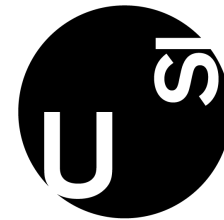


Evaluating SZZ Implementations Through a Developer-informed Oracle

Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano,
Gabriele Bavota, Michele Lanza, Rocco Oliveto



UNIVERSITÀ
DEGLI STUDI
DEL MOLISE



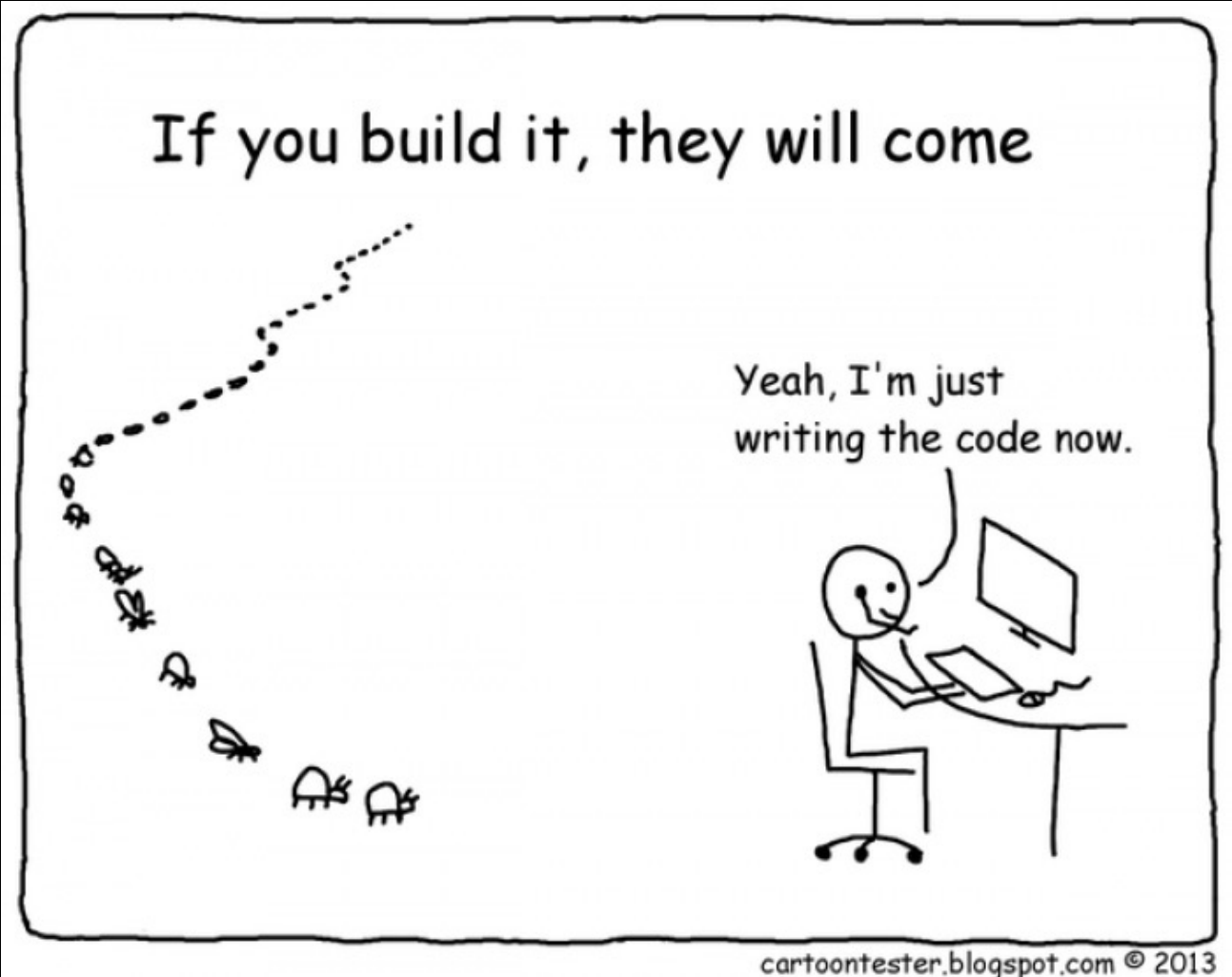
Università
della
Svizzera
italiana



43rd INTERNATIONAL
CONFERENCE ON SOFTWARE ENGINEERING
MAY 23-29, 2021
VIRTUAL (originally Madrid, Spain)

@ICSEconf





Where do bugs come from?



Understanding where bugs are introduced allows to...

Find out changes that can lead to a **problem**
and avoid them in future





Understanding where bugs are introduced allows to...

Estimate how much a program is **error-prone**





Understanding where bugs are introduced allows to...

Better allocate resources in **testing** activities



Śliwerski Zimmermann Zeller

@ MSR 2005

When Do Changes Induce Fixes?

(On Fridays.)

Jacek Śliwerski
International Max Planck Research School
Max Planck Institute for Computer Science
Saarbrücken, Germany
sliwers@mpi-sb.mpg.de

Thomas Zimmermann Andreas Zeller
Department of Computer Science
Saarland University
Saarbrücken, Germany
(tz, zeller}@acm.org

ABSTRACT

As a software system evolves, programmers make changes that sometimes cause problems. We analyze CVS archives for *fix-inducing changes*—changes that lead to problems, indicated by fixes. We show how to automatically locate fix-inducing changes by linking a version archive (such as CVS) to a bug database (such as BUGZILLA). In a first investigation of the MOZILLA and ECLIPSE history, it turns out that fix-inducing changes show distinct patterns with respect to their size and the day of week they were applied.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*corrections, version control*; D.2.8 [Metrics]: Complexity measures

General Terms

Management, Measurement

1. INTRODUCTION

When we mine software histories, we frequently do so in order to detect patterns that help us understanding the current state of the system. Unfortunately, not all changes in the past have been beneficial. Any bug database will show a significant fraction of problems that are reported some time after some change has been made.

