

We observe that 6.5% of the bug-introducing lines produced by MA-SZZ are deemed as refactoring changes by RefDiff. Additionally, 19.9% of the lines that were modified during bug-fix changes should not be traced by SZZ, since they are also identified as refactoring. Our observations suggest SZZ may be considerably improved by handling refactoring changes.

B. RQ2. How many false bug-introducing changes can be removed from the SZZ-generated data?

Approach: We incorporate the RefDiff tool on top of MA-SZZ to identify refactoring changes during the analyses of both bug-fixing changes and bug-introducing changes (*i.e.*, RA-SZZ). In case refactoring changes are found in bug-fixes, RA-SZZ does not trace them back in history. As for bug-introducing changes, RA-SZZ does not stop at refactoring changes. Instead, RA-SZZ traces back further in history to find the most likely bug-introducing change.

Modified RefDiff. Figure 3 shows how we adapt RefDiff to work in tandem with our RA-SZZ algorithm. The first step is to gather the refactoring information for each change in our entire dataset. For each file in each change, RefDiff receives the previous version of the file (*i.e.*, before the change) and the current version of the file (*i.e.*, the state of the file after the change). For each file, RefDiff builds an AST and performs a match of the parts that were modified between these files. Next, RefDiff uses heuristics to identify refactoring changes that were performed during these modifications. Then, we add a functionality to RefDiff in order to capture specific information that will be necessary for SZZ. In particular, we capture the *project*, *change revision* (*i.e.*, the commit ID), *refactoring type*, *path of the files*, and *start* and *end* lines of the refactoring. Finally, we store this *refactoring data* in a database that will be later used by our RA-SZZ.

TABLE II: Refactoring changes in bug-introducing lines. The number of bug-introducing changes of this table was produced by the MA-SZZ algorithm.

System	#lines	#refac-lines	refac-proportion
ActiveMQ	19,193	1,322	6.9%
Camel	20,366	2,284	11.2%
Derby	35,038	1,166	3.3%
Geronimo	45,744	4,551	9.9%
Hadoop Common	39,887	4,091	10.3%
HBase	236,761	12,101	5.1%
Mahout	14,858	369	2.5%
OpenJPA	7,866	854	10.9%
Pig	18,807	1,474	7.8%
Tuscany	30,320	2,350	7.8%
Total	**468,840**	**30,562**	**6.5%**

#lines: Number of bug-introducing lines flagged by MA-SZZ; **#refac-lines:** Number of refactoring lines flagged as bug-introducing by MA-SZZ; **refac-proportion:** Proportion of refactorings in bug-introducing lines.

TABLE III: Refactoring changes during bug-fixes. The number of lines presented in this table consists of modified and removed lines within bug-fixes.

System	#lines	#refac-lines	refac-proportion
ActiveMQ	19,348	3,065	16%
Camel	20,669	3,793	18.4%
Derby	52,729	6,323	12%
Geronimo	49,763	9,650	19.4%
Hadoop	49,191	13,803	28.1%
HBase	283,124	62,004	21.9%
Mahout	9,185	1,411	15.4%
OpenJPA	19,801	3,166	16%
Pig	20,806	3,611	17.4%
Tuscany	32,602	4,072	12.5%
Total	**557,218**	**110,928**	**19.9%**

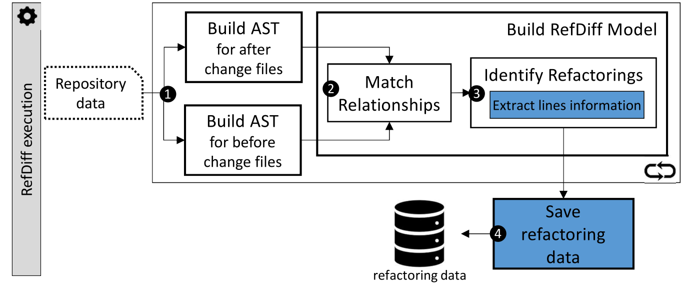

Fig. 3: An overview of our modified RefDiff (the blue boxes highlight our modifications)

RA-SZZ overview. Once we have our *refactoring data*, RA-SZZ is able to perform the refactoring checks in bug-fixes and bug-introducing changes. Figure 4 shows an overview of how RA-SZZ leverages the *refactoring data* to perform its analyses.

