Highlights

A Comprehensive Evaluation of SZZ Variants Through a Developer-informed Oracle

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

- Developer-informed dataset for the evaluation of SZZ, composed of 2,304 instances;
- R-SZZ is the best performing variant of SZZ, among the 9 evaluated;
- Heuristics to process added lines, based on Definition-Use chains, improve SZZ;
- Filtering revert commits provides a small but concrete improvement to SZZ;

A Comprehensive Evaluation of SZZ Variants Through a Developer-informed Oracle

Giovanni Rosa[a], Luca Pascarella[b], Simone Scalabrino[a], Rosalia Tufano[b], Gabriele Bavota[b], Michele Lanza[b], Rocco Oliveto[a]

[a] *STAKE Lab @ University of Molise, Italy*
[b] *Software Institute @ USI Università della Svizzera Italiana, Switzerland*

Abstract

Automatically linking bug-fixing changes to bug-inducing ones (BICs) is one of the key data-extraction steps behind several empirical studies in software engineering. The SZZ algorithm is the *de facto* standard to achieve this goal, with several improvements proposed over time. Evaluating the performance of SZZ implementations is, however, far from trivial. In previous works, researchers (i) manually assessed whether the BICs identified by the SZZ implementation were correct or not, or (ii) defined oracles in which they manually determined BICs from bug-fixing commits. However, researchers have limited knowledge of the studied systems, so their evaluation might be either biased or simply erroneous. Ideally, the original developers should be involved in defining an oracle to evaluate SZZ implementations. We propose a methodology to define a "developer-informed" oracle for evaluating SZZ implementations. We use Natural Language Processing (NLP) to identify bug-fixing commits in which developers explicitly reference the commit(s) that introduced the fixed bug. A manual filtering step followed this to ensure the oracle's quality and accuracy. We use the built oracle to extensively evaluate existing SZZ variants defined in the literature. We also introduce and evaluate two variants aimed at addressing two weaknesses we observed in state-of-the-art implementations.

Keywords: SZZ, Defect Prediction, Empirical Study

Email addresses: `giovanni.rosa@unimol.it` (Giovanni Rosa), `luca.pascarella@usi.ch` (Luca Pascarella), `simone.scalabrino@unimol.it` (Simone Scalabrino), `rosalia.tufano@usi.ch` (Rosalia Tufano), `gabriele.bavota@usi.ch` (Gabriele Bavota), `michele.lanza@usi.ch` (Michele Lanza), `rocco.oliveto@unimol.it` (Rocco Oliveto)

Preprint submitted to Journal of Systems and Software

1. Introduction

The revision history of long-lived software projects features plenty of *corrective changes*, *i.e.,* modifications aimed at fixing bugs. For each corrective change – or *bug-fixing commit* – it exists a non-empty set of commits that introduced the addressed bug. While the performed bug-fixing activity is often explicitly documented in the commit message, the same obviously does not happen for the commits introducing bugs. Therefore, while such a linking can be useful to conduct empirical studies on the characteristics of changes that introduce bugs (Bavota and Russo, 2015; Tufano et al., 2017; Aman et al., 2019; Chen and Jiang, 2019) or to validate defect prediction techniques (Hata et al., 2012; Tan et al., 2015; Pascarella et al., 2019; Yan et al., 2020; Fan et al., 2019), it is challenging to establish.

In 2005, Śliwerski et al. (2005) proposed the SZZ algorithm to address such a problem. Given a bug-fixing commit C_{BF}, the SZZ algorithm identifies a set of commits that likely introduced the error fixed in C_{BF}. These commits are named "bug-inducing" commits. In a nutshell, SZZ identifies the last change (commit) to each source code line changed in C_{BF} (*i.e.,* changed to fix the bug). This is done by relying on the annotation/blame feature of versioning systems. The identified commits are considered as the ones that later on triggered the bug-fixing commit C_{BF}.

Since the original work was published, several researchers have proposed variants of the original algorithm, with the goal of improving its accuracy (Kim et al., 2006; Williams and Spacco, 2008a; Davies et al., 2014; Da Costa et al., 2016; Neto et al., 2018, 2019). For example, a limitation of the original SZZ algorithm is that it considers changes to code comments and whitespaces like any other change. Therefore, if a comment is modified in C_{BF}, the latest change to such a comment is mistakenly considered as a BIC. Therefore, Kim et al. (2006) introduced a variant which ignores such changes. Similarly, other variants ignore non-executable statements (*e.g.,* `import` statements) (Williams and Spacco, 2008a), meta-changes (*e.g.,* merge commits) (Da Costa et al., 2016), and refactoring operations (*e.g.,* variable renaming) (Neto et al., 2018, 2019).