In this work, we attempt to identify those *changes that caused problems*. The basic idea is as follows:

1. We start with a bug report in the bug database, indicating a *fixed problem*.
2. We extract the associated change from the version archive, thus giving us the *location* of the fix.
3. We determine the *earlier change* at this location that was applied before the bug was reported.

This earlier change is the one that *caused* the later fix. We call such a change *fix-inducing*.

What can one do with fix-inducing changes? Here are some potential applications:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR '05 May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00.

Which change properties may lead to problems? We can investigate which properties of a change correlate with inducing fixes, for instance, changes made on a specific day or by a specific group of developers.

How error-prone is my product? We can assign a *metric* to the product—on average, how likely is it that a change induces a later fix?

How can I filter out problematic changes? When extracting the architecture via co-changes from a version archive, there is no need to consider fix-inducing changes, as they get undone later.

Can I improve guidance along related changes? When using co-changes to guide programmers along related changes, we would like to avoid fix-inducing changes in our suggestions.

This paper describes our first experiences with fix-inducing changes. We discuss how to extract data from version and bug archives (Section 2) and how we link bug reports to changes (Section 3). In Section 4, we describe how to identify and locate fix-inducing changes. Section 5 shows the results of our investigation of the MOZILLA and ECLIPSE. It turns out that fix-inducing changes show distinct patterns with respect to their size and the day of week they were applied. Sections 6 and 7 close with related and future work.

2. WHAT'S IN OUR ARCHIVES?

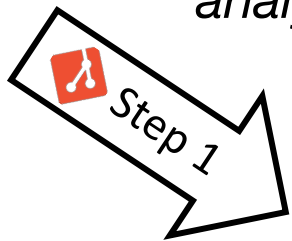
For our analysis we need all changes and all fixes of a project. We get this data from *version archives* like CVS and *bug tracking systems* like BUGZILLA.

A CVS archive contains information about changes: Who changed what, when, why, and how? A change δ transforms a revision r_1 to a revision r_2 by inserting, deleting, or changing lines. We will later investigate changes on the line level. Several changes $\delta_1, \dots, \delta_n$ form a *transaction* t if they were submitted to CVS by the same developer, at the same time, and with the same log message, i.e., they have been made with the same intention, e.g. to fix a bug or to introduce a new feature. As CVS records only individual changes to files, we group these to transactions with a *sliding time window* approach [12].

A CVS archive also lacks information about the *purpose* of a change: Did it introduce a new feature or did it fix a bug? Although it is possible to identify such reasons solely with log messages [7], we combine both CVS and BUGZILLA for this step because this increases the precision of our approach.

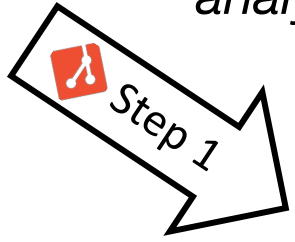
A BUGZILLA database collects bug reports that are submitted by a *reporter* with a *short description* and a *summary*. After a bug has been submitted, it is discussed by developers and users who provide additional *comments* and may create *attachments*. After the