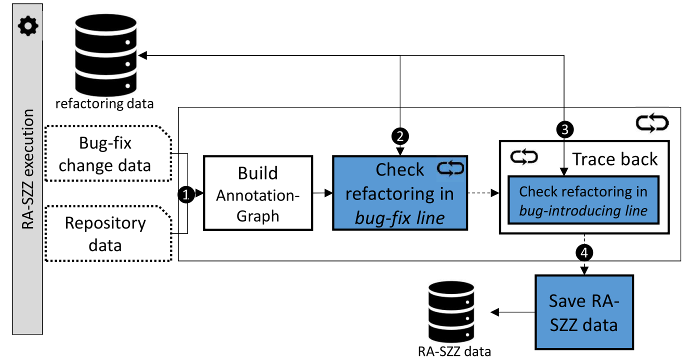

Fig. 4: Overview of RA-SZZ.

First, RA-SZZ builds an annotation graph of the modified parts of bug-fixes (*i.e.*, either modified or removed lines). At this step, the first check for refactoring changes occurs at the bug-fix level. By using the *refactoring data*, RA-SZZ checks whether the removed/modified lines are within a refactoring interval. In case the removes/modified lines are refactoring changes, RA-SZZ does not include them in the

annotation graph. Next, RA-SZZ starts the bug-introducing changes search. While searching for bug-introducing changes, RA-SZZ also checks whether these bug-introducing candidates contain refactoring changes. Finally, RA-SZZ stores the bug-introducing changes information in a database.

Refactoring in bug-introducing changes. While checking for refactoring changes in bug-introducing changes, RA-SZZ may perform additional steps. Figure 5 shows how RA-SZZ behaves when checking for refactoring changes in bug-introducing changes.

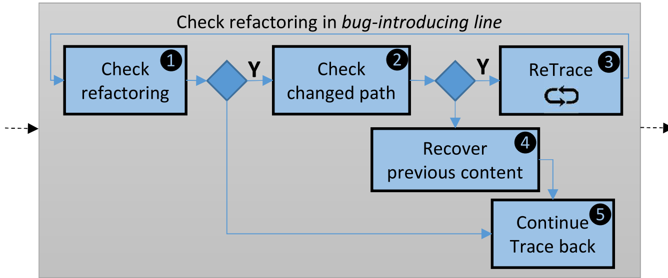

Fig. 5: Checking refactoring in bug-introducing changes

For each line in a bug-introducing change RA-SZZ checks whether there exists a refactoring. In case RA-SZZ finds a refactoring, the algorithm checks whether the path of the line was modified (*e.g.*, when the *move class* or *rename class* refactoring changes are performed). If that is the case, RA-SZZ has to *re-trace* the bug-introducing changes using the previous different path and repeat the refactoring checking process. Otherwise, RA-SZZ performs an attempt to recover the previous content of the line before the refactoring (*i.e.*, by using the *refactoring data*) in order to continue the bug-introducing changes search. This entire process is repeated until no refactoring changes are found.

RA-SZZ vs. MA-SZZ. Finally, we compare the results obtained for MA-SZZ with the results of RA-SZZ. We gradually compare our obtained results in a similar fashion as Kim *et al.* [15], *i.e.*, we assume that the results of RA-SZZ are more precise, since RA-SZZ handles refactoring changes. Hence, we assume that RA-SZZ outperforms MA-SZZ to the extent that RA-SZZ is able to detect refactoring changes in our dataset. Our comparisons allow us to measure how refactoring changes impact SZZ. First, we compare the number of bug-introducing lines that are produced by RA-SZZ and MA-SZZ. By doing so, we can compare how many false positives MA-SZZ produces by not considering refactoring changes. Finally, we also check how many change revisions RA-SZZ can trace back further when compared to MA-SZZ.