Despite the growth of the number of SZZ variants introduced to achieve higher and higher levels of accuracy, da Costa *et al.* highlighted (Da Costa et al., 2016) that the performed accuracy evaluations mostly rely on manual analysis performed on the output of the proposed SZZ variants (Śliwerski et al., 2005; Kim et al., 2006; Williams and Spacco, 2008a; Davies et al., 2014). Researchers themselves usually perform such a validation, despite not being the original developers of the studied systems and, thus, not always having the knowledge needed to correctly identify the bug introducing commit. Other researchers, instead, defined a ground truth to evaluate the performance of their variants (Neto et al., 2019). Also in these cases, however, researchers completed such a task. Therefore, there is a clear need for oracles defined by exploiting the knowledge of people who worked on the system (Da Costa et al., 2016). Still, directly involving them to manually evaluate a large sample of BICs is impractical (Da Costa et al., 2016).

In this paper, we extend our ICSE'21 paper (Rosa et al., 2021) in which we addressed this problem by introducing a methodology to build a "developer-informed" oracle for the evaluation of SZZ variants. To explain the core idea, let us take as an example commit `31063db` from the `mrc0mmand/systemd` GitHub project, accompanied by a commit message saying: "*sd-device: keep escaped strings in DEVLINK= property. This fixes a bug introduced by 87a4d41. Fixes systemd#17772*". The developer fixing the bug is explicitly documenting the commit that introduced such a bug. Based on this observation, we defined strict NLP-based heuristics to automatically detect messages of bug-fixing commits in which developers explicitly reference the commit(s) that introduced the fixed bug. We call such commits "referenced bug-fixing commits". It is worth noting that such a process is not meant to be exhaustive, *i.e.,* we do not aim at finding *all* the referenced bug-fixing commits. Instead, we mainly aim at obtaining a high-quality dataset of commits that are very likely induced a bug-fix.

We used our NLP-based heuristics to filter all the commits done on GitHub public repositories between March 2011 and the end of January 2021 by relying on GitHub Archive (Grigorik, 2012), a public service which archives all public events occurred on GitHub. Compared to our previous paper, we have analyzed 9 additional months of GitHub events. From a set of 24,042,335 (*i.e.,* 4.4M more than our previous paper), our heuristics identified 4,585 possible referenced bug-fixing commits. To further increase the quality of our dataset, we manually validated such commits, aiming at verifying whether the commit message was clearly documenting the bug-inducing commit. Besides, we annotated possible issues from the issue-tracker explicitly referenced by developers since such a piece of information is exploited by some SZZ variants. In the end, we obtained a dataset including 2,304 referenced bug-fixing commits (*i.e.,* 22% more than our previous paper), with 212 also including information about the fixed issue(s).

After manually analyzing cases in which all SZZ variants failed to detect the correct BIC, we found two main limitations of existing approaches: (i) they do not take into account added lines, but only deleted lines, since those are the ones on which it is possible to use the `blame` command; (ii) they are confused by revert commits, which reset previous changes not allowing SZZ to find the actual BICs. Therefore, we introduce two novel heuristics that aim at overcoming such limitations. In the first, given the set of added lines, we detect the lines directly affected by them by relying on Definition-Use chains. Then, we detect changes that introduced such lines. In the second heuristic, we detect revert commits by using NLP-based heuristics, and we discard them when they are selected as candidate BICs.

We tested the new heuristics we introduced in isolation, to understand to what extent they affect the accuracy. Our results show that the Definition-Use heuristic allows finding BICs in cases in which other SZZ variants do not work. On the other hand, the revert-ignoring heuristic provides a small advantage in terms of precision (+1%), without paying any price in terms of recall.

To summarize, the novel contributions provided in this paper with respect to our previous paper (Rosa et al., 2021) are the following:

1. We extended the dataset by including 9 additional development months on GitHub, resulting in 4.4M additional commits analyzed and 421 new instances in the final dataset;
2. We replicated our experiments on the new dataset;
3. Based on our finding, we introduced and evaluated two new SZZ variants, showing that both of them slightly improve the effectiveness of SZZ.