*bug report
analysis*



SZZ in a nutshell

*bug report
analysis*



(A)
Bug-fixing
commit



(B)
git blame

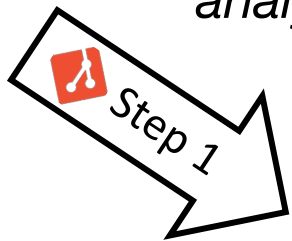


(C)
Buggy
commit



SZZ in a nutshell

*bug report
analysis*



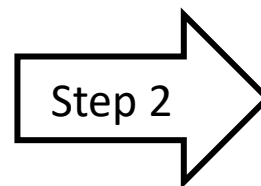
(A)
Bug-fixing
commit



(B)
git blame



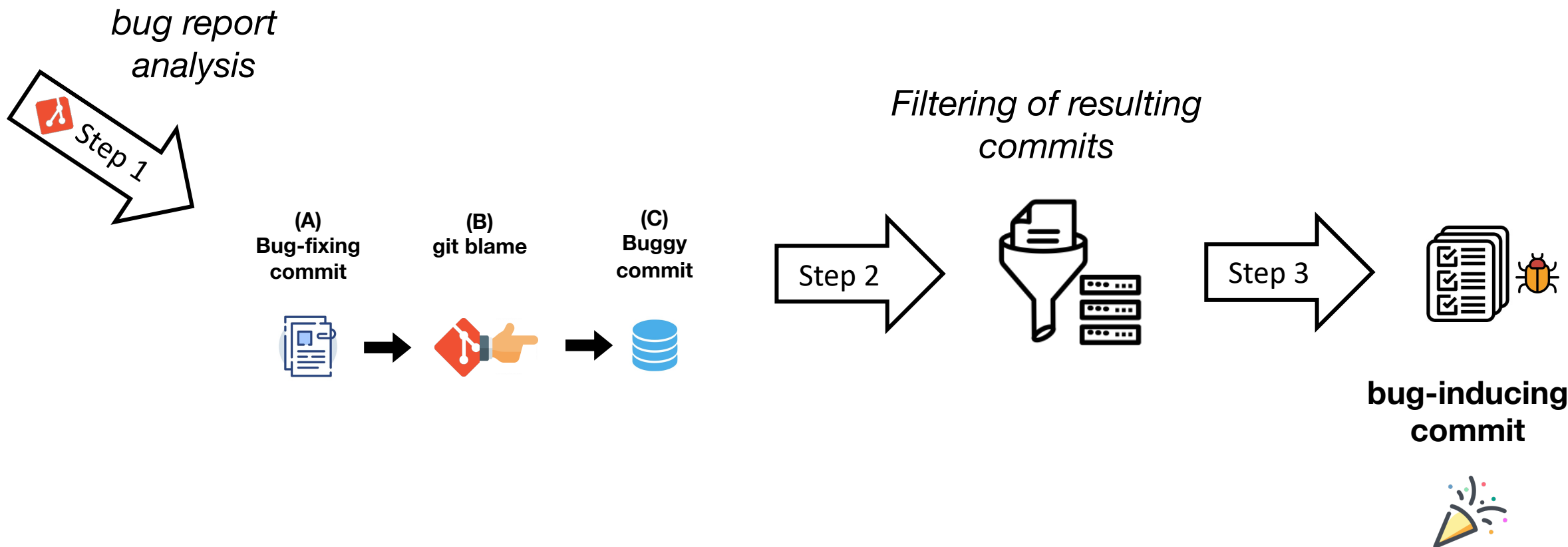
(C)
Buggy
commit



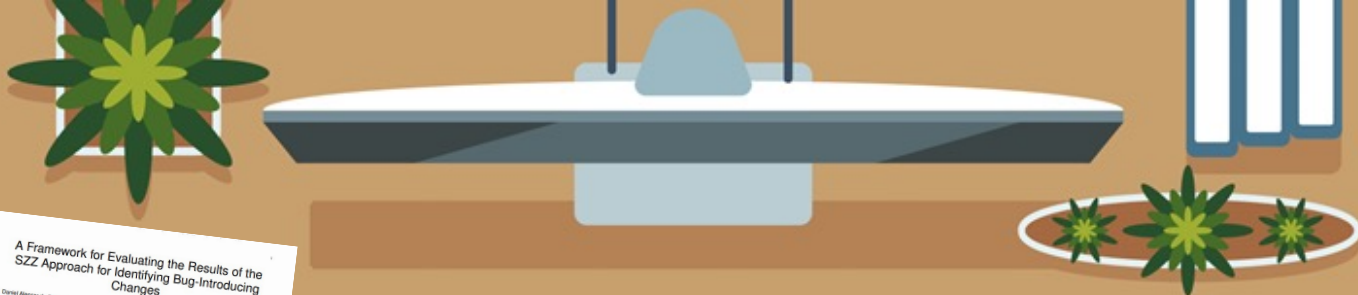
*Filtering of resulting
commits*



SZZ in a nutshell



SZZ in a nutshell



When Do Changes Induce Fixes?

Abstract: This paper presents a framework for identifying bug-introducing changes in software systems. The framework is based on the SZZ algorithm, which is used to identify changes in the code. The framework is evaluated using a set of benchmarks, and the results show that it is able to identify a high percentage of bug-introducing changes.

Authors: Jakob Stenroos, Thomas Zimmermann, Andreas Zeller, Daniel Menzies, and Minkyu Kim.

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

Abstract: This paper presents a framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. The framework is based on the SZZ algorithm, which is used to identify changes in the code. The framework is evaluated using a set of benchmarks, and the results show that it is able to identify a high percentage of bug-introducing changes.

Authors: Daniel Menzies, Jakob Stenroos, Thomas Zimmermann, Minkyu Kim, and Andreas Zeller.

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

Abstract: This paper presents a framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. The framework is based on the SZZ algorithm, which is used to identify changes in the code. The framework is evaluated using a set of benchmarks, and the results show that it is able to identify a high percentage of bug-introducing changes.

Authors: Daniel Menzies, Jakob Stenroos, Thomas Zimmermann, Minkyu Kim, and Andreas Zeller.

When Do Changes Induce Fixes?

Abstract: This paper presents a framework for identifying bug-introducing changes in software systems. The framework is based on the SZZ algorithm, which is used to identify changes in the code. The framework is evaluated using a set of benchmarks, and the results show that it is able to identify a high percentage of bug-introducing changes.

Authors: Jakob Stenroos, Thomas Zimmermann, Andreas Zeller, Daniel Menzies, and Minkyu Kim.

Software Evolution and Process

Abstract: This paper presents a framework for identifying bug-introducing changes in software systems. The framework is based on the SZZ algorithm, which is used to identify changes in the code. The framework is evaluated using a set of benchmarks, and the results show that it is able to identify a high percentage of bug-introducing changes.

Authors: Jakob Stenroos, Thomas Zimmermann, Andreas Zeller, Daniel Menzies, and Minkyu Kim.

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

Abstract: This paper presents a framework for identifying bug-introducing changes in software systems. The framework is based on the SZZ algorithm, which is used to identify changes in the code. The framework is evaluated using a set of benchmarks, and the results show that it is able to identify a high percentage of bug-introducing changes.

Authors: Jakob Stenroos, Thomas Zimmermann, Andreas Zeller, Daniel Menzies, and Minkyu Kim.

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

Abstract: This paper presents a framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. The framework is based on the SZZ algorithm, which is used to identify changes in the code. The framework is evaluated using a set of benchmarks, and the results show that it is able to identify a high percentage of bug-introducing changes.

Authors: Daniel Menzies, Jakob Stenroos, Thomas Zimmermann, Minkyu Kim, and Andreas Zeller.

Different SZZ variants proposed

There is a problem

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, Software engineering.

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)).