Result: *RA-SZZ decreases 20.8% of the bug-introducing lines that were flagged as bug-introducing by MA-SZZ (i.e., false positives).* Although this result does not prevent that RA-SZZ flags extra false positives when tracing more into history, due to other SZZ limitations not yet addressed, such as backout changes and initial code importing changes [17]. But at least RA-SZZ can avoid that those refactorings are erroneously flagged as bug-introducing changes thus contributing to improve the SZZ algorithm.

Table IV compares the MA-SZZ and RA-SZZ algorithms in terms of bug-introducing lines that are generated by each algorithm. We also present the decrease in the percentage of bug introducing lines after applying RA-SZZ to our studied systems. Our results show that RA-SZZ decreases the bug-introducing ratio from 14.8% (Tuscany system) to 34.6% (Mahout) in our studied systems. In the overall mean, there was a 20.8% decrease in the bug-introducing lines in our studied systems.

TABLE IV: The overall decrease of bug-introducing lines when comparing MA-SZZ and RA-SZZ.

System	MA-SZZ	RA-SZZ	% reduction
ActiveMQ	19,193	15,875	17.3%
Camel	20.366	16,291	20%
Derby	35,038	29,570	15.6%
Geronimo	45,744	33,987	25.7%
Haddop Common	39,887	28,491	28.6%
HBase	236,761	189,192	20.1%
Mahout	7,866	5,145	34.6%
OpenJPA	14,585	11,571	22.1%
Pig	18,807	15,367	18.3%
Tuscany	30,320	25,831	14.8%
Total	**468,840**	**371,320**	**20.8%**

In addition, Figure 6 shows the distributions of bug-introducing lines that are produced by MA-SZZ and RA-SZZ. We observe that the distribution of bug-introducing lines becomes less skewed when applying MA-SZZ. Our results suggest that RA-SZZ may be of considerable help in the software engineering community by reducing the noise in SZZ-generated data due to refactoring changes.

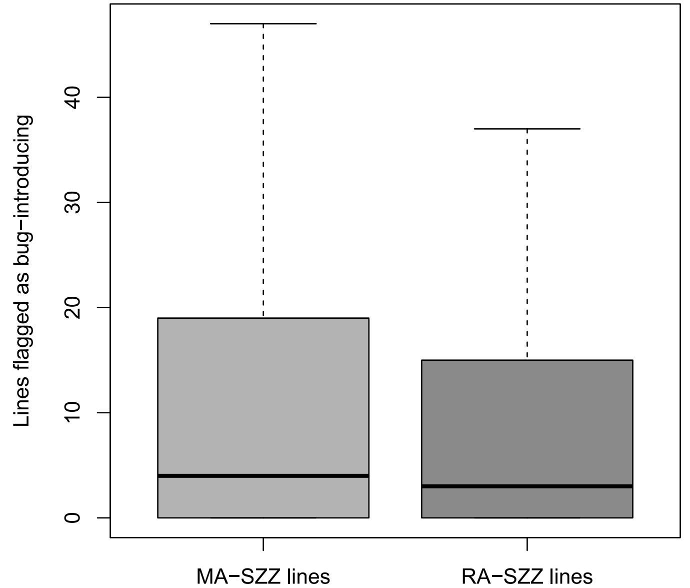

Fig. 6: Comparison between MA-SZZ and RA-SZZ for flagged bug-introducing lines per bug-fix

RA-SZZ traces the history further for 41% of the bug-introducing lines deemed as refactoring changes. We also

TABLE V: Summary about the additional changes that RA-SZZ can search further in history.