2. Background and Related Work

We start by presenting several variants of the SZZ algorithm (Śliwerski et al., 2005) proposed in the literature over the years. Then, we discuss how those variants have been used in SE research community.

2.1. SZZ Variants

Table 1 presents the SZZ variants proposed in the literature. We report for each of them its name and reference, the approach it builds upon (*i.e.,*

Approach name	**Reference**	**Based on**	**Used by**	**Oracle type**	**# Projects**	**# Bug Fixes**
B-SZZ	Śliwerski et al. (2005)		(Palomba et al., 2018; Pascarella et al., 2019; Çaglayan and Bener, 2016; Wen et al., 2016; Posnett et al., 2013; Kim et al., 2008; Tan et al., 2015; Kononenko et al., 2015; Wehaibi et al., 2016; Lenarduzzi et al., 2020a)	//	//	//
AG-SZZ	Kim et al. (2006)	B-SZZ	(Tufano et al., 2017; Bernardi et al., 2018; Hata et al., 2012; Rahman et al., 2011; Eyolfson et al., 2014; Misirli et al., 2016; Canfora et al., 2011; Prechelt and Pepper, 2014; Bird et al., 2009a)	Manually defined (researchers)	2	301
DJ-SZZ	Williams and Spacco (2008a)	AG-SZZ	(Marinescu et al., 2014; Borg et al., 2019; Bavota and Russo, 2015; Tóth et al., 2016; Fan et al., 2019; Karampatsis and Sutton, 2020; Rodríguez-Pérez et al., 2020, 2018)	Manually defined (researchers)	1	25
L-SZZ & R-SZZ	Davies et al. (2014)	AG-SZZ	(Da Costa et al., 2016)	Manually defined (researchers)	3	174
MA-SZZ	Da Costa et al. (2016)	AG-SZZ	(Fan et al., 2019; Neto et al., 2018, 2019; Tu et al., 2020; Aman et al., 2019; Chen and Jiang, 2019)	Automatically computed metrics	10	2,637
RA-SZZ	Neto et al. (2018)	MA-SZZ	(Fan et al., 2019; Neto et al., 2018; Yan et al., 2020)	Manually defined (researchers)	10	365
RA-SZZ*	Neto et al. (2019)	RA-SZZ	None	Manually defined (researchers)	10	365
A-SZZ	Sahal and Tosun (2018)	B-SZZ	None	Manually defined (researchers)	2	251

Table 1: Variants of the SZZ algorithm. For each one, we specify (i) the algorithm on which it is based, (ii) references of works using it, (iii) the oracle used in the evaluation (how it was built, number of projects and bug fixes considered).

the starting point on which the authors provide improvements), some references to works that used it, and information about the oracle used for the evaluation. Specifically, we report how the oracle was built and the number of projects/bug reports considered.

All the approaches that aim at identifying bug-inducing commits (BICs) rely on two elements: (i) the revision history of the software project, and (ii) an issue tracking system (optional, needed only by some SZZ implementations).

The original SZZ algorithm was proposed by Śliwerski et al. (2005) (we refer to it as B-SZZ, following the notation provided by Da Costa et al. (2016)). B-SZZ takes as input a bug report from an issue tracking system, and tries to find the commit that fixes the bug. To do this, B-SZZ uses a two-level confidence level: *syntactic* (possible references to the bug ID in the issue tracker) and *semantic* (*e.g.,* the bug description is contained in the commit message). B-SZZ relies on the CVS `diff` command to detect the lines changed in the fix commit and the `annotate` command to find the commits in which the lines were modified. Using this procedure, B-SZZ determines the *earlier* change at the location of the fix. Potential bug-inducing commits performed after the bug was reported are always ignored.

Kim et al. (2006) noticed that B-SZZ has limitations mostly related to formatting/cosmetic changes (*e.g.,* moving a bracket to the next line). Such changes can deceive B-SZZ: B-SZZ (i) can report as BIC a revision which only changed the code formatting, and (ii) it can consider as part of a bug-fix a formatting change unrelated to the actual fix. They introduce a variant (AG-SZZ) in which they used an annotation graph, a data structure associating the modified lines with the containing function/method. AG-

SZZ also ignores the cosmetic parts of the bug-fixes to provide more precise results.