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

- 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@ppgic.ufrn.br, {uira,roberha}@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

Evaluating and comparing the SZZ variants

Da Costa et al. @ TSE 2016

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

Da Costa, S. Shrivastava, M. S. Hameed, A. E. Hassan

Abstract—Software engineering (SE) researchers have used the SZZ approach for identifying bug-introducing changes (BICs) in software repositories. However, the SZZ approach has been made to evaluate its results. Such an evaluation is a challenging task because the ground truth is not readily available. In this paper, we acknowledge 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 reason 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 detector, Software repository mining, Software engineering

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 Slivovski, 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)).

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 *not-in-time* (NIT) 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

- D. da Costa, D. Ribeiro, and R. Coelho are with the Department of Informatics and Applied Mathematics (DIAMAp), Federal University of Rio Grande do Norte, Brazil.
E-mail: {daucosta@lappos.ufrn.br, {dribeiro@lappos.ufrn.br}
- Sumeet Shrivastava is with the Department of Electrical and Computer Engineering, McGill University, Canada.
E-mail: sumee@ece.mcgill.ca
- Dong Sheng is with the Department of Computer Science and Software Engineering, Concordia University, Canada.
E-mail: sheng@atcs.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

Small datasets used for evaluation Evaluating and comparing the SZZ variants

Da Costa et al. @ TSE 2016

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

Small datasets used for evaluation

Evaluating and comparing the SZZ variants

Validation manually performed by researchers

Da Costa et al., 2016

been made to evaluate its results. Such an evaluation is a challenging task because the ground truth is not readily available. In this paper, 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 reason 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 detector, Software repository mining, Software engineering

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 prior bug-introducing changes was proposed by Shih et al. [3]. This approach, known as SZZ, locates bug-fixing changes by tracing back through the code history to find when the changed code was introduced (i.e., the supposed bug-introducing change(s)).

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 instance, one way to avoid the introduction of bugs

is to avoid introducing quality attributes such as long change lists [12]. Other studies have used SZZ to identify the nature 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 manually inspect a large sample for analysis. While the prior manual analysis provides valuable insights, the domain experts (e.g., the bug fixers or testers) were not consulted. These experts could offer insight 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

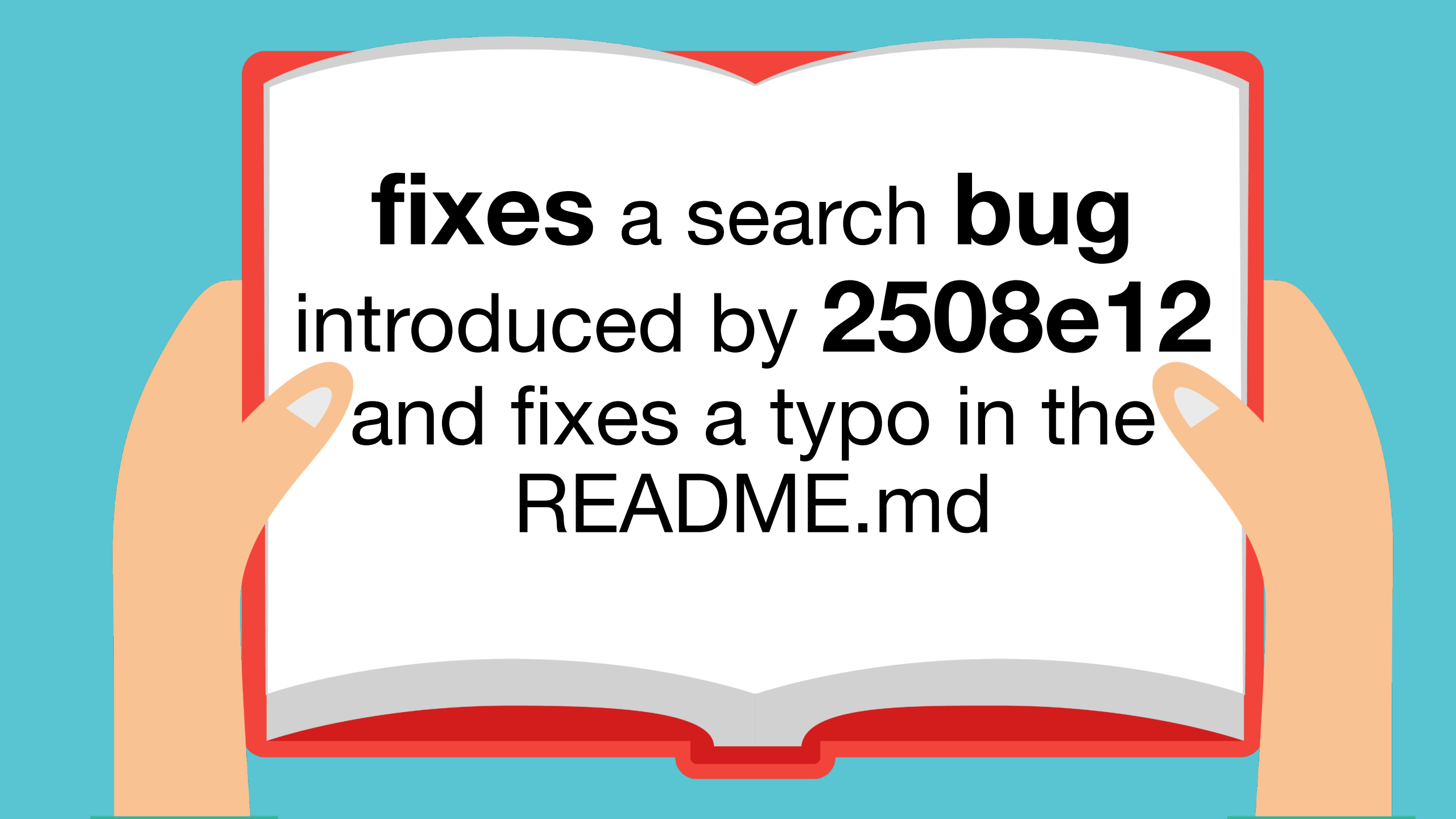
- D. da Costa, D. Kulkarni, and R. Lachin are with the Department of Informatics and Applied Mathematics (DIAMAP), Federal University of Rio Grande do Norte, Brazil.
E-mail: {dada@disf.ufrn.br, dachin@disf.ufrn.br, dachin@disf.ufrn.br}
- Steve McIntosh is with the Department of Electrical and Computer Engineering, McGill University, Canada.
E-mail: steve.mcintosh@mcgill.ca
- Danyang Sheng is with the Department of Computer Science and Software Engineering, Concordia University, Canada.
E-mail: sheng@atena.concordia.ca
- Ahmed E. Hassan is with the Software Analysis and Intelligence Lab (SAIL), School of Computing, Queen's University, Canada.
E-mail: ahmedh@post.queensu.ca