System	#Refac	Trace further		Additional changes		
		#	%	avg	max	min
ActiveMQ	646	332	51.4%	22	114	2
Camel	1,477	613	43.3%	13	112	2
Derby	766	317	41.4%	16	102	2
Geronimo	2,284	921	40.3%	13	88	2
Haddop	1,734	641	37%	18	219	2
HBase	2,973	1,194	40.2%	18	172	2
Mahout	195	89	45.6%	10	26	2
OpenJPA	357	108	30.3%	17	122	2
Pig	660	326	49.4%	9	54	2
Tuscany	1,301	517	39.7%	11	54	2
	12,333	5,058	41%	7	219	2

#refac: Number of refactored lines in bug-introducing changes; **trace further:** Number and Percentage of further trace attempts that RA-SZZ performs per system/line; **Additional changes:** the average, maximum and minimum of additional investigated revisions, respectively, per system/line.

analyze how many of the refactoring changes that are identified by RA-SZZ may be traced further in history. Table V shows that out of 12,333 bug-introducing lines, RA-SZZ can trace the history further for 5,058 lines (41%). We also observe that from these 41%, RA-SZZ analyzes a mean of 7 additional changes when tracing the history further (with the maximum of 219 changes)

Figure 7 shows an example of a refactoring change that RA-SZZ was able to trace further in history. When analyzing bug `HBASE-1607`, MA-SZZ flagged the line `88` of the file `Store.java` within the change `747672` as a bug-introducing line. The content of line `88` is a Java class declaration. Change `747672` consists of renaming the name of the class from `HStore` to `Store` (*i.e.*, a refactoring change of type *renaming*). In this example, RA-SZZ was able to identify that the file-path changed to `/hadoop/hbase/trunk/src/java/org/apache/hadoop/hbase/re-gionserver/HStore.java`. RA-SZZ was then able to trace further in history across 96 change revisions until change `630550`. At change `630550`, RA-SZZ identified another refactoring change of type *move class*. Finally, RA-SZZ traced further in history across 16 additional changes until change `611519`, which was flagged as bug-introducing. Without the ability of identifying refactoring changes, MA-SZZ would stop tracing at change `747672`.

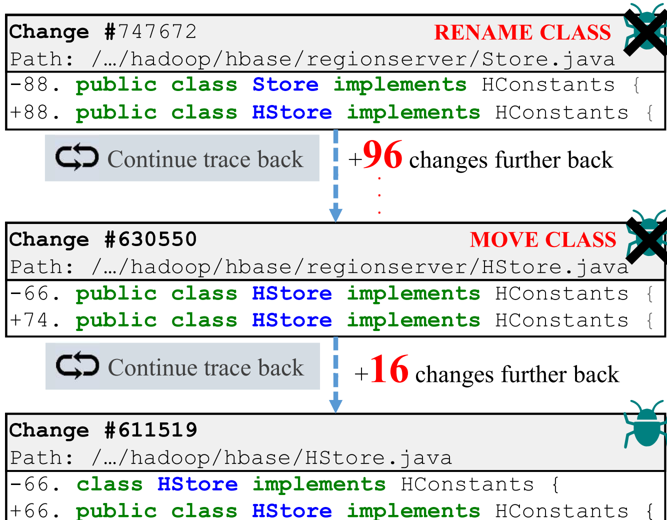

Fig. 7: Example of trace back process using RA-SZZ

> *We observe that RA-SZZ decreases 20.8% of the bug-introducing lines that were produced by MA-SZZ (false positives). In addition, RA-SZZ was able to trace the history further for 41% of the bug-introducing lines that were deemed as refactoring changes by RefDiff.*

C. RQ3. Can we find other change patterns that are not supported by RefDiff?

Approach: We perform a manual analysis of 365 bug-introducing lines that are produced by RA-SZZ. Our goal is to check whether there exist additional *equivalent changes* (see Definition 5) that RA-SZZ is erroneously flagging as bug-introducing. Our 365 bug-introducing lines data is a randomized sample from a population of 7,275 refactoring lines—that are flagged as bug-introducing by MA-SZZ—with a confidence level of 95% and confidence interval of 5%.