Williams and Spacco (2008a) improved the AG-SZZ algorithm in two ways: first, they use a line-number mapping approach (Williams and Spacco, 2008b) instead of the annotation graph introduced by Kim et al. (2006); second, they use DiffJ (Pace, 2007), a Java syntax-aware diff tool, which allows their approach (which we call DJ-SZZ) to exclude non-executable changes (*e.g.,* `import` statements).

Davies et al. (2014) propose two variations on the criterion used to select the BIC among the candidates: L-SZZ uses the largest candidate, while R-SZZ uses the latest one. These improvements were done on top of the AG-SZZ algorithm.

MA-SZZ, introduced by Da Costa et al. (2016), excludes from the candidate BICs all the *meta-changes*, *i.e.,* commits that do not change the source code. This includes (i) branch changes, which are copy operations from one branch to another, (ii) merge changes, which consist in applying the changes performed in a branch to another one, and (iii) property changes, which only modify file properties (*e.g.,* permissions).

To further reduce the false positives, two new variants were introduced by Neto *et al.*, RA-SZZ (Neto et al., 2018) and RA-SZZ* (Neto et al., 2019). Both exclude from the BIC candidates the refactoring operations, *i.e.,* changes that should not modify the behavior of the program. Both approaches use state-of-the-art tools: RA-SZZ uses RefDiff (Silva and Valente, 2017), while RA-SZZ* uses Refactoring Miner (Tsantalis et al., 2018), with the second one being more effective (Neto et al., 2019).

The presented variants of SZZ do not parse lines added in bug-fixing commits (*e.g.,* an added `if` statement checking for `null` values). This is because a line added does not have a change history when processed by SZZ using the Annotation Graph (Kim et al., 2006) or the Line-Number mapping (Śliwerski et al., 2005). As we discussed in our previous work (Rosa et al., 2021), there are however cases in which lines added while fixing a bug can point to the correct bug-inducing change. Sahal and Tosun (2018) proposed the first approach to include in SZZ support for added lines (from here on A-SZZ). Specifically, when the bug-fixing changes add new lines, A-SZZ identifies the code blocks encapsulating them. Then, A-SZZ considers the set of lines in the block and discards the cosmetic changes and comment lines. Finally, it runs the original SZZ algorithm as if the remaining lines of the block were modified in the commit.

Concerning the empirical evaluations performed in the literature, the original SZZ was not evaluated (Śliwerski et al., 2005). Instead, all its variants, except MA-SZZ, were manually evaluated by their authors. One of them, RA-SZZ* (Neto et al., 2019), used an external dataset, *i.e.,* Defect4J (Just et al., 2014). MA-SZZ was evaluated using automated metrics, namely *earliest bug appearance*, *future impact of a change*, and *realism of bug introduction* (Da Costa et al., 2016).

Tool name	**Approach**	**Public repository**
SZZ Unleashed (Borg et al., 2019)	∼DJ-SZZ (Williams and Spacco, 2008a)	`https://github.com/wogscpar/SZZUnleashed`
OpenSZZ (Lenarduzzi et al., 2020b)	∼B-SZZ (Śliwerski et al., 2005)	`https://github.com/clowee/OpenSZZ`
PyDriller (Spadini et al., 2018)	∼AG-SZZ (Śliwerski et al., 2005)	`https://github.com/ishepard/pydriller`

Table 2: Open-source tools implementing SZZ.

In Table 2 we list the open-source implementations of SZZ. SZZ Unleashed (Borg et al., 2019) partially implements DJ-SZZ: it uses line-number mapping (Williams and Spacco, 2008a) but it does not rely on DiffJ (Pace, 2007) for computing diffs, also working on non-Java files. It does not take into account meta-changes (Da Costa et al., 2016) and refactorings (Neto et al., 2019).

OpenSZZ (Lenarduzzi et al., 2020b) implements the basic version of the approach, B-SZZ. Since it is based on the git `blame` command, it implicitly uses the annotated graph (Kim et al., 2006).

PyDriller (Spadini et al., 2018), a general purpose tool for analyzing git repositories, also implements B-SZZ. It uses a simple heuristic for ignoring C- and Python-style comment lines, as proposed by Kim et al. (2006). We do not report in Table 2 a comprehensive list of all the SZZ implementations that can be found on GitHub, but only the ones presented in papers.