Da Costa et al. @ TSE 2016

Define a dataset validated by the developers



The way



fixes a search **bug**
introduced by **2508e12**
and fixes a typo in the
README.md

Developer-informed dataset

2011



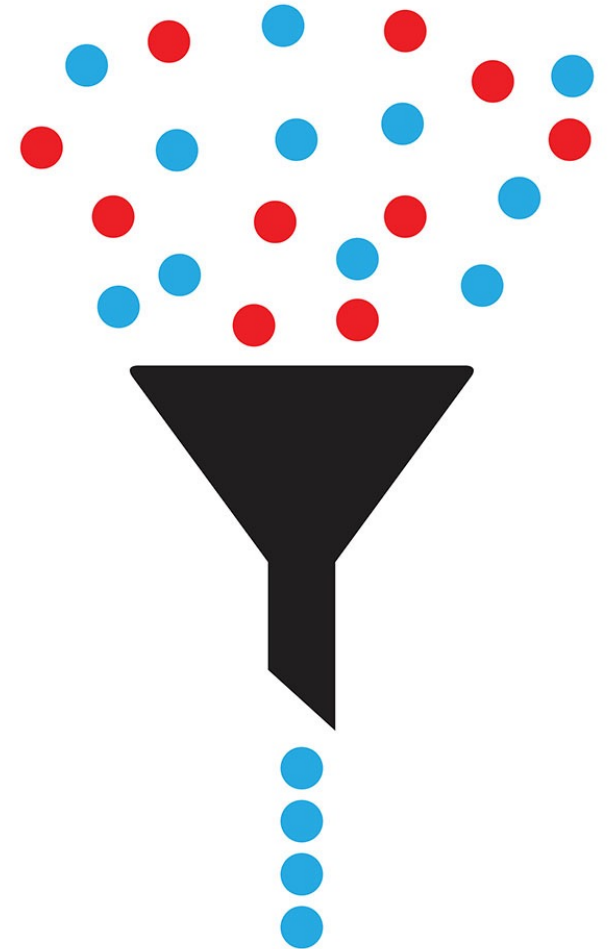
2020

GitHub

Mining of commits

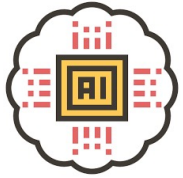


keyword-based filter

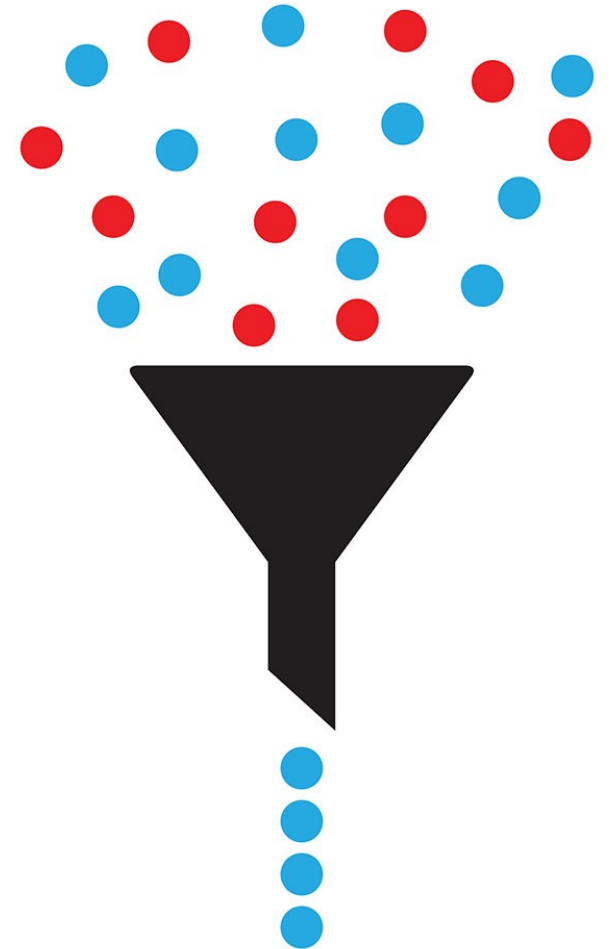




keyword-based filter

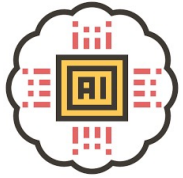


AI-powered syntax analysis





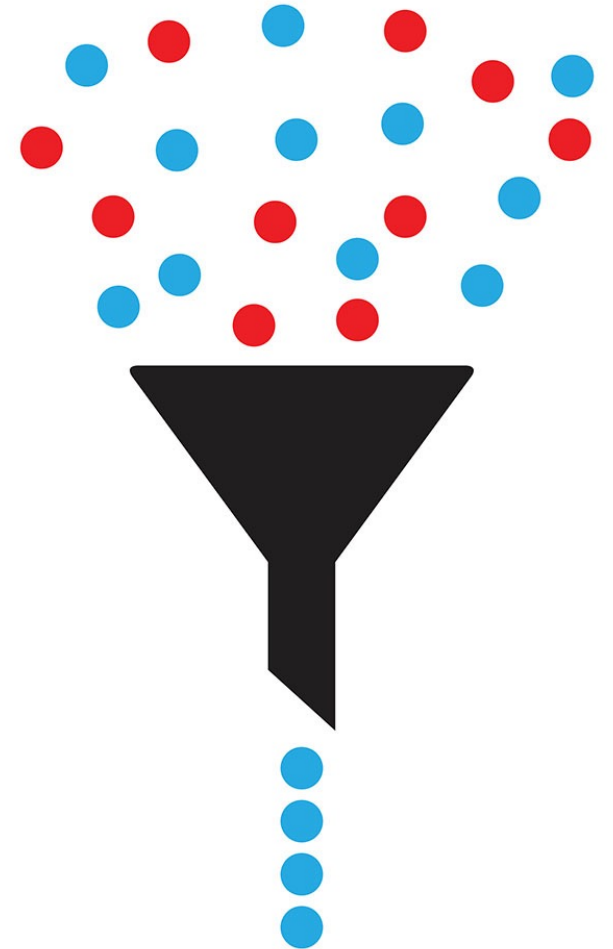