Result: ***We observe that 47.95% of our manually analyzed lines are related to equivalent changes that RA-SZZ should disregard as bug-introducing.*** We perform a manual analysis to identify why RA-SZZ was not able to trace the history further for 59% (7,275) of the bug-introducing lines that were deemed as refactoring changes (see RQ2). We manually investigate a sample of 365 bug-introducing lines (95% confidence level). We found that 52.05% of these lines are related to refactoring changes, such as *change class modifier*, *added parameter*, and *change method signature*. These changes could not be traced further by RA-SZZ because they do not have an associated line in the previous change (*e.g.*, these lines were added after the refactoring was performed). However, the remaining 47.95% should have been traced further. These 47.95% of lines were related to *equivalent changes* (see Definition 5). Table VI shows an overall statistics of the identified

TABLE VI: Change patterns identified during manual analysis

Change Patterns	#cases	% lines
Scope adjustment	10	5.75%
Multiple refactoring	58	15.89%
Extrac method adjustment	20	5.48%
Undetected additional refatoring	31	8.49%
Unncessary code	9	2.47%
Unncessary this	3	0.82%
Break if-statement	2	0.55%
Temporary variable	12	3.29%
Swap iteration style	7	1.92%
Change interface or superclass	4	1.1%
Code verbosity	3	0.82%
Cast and log changes	5	1.37%

equivalent changes. For example, there were 58 occurrences of multiple refactorings which represent 15.89% of all analysed bug-introducing lines. Furthermore, we present examples for some occurrences of the observed *equivalent changes* below. The data used in this manual analysis is available online to the interested reader.[4]

- *Change interface or superclass* [49]: Changes that modify interfaces and/or a superclass of existing classes, while maintaining system behaviour. For example, in change `492404` (Listing 1—2), a rename class (detected by RA-SZZ) and a change on the inheritance and realization relationships are performed (line `26`—Listing 2 is not detected by RA-SZZ):

```
−23. public class ABean_JPA extends ABean{
```

Listing 1: Change `492404`, `ABean_JPA.java`, OpenJPA

```
+26. public class ExampleABean_JPA extends ABean
     implements Cmp2Entity{
```

Listing 2: Change `492404`, `ExampleABean_JPA.java`, OpenJPA

- *Temporary variable addition/deletion* [49]: When the change only adds or removes temporary variables in source code, *e.g.*, change `201894` (Listing 3):

```
−186. taskReports.add(tip.getStatus());
−187. if (tip.getStatus().getRunState() !=
      TaskStatus.RUNNING) {
+186. TaskStatus status=tip.createStatus();
+187. taskReports.add(status);
+188. if (status.getRunState() != TaskStatus.
      RUNNING) {
```

Listing 3: Change `201894`, `TaskTracker.java`, HBase

- *Unnecessary code addition/deletion* [49]: This situation occurs when unnecessary code structures, (*e.g.*, the use of braces in an `if`-scope with just one statement) are removed or added in the source code. For example, in change `393035` (Listing 4):

```
−121. if (parentName == null) {
−122.     return null;
−123. }
+154. if (parentName == null)
+155.     return null;
```

Listing 4: Change `393035`, `FSDirectory.java`, Haddop

- *Swapping condition of an if-statement.* [50] This pattern occurs when breaking a conditional expression in multiple `if`-statements (or the reverse), *e.g.*, change `783632` (Listing 5):

```
−105. } else if (contentType.startsWith("text/
      plain")) {
+132. } else {
+133.     if (contentType.startsWith("text/plain"
      )) {
```

Listing 5: Change `783632`, `MailBinding.java`, Camel

[4]https://sites.google.com/view/refactoringszz/

- *Swap iteration style* [17]: This pattern occurs when the iteration style is changed. For example, change `109872` from Geronimo system moves the method `addRoleMappings` from `JettyXMLConfiguration` (Listing 6) to `SecurityContextBeforeAfter` (Listing 7) class. However, RA-SZZ could not recover the previous version of line `+117` (see Listing 7), since change `109872` also swaps the iteration style in this line during the move operation;

```
−337. Iterator realms = role.getRealms().values
      ().iterator();
−338. while (realms.hasNext()) {
```

Listing 6: Change `109872`, `JettyXMLConfiguration.java`, Geronimo

```
+117. for (Iterator realms = role.getRealms().
      values().iterator(); realms.hasNext();) {
```