2.2. SZZ in Software Engineering Research

The original SZZ algorithm and its variations were used in a plethora of studies. We discuss some examples, while for a complete list we refer to the extensive literature review by Rodríguez-Pérez et al. (2018), featuring 187 papers.

SZZ has been used to run several empirical investigations having different goals (Çaglayan and Bener, 2016; Lenarduzzi et al., 2020a; Wehaibi et al., 2016; Tufano et al., 2017; Bernardi et al., 2018; Eyolfson et al., 2014; Misirli et al., 2016; Canfora et al., 2011; Prechelt and Pepper, 2014; Bird et al., 2009a; Rodríguez-Pérez et al., 2018; Aman et al., 2019; Chen and Jiang, 2019; Posnett et al., 2013; Karampatsis and Sutton, 2020; Bavota and Russo, 2015; Kononenko et al., 2015; Palomba et al., 2018). For example, Aman et al. (2019) studied the role of local variable names in fault-introducing commits and they used SZZ to retrieve such commits, while Palomba et al. (2018) focused on the impact of code smells, and used SZZ to determine whether an artifact was smelly when a fault was introduced. Many studies also leverage SZZ to evaluate defect prediction approaches (Kim et al., 2008; Tan et al., 2015; Hata et al., 2012; Rahman et al., 2011; Tóth et al., 2016; Tu et al., 2020; Wen et al., 2016; Yan et al., 2020; Fan et al., 2019; Pascarella et al., 2019).

Looking at Table 1 it is worth noting that, despite its clear limitations (Kim et al., 2006), many studies, even recent ones, still rely on B-SZZ (Palomba et al., 2018; Pascarella et al., 2019; Çaglayan and Bener, 2016; Wen et al., 2016; Posnett et al., 2013; Kim et al., 2008; Tan et al., 2015; Kononenko et al., 2015; Wehaibi et al., 2016; Lenarduzzi et al., 2020a) (the approaches that use git implicitly use the annotation graph defined by Kim et al. (2006)). Improvements are only slowly adopted in the literature, possibly due to the fact that some of them are not released as tools and that the two standalone tools providing a public SZZ implementation were released only recently (Lenarduzzi et al., 2020b; Borg et al., 2019).

The studies most similar to ours are the one by Da Costa et al. (2016), the one by Rodríguez-Pérez et al. (2020) and the one by Herbold et al. (2022). Both report a comparison of different SZZ variants. Da Costa et al. (2016) defined and used a set of metrics for evaluating SZZ implementations without relying on a manually defined oracle. However, they specify that, ideally, domain experts should be involved in the construction of the dataset (Da Costa et al., 2016), which motivated our study. Rodríguez-Pérez et al. (2018) introduced a model for distinguishing bugs caused by modifications to the source code (the ones that SZZ algorithms can detect) and the ones that are introduced due to problems with external dependencies. They also used the model to define a manually curated dataset on which they evaluated SZZ variants. Their dataset is created by researchers and not domain experts. In our study, instead, we rely on the explicit information provided by domain experts in their commit messages. Herbold et al. (2022) conducted an empirical analysis on the defect labels (*i.e.,* bugfix commits) identified by SZZ and the impact on commonly used features for defect prediction. Their results, evaluated on a dataset of 38 Apache projects, show that SZZ is able to cor-

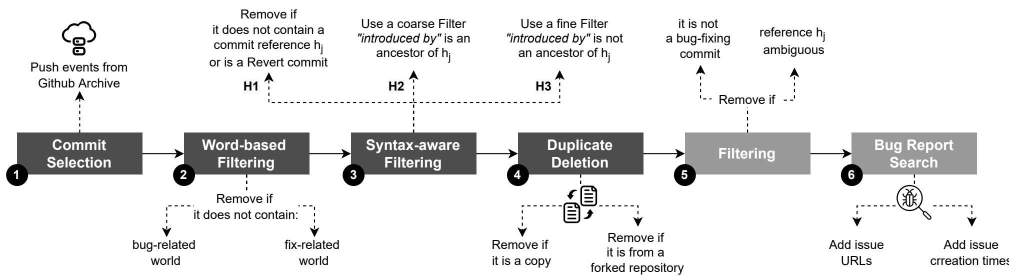


Figure 1: Process used for building the dataset. Steps 5 and 6 are the result of a manual evaluation.

rectly identify only half of the bug fixing commits, and using more features is not significant for defect prediction. In our study, we mainly focus on the construction of an evaluation dataset for SZZ, comparing the main variants proposed in literature.