keyword-based filter



AI-powered **syntax analysis**



duplicate commits removal





**False
positives**



**Bug report
data**



Manual validation

Commit
message

fixes **#1740** quote pov-ray binary on windows
this fixes a bug introduced by #3523741...

URL

<https://tracker.freecadweb.org/view.php?id=1740>

**Date when the
issue is reported**

Date Submitted
2014-09-10 22:57

Bug report data

Analyzed commits:

19,6M

Extracted commits:

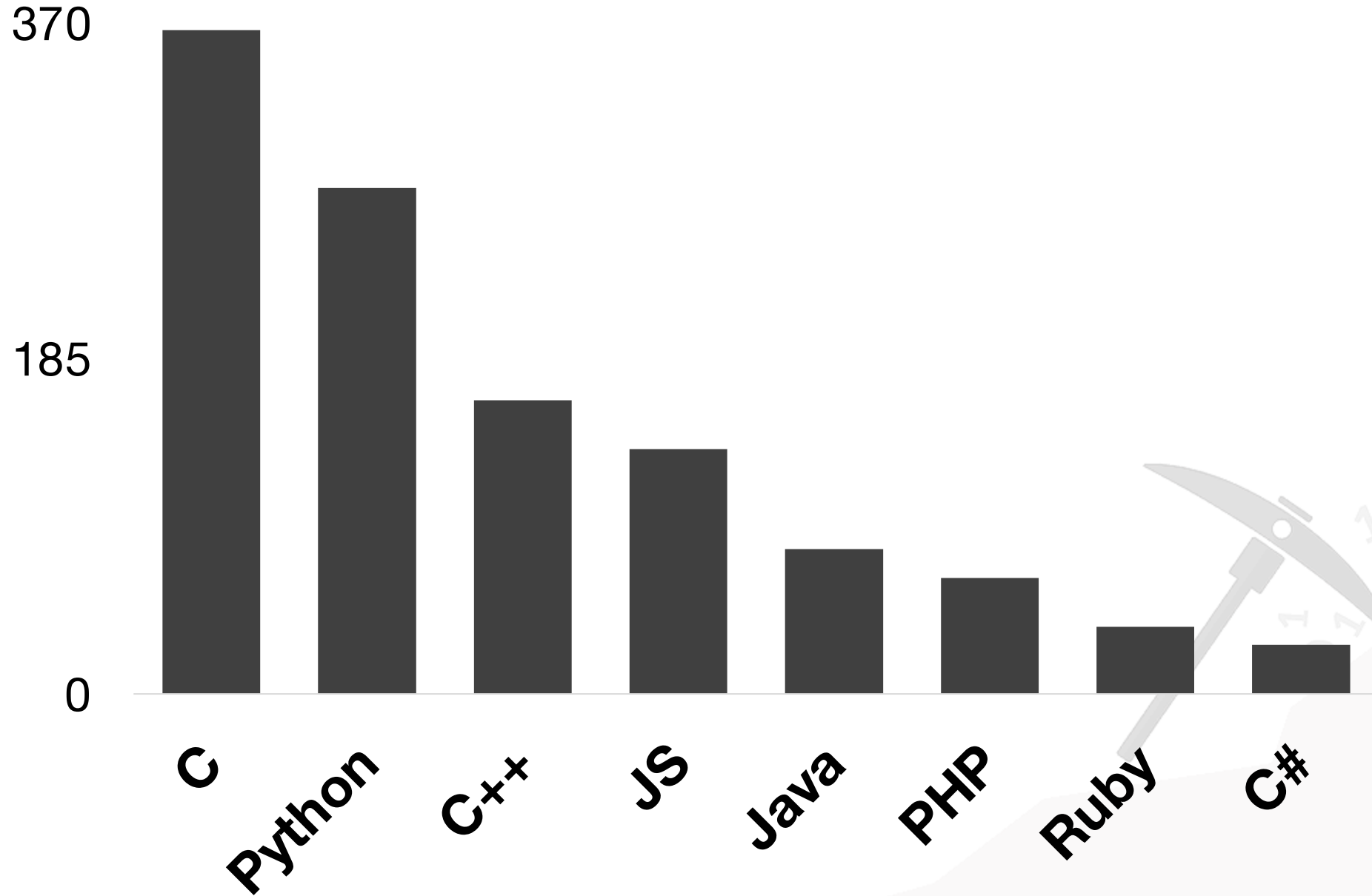
3,6k

After manual validation:

1,9k



Top programming languages



Final number of commits:

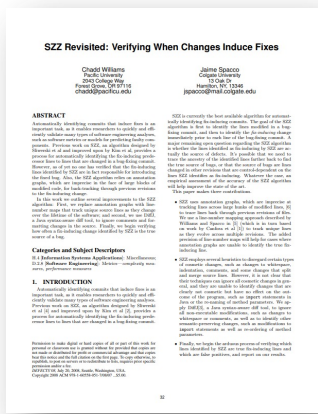
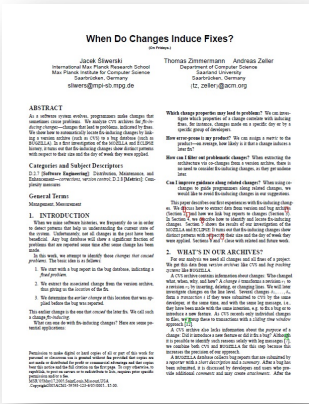
1,1k

Commits with issue report:

129



**How do different variants of SZZ
perform in identifying
bug-inducing changes?**

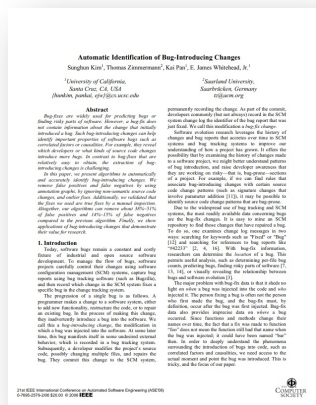


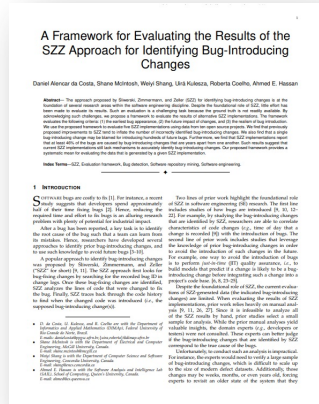
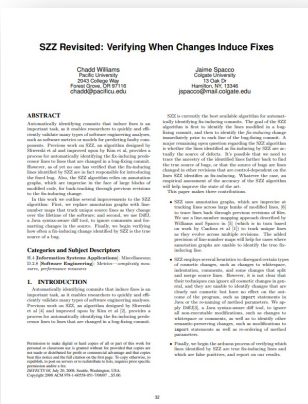
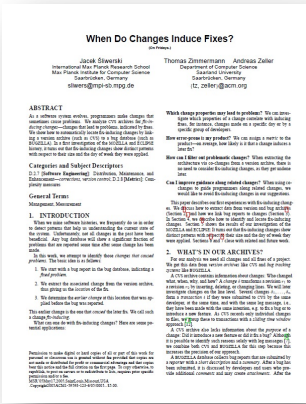
B-SZZ
Slivewski et al. @ MSR 2005

DJ-SZZ
Williams and Spacco @ ISSTA 2008

AG-SZZ
Kim et al. @ ASE 2006

R-SZZ e L-SZZ
Davies et al. @ JSE 2013





B-SZZ

Śliwerski et al. @ MSR 2005

DJ-SZZ

Williams and Spacco @ ISSTA 2008

MA-SZZ

Da Costa et al. @ TSE 2016

AG-SZZ

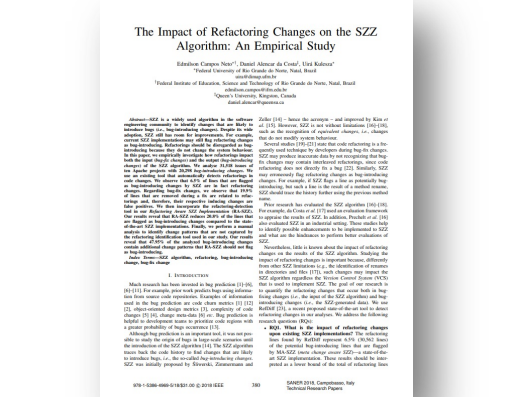
Kim et al. @ ASE 2006

R-SZZ e L-SZZ

Davies et al. @ JSE 2013

RA-SZZ

Neto et al. @ SANER 2018





SZZ Unleashed
(DJ-SZZ)



PyDriller
(AG-SZZ)