Listing 7: Change `109872`, `SecurityContextBeforeAfter.java`, Geronimo

- *Scope adjustment during extraction*: This pattern occurs when a method is extracted from its original class to a different class. In some cases, it is also necessary to adjust the scope of the variables that are used in the original method call. For example, in change `630545`, the method `regionServerStartup` (which is not shown) was extracted from the `HMaster` class (Listing 8) to the `ServerManager` class (Listing 9). After being extracted, the new method cannot access the variable `closed` of its original class (line `110`—Listing 8) because such a variable had a private access in its original class (*i.e.*, `HMaster`, at line `87`—Listing 8). To solve this problem, an instance variable of type `HBase` is declared after the method extraction (line `69`—Listing 9). Finally, the variable `closed` may be accessed in the extracted method via instance variable. (line `118`—Listing 9).

```
−87. public class HMaster extends ... {
−88.   HMasterRegionInterface {
...
110.   volatile AtomicBoolean closed = new
       AtomicBoolean(true);
...
−712.      if (!closed.get()) {
```

Listing 8: Change `630545`, `HMaster.java`, HBase

```
+50. class ServerManager implements ... {
...
+69.   private HMaster master;
...
+118.      if (!master.closed.get()) {
```

Listing 9: Change `630545`, `ServerManager.java`, HBase

- *Undetected refactoring*: In some cases, we identify that an additional refactoring was not recognized by RA-SZZ. For example, a renamed variable (undetected) within the scope of a moved method (detected) (*e.g.*, change `393677`);

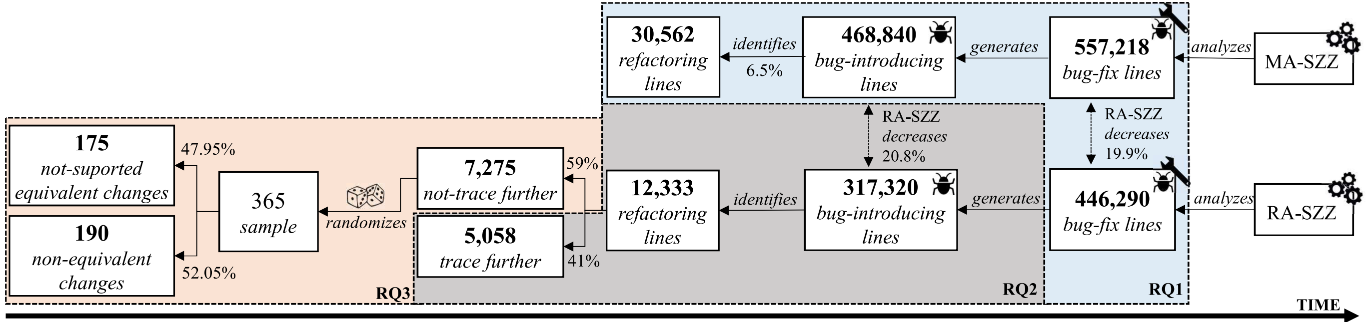

Fig. 8: Overview of the results obtained in this study for each RQ

- *Multiple refactoring changes that lead to a doubtful previous path*: This case occurs when several refactoring changes (with different origin paths) are found in the same line. An implementation of SZZ should be able to fork the refactoring change into different paths while tracing back further in history (*e.g.*, change `617338`).
- *Other equivalent changes*: Several other minor equivalent changes were identified, such as the addition/removal of the `this` keyword [49]. The initialization with a declaration of a global variable and deep semantics [49], among others.

Our results reveal that 47.95% of the analyzed bug-introducing lines are related to equivalent changes that RA-SZZ should not flag as bug-introducing. Among them we found 15.89% to represent multiple refactorings per line; 8.49% consist of undetected refactoring; 2.47% of addition/removal of unnecessary code etc.

Result Summary. Finally, Figure 8 shows an overview of the results obtained in this study for each RQ.

V. Threats to Validity

In this section, we report the threats to the validity of our study.