3. Defining a Developer-informed Dataset for SZZ

In this section, we present a methodology to build a dataset of bug-inducing commits by exploiting information provided by developers when fixing bugs. Our methodology reduces the manual effort required for building such a dataset and more important, does not assume technical knowledge of the involved source code on the researchers' side.

The proposed methodology involves two main steps: (i) automatic mining from open-source repositories of bug-fixing commits in which developers explicitly indicate the commit(s) that introduced the fixed bug, and (ii) a manual filtering aimed at improving the dataset quality by removing ambiguous commit messages that do not give confidence in the information provided by the developer. In the following, we detail these two steps. The whole process is depicted in Fig. 1.

3.1. Mining Bug-fixing and Bug-inducing Commits

There are two main approaches proposed in the literature for selecting bug-fixing commits. The first one relies on the linking between commits and issues (Bissyande et al., 2013): issues labeled with "bug", "defect", etc. are mined from the issue tracking system, storing their issue ID (*e.g., systemd#17772*). Then, commits referencing the issue ID are mined from the versioning system and identified as bug-fixing commit. While such a heuristic is fairly precise, it has two important drawbacks that make it unsuitable for our work. First, the link to the issue tracking system must be known and a specific crawler for each different type of issue tracker (*e.g.,* Jira, Bugzilla, GitHub, etc.) must be built.

Second, projects can use a customized set of labels to indicate bug-related issues. Manually extracting this information for a large set of repositories is expensive. The basic idea behind this first phase is to use the commit messages to identify bug-fixing commits: we automatically analyze bug-fixing commit messages searching for those explicitly referencing bug-inducing commits.

As a preliminary step, we mined GH Archive (Grigorik, 2012) which provides, on a regular basis, a snapshot of public events generated on GitHub in the form of JSON files.

We mined the time period going from March 1^{st} 2011 to January 28^{th} 2021[1], extracting 24,042,335 commits performed in the context of *push* events: such events gather the commits done by a developer on a repository before performing the *push* action. Considering the goal of building an oracle for SZZ algorithms, we are not interested in any specific programming language. We performed three steps to select a candidate set of commits to manually analyze in the second phase: (i) we selected a first candidate set of bug-fixing commits, (ii) we used syntax-aware heuristics to refine such a set, and (iii) we removed duplicates.

3.1.1. Word-Based Selection of Bug-Fixing Commits

To identify bug-fixing commits, we first apply a lightweight regular expression on all the commits we gathered, as done in previous work (Fischer et al., 2003; Tufano et al., 2019). We mark as potential bug-fixes all commits accompanied by a message including at least a fix-related word[2] and a bug-related word[3]. We exclude the messages that include the word *merge* to ignore merge commits. Note that we do not need such a heuristic to be 100% precise, since two additional and more precise steps will be performed on the identified set of candidate fixing commits to exclude false positives (*i.e.,* a NLP-based step and a manual analysis).

[1] As compared to the ICSE'21 paper (Rosa et al., 2021) this manuscript extends, we analyze nine additional months of development, resulting in 4.4M additional commits.
[2] *fix* or *solve* [3] *bug*, *issue*, *problem*, *error*, or *misfeature*

3.1.2. Syntax-Aware Filtering of Referenced Bug-Fixing Commits

We needed to select from the set of candidate bug-fixing commits only the ones in which developers likely referenced the bug-inducing commit(s) (*i.e.,* referenced bug-fixing commits). We used the syntax-aware heuristics described below to do this. The first author defined such heuristics through a trial-and-error procedure, taking a 1-month time period of events on GH Archive to test and refine different versions of the heuristics, manually inspecting the achieved results after each run. The final version has been consolidated with the feedback of two additional authors.

As a preliminary step, we used the `doc.sents` function of the spaCy[4] Python module for NLP to extract the set S_c of sentences composing each commit message c.

For each sentence $s_i \in S_c$, we used spaCy to build its word dependency tree t_i, *i.e.,* a tree containing the syntactic relationships between the words composing the sentence. Fig. 2 provides an example of t_i generated for the sentence "*fixes a search bug introduced by 2508e12*".