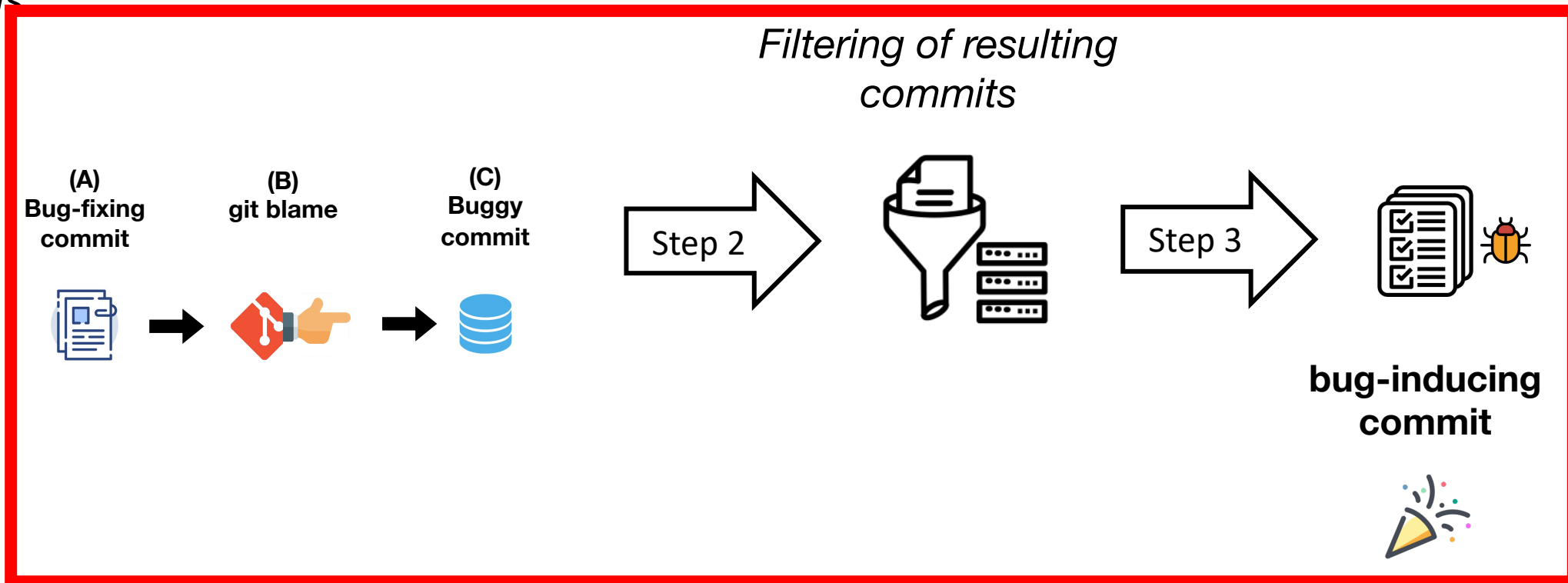
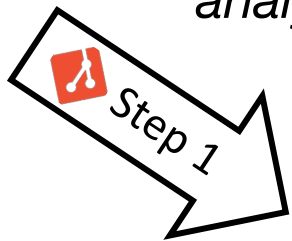
OpenSZZ
(B-SZZ)



RA-SZZ
(RA-SZZ)

Open-Source implementations

bug report
analysis



Our experiment



Precision

0.66 (R-SZZ)

Recall

0.72 (SZZ@UNL)

F1-score

0.61 (R-SZZ)

Results



Precision

0.66 (R-SZZ)

0.09 (SZZ@UNL)

Recall

0.72 (SZZ@UNL)

0.19 (SZZ@OPN)
Java only

F1-score

0.61 (R-SZZ)

0.16 (SZZ@UNL)

Results



Qualitative Analysis

What have we learned?





“ The **buggy line** is not always impacted in the **bug-fix** ,”

Lesson 1

**“ SZZ is sensible to
history rewritings ,”**



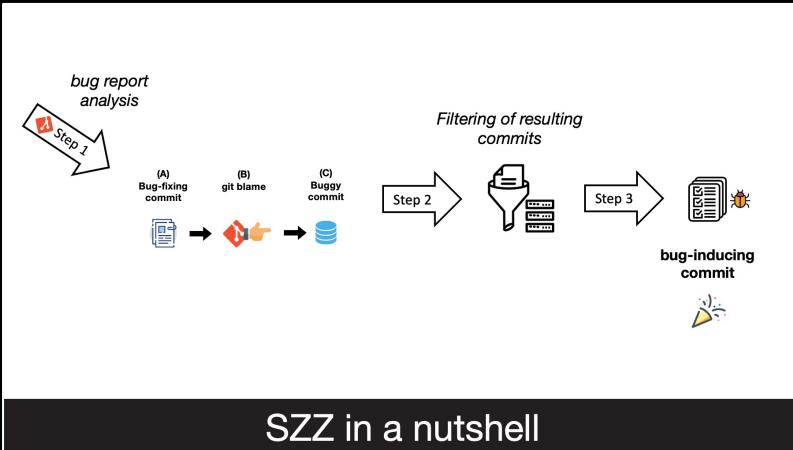
Lesson 2



“ Looking at the
big picture in
code changes ,”

Lesson 3

Summary



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

Small datasets used for evaluation

Evaluating and comparing the SZZ variants

Validation manually performed by researchers

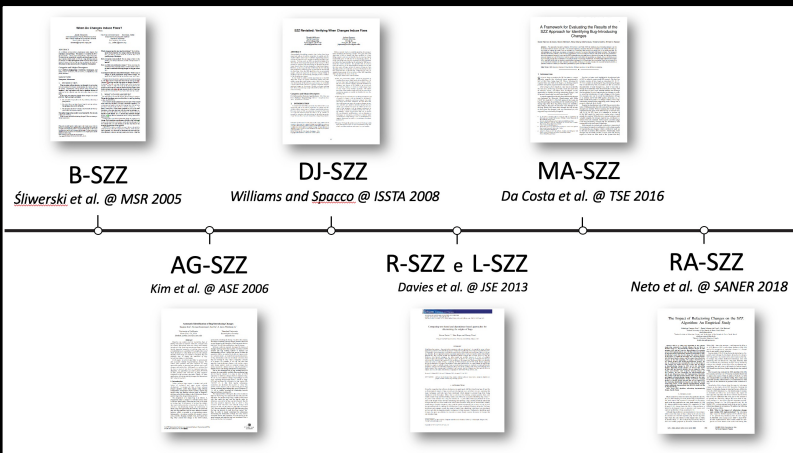
Da Costa et al. @ TSE 2016

keyword-based filter

AI-powered syntax analysis

duplicate commits removal

3 Heuristic approach



	✓	✗
Precision	0.66 (R-SZZ)	0.09 (SZZ@UNL)
Recall	0.72 (SZZ@UNL)	0.19 (SZZ@OPN) Java only
F1-score	0.61 (R-SZZ)	0.16 (SZZ@UNL)

Results

What have we learned?

Take a look at our SZZ implementation!

<https://github.com/grosa1/pyszz>