A. Internal Validity Threats

The interval validity is concerned with the causal conclusions that are drawn based on the analyses of a study. In this matter, we use the RefDiff tool to identify refactoring changes. RefDiff can only identify 13 types of refactoring out of the 63 types that are catalogued by Fowler [22]. In particular, this threat does not impair our study but limits our results. On the other hand, RefDiff may still generate false positives. However, a recent study shows that RefDiff obtained the best precision and recall for refactoring identification when compared to the other existing tools [23], which is the reason as to why we choose RefDiff to implement RA-SZZ.

B. Construct Validity Threats

The construct validity is concerned with the assumptions behind the measures of a study. In this concern, we assume that *refactoring changes* (Definition 3) should not introduce bugs (as defined by Fowler). Nevertheless, we acknowledge the research of Soares *et al.* [51] [52], which observes that some *refactoring attempts* may be defective in specific situations. However, the authors observed that less than 1% of the refactoring attempts were defective [52]. In addition, none of our studied samples are included in the situations described by Soares *et al.* [51].

C. External Validity Threats

The external threats to the validity are concerned with the ability to generalize the findings of a study to external populations (*i.e.*, in our case, software systems). In this matter, we study 10 Apache open source systems to identify how many refactoring changes are erroneously flagged as bug-introducing. The used projects comprise different application domains—*e.g.*, messaging queue, database, service-oriented architecture *etc*—and they are of different sizes. We acknowledge that we cannot generalize our observations to other different software systems. Nevertheless, the main goal of our study is not to reach generalizability, since the amount of refactoring may vary from project to project depending on the practices of the development team. Instead, our main goal is to highlight that current SZZ implementations generate inaccurate results by not handling refactoring changes.

VI. Conclusions

In this paper, we study the impact of refactoring changes on existing SZZ implementations. We use the RefDiff tool to identify 13 types of refactoring that are interleaved in bug-fixes, and that may be flagged as bug-introducing by SZZ. We also propose the *Refactoring Aware SZZ* (RA-SZZ), which is able to identify refactoring changes while tracing in the code history. Finally, we perform a manual analysis of the bug-introducing lines that could not be traced further by RA-SZZ. Among our main findings, we observe that:

- 6.5% of the bug-introducing lines identified by MA-SZZ are actually related to refactoring changes. Such an observation should be interpreted as a lower bound observation, since RefDiff can detect only 13 types of refactoring changes;
- 19.9% of the bug-introducing lines that are modified during bug-fixes are related to refactoring changes and should not be traced further by SZZ;
- Our RA-SZZ removes 20.8% of the lines that are flagged as bug-introducing changes by MA-SZZ. Moreover, RA-SZZ was able to trace further the history of 5,058 refactored lines;
- 47.95% out of the 365 manually analyzed bug-introducing lines contained *equivalent changes* that RA-SZZ could not trace the history further. Examples of these equivalent changes are undetected refactoring (8.49%), multiple refactoring per line (15.89%), addition/deletion of unnecessary code (2.47%), and swap iteration style (1.92%).

SZZ is often used to provide support to empirical studies. However, our study suggests that the results of previous research might be tainted by not considering refactoring changes when running the SZZ algorithm. We suggest that future work should replicate previous analyses that relied on SZZ to evaluate the extent of the impact that refactoring changes may have on the results of previous research. Additionally, we plan to evaluate RA-SZZ using the evaluation framework proposed by da Costa *et al.* [17]. Finally, we are working on building a ground truth to evaluate both existing and future SZZ implementations. This ground truth data is being constructed in partnership with a local software development company.

ACKNOWLEDGEMENTS

This work is partially supported by the National Institute of Science and Technology for Software Engineering (INES), CNPq grant 465614/2014-0, and National Council for Scientific and Technological Development, CNPq grants 459717/2014-6 and 312044/2015-1.

REFERENCES

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

[2] E. Giger, M. Pinzger, and H. C. Gall, "Comparing fine-grained source code changes and code churn for bug prediction," in *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*. ACM, 2011, pp. 83–92.