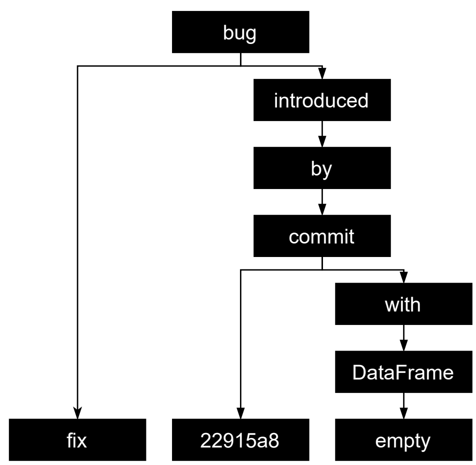


Figure 2: Example of word dependency tree built by spaCy.

By navigating the word dependency tree, we can infer that the verb "fix" refers to the noun "bug", and that the verb "introduced" is linked to commit id `2508e12` through the "by" apposition.

H1: Exclude Commits Without Reference and Reverts. We split each $s_i \in S_c$ into words and we select all its commit hashes $H(s_i)$ using

[4] https://spacy.io/

a regular expression[5]. We ignore all the s_i for which $H(s_i)$ is empty (*i.e.,* which do not mention any commit hash). Similarly, we filter out all the s_i that either (i) start with a commit hash, or (ii) include the verb "revert" referring to any $h_j \in H(s_i)$. We keep all the remaining s_i. We exclude the commits that do not contain any valid sentence as for this heuristic. We use the $H(s_i)$ extracted with this heuristic also for the following heuristics.

H2: Coarsely Filter Explicit Introducing References. If one of the ancestors of h_j is the verb "introduce" (in any declension), as it happens in Fig. 2, we consider this as a strong indication of the fact that the developer is indicating h_j as (one of) the bug-inducing commit(s). In this case, we check if h_j also includes at least one of the fix-related words[2] **and** one of the bug-related words[3] as one of its ancestors or children. At least one of the two words (*i.e.,* the one indicating the fixing activity or the one referring to a bug) must be an ancestor. We do this to avoid erroneously selecting sentences such as "*Improving feature introduced in 2508e12 and fixed a bug*", in which both the fix-related and the bug-related word are children of h_j.

For example, the h_j in Fig. 2 meets this constraint since it has among its ancestors both *fix* and *bug*. We also exclude the cases in which the words *attempt* or *test* (again, in different declensions) appear as ancestors of h_j. We do this to exclude false positives observed while experimenting with earlier versions of this heuristic.

For example, the sentence "*Remove attempt to fix error introduced in 2f780609*" belongs to a commit that aims at reverting previous changes. Similarly, the sentence "*Add tests for the fix of the bug introduced in 2f780609*" most likely belongs to the message of a test-introduction commit.

H3: Finely Filter Non-Explicit Introducing References. If h_j does not contain the verb "introduce" as one of its ancestors, we apply a finer filtering heuristic: both a word indicating a fixing activity **and** a word indicating a bug must appear as one of h_j's ancestors. Also, we define a list of stop-words that must not appear either in the h_j's ancestor as well as in the dependencies (*i.e.,* ancestors and children) of the "fixing activity" word. Such a stop-word list, derived through a trial-and-error procedure, includes eight additional words (*was*, *been*, *seem*, *solved*, *fixed*, *try*, *trie* (to capture *tries* and *tried*), and *by*), besides *attempt* and *test* also used in H2. This allows, for example, to exclude sentences such as "*This definitely fixes the bug I tried to fix in commit 26f3fe2*", meets all selection criteria for H3, but

[5] [0-9a-f]{6,40}

it is a false positive.

3.1.3. Deletion of Duplicate Commits

We saved the list of commits including at least one sentence s_i meeting H1 and either H2 or H3 in a MySQL database. Since we analyzed a large set of projects, it was frequent that some commits were duplicated due to the fact that different forks of a given project are available. As a final step, we removed such duplicates, keeping only the commit of the main project repository.

Out of the 24,042,335 parsed commits, the automated filtering selected 4,585 commits. Our goal with the above described process is not to be exhaustive, *i.e.,* we do not want to identify all bug-fixing commits in which developers indicated the bug-inducing commit(s), but rather to obtain a high-quality dataset of commits that were certainly of the bug-inducing