[3] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

[4] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 452–461.

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

[6] S. Shivaji, J. E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve bug prediction," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2009, pp. 600–604.

[7] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2012, pp. 171–180.

[8] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim, "Reducing features to improve code change-based bug prediction," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 552–569, 2013.

[9] H. Hata, O. Mizuno, and T. Kikuno, "Reconstructing fine-grained versioning repositories with git for method-level bug prediction," *IWESEP 10*, pp. 27–32, 2010.

[10] ——, "Bug prediction based on fine-grained module histories," in *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 2012, pp. 200–210.

[11] D. Di Nucci, F. Palomba, S. Siravo, G. Bavota, R. Oliveto, and A. De Lucia, "On the role of developer's scattered changes in bug prediction," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 241–250.

[12] A. T. Misirli, E. Shihab, and Y. Kamei, "Studying high impact fix-inducing changes," *Empirical Software Engineering*, vol. 21, no. 2, pp. 605–641, 2016.

[13] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 31–41.

[14] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, ACM. ACM, 2005, pp. 1–5. [Online]. Available: https://10.1145/1083142.1083147

[15] S. Kim, T. Zimmermann, K. Pan, E. James Jr *et al.*, "Automatic identification of bug-introducing changes," in *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*. IEEE, 2006, pp. 81–90.

[16] L. Prechelt and A. Pepper, "Why software repositories are not used for defect-insertion circumstance analysis more often: A case study," *Information and Software Technology*, vol. 56, no. 10, pp. 1377–1389, 2014.

[17] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering (TSE)*, vol. 43, no. 7, pp. 641–657, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2616306

[18] S. Davies, M. Roper, and M. Wood, "A preliminary evaluation of text-based and dependency-based techniques for determining the origins of bugs," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 2011, pp. 201–210.

[19] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 50.

[20] E. Murphy-Hill, C. Parnin, and A. P. Black, "How we refactor, and how we know it," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, 2012.

[21] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, "A multidimensional empirical study on refactoring activity," in *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*. IBM Corp., 2013, pp. 132–146.

[22] M. Fowler and K. Beck, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[23] D. Silva and M. T. Valente, "Refdiff: detecting refactorings in version histories," in *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017, pp. 269–279.

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

[25] M. Fischer, M. Pinzger, and H. Gall, "Analyzing and relating bug report data for feature tracking," in *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, vol. 3, 2003, p. 90.

[26] ——, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003, pp. 23–32.

[27] H. Hata, O. Mizuno, and T. Kikuno, "Fault-prone module detection using large-scale text features based on spam filtering," *Empirical Software Engineering*, vol. 15, no. 2, pp. 147–165, 2010.

[28] V. H. Nguyen and F. Massacci, "The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM, 2013, pp. 493–498.

[29] V. S. Sinha, S. Sinha, and S. Rao, "Buginnings: identifying the origins of a bug," in *Proceedings of the 3rd India software engineering conference*. ACM, 2010, pp. 3–12.

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

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[47] 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.

[39] 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.

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

[41] 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.

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

[43] 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*. ACM, 2014, pp. 1152–1157. [Online]. Available: https://10.1145/2554850.2555030

[44] 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.

[45] 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.

[46] 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.

[48] 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.

[49] Y. Jung, H. Oh, and K. Yi, "Identifying static analysis techniques for finding non-fix hunks in fix revisions," in *Proceedings of the ACM first international workshop on Data-intensive software management and mining*. ACM, 2009, pp. 13–18.

[50] D. Jackson, D. A. Ladd *et al.*, "Semantic diff: A tool for summarizing the effects of modifications." in *ICSM*, vol. 94, 1994, pp. 243–252.

[51] G. Soares, R. Gheyi, D. Serey, and T. Massoni, "Making program refactoring safer," *IEEE software*, vol. 27, no. 4, pp. 52–57, 2010.

[52] G. Soares, R. Gheyi, and T. Massoni, "Automated behavioral testing of refactoring engines," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 147–162, 2013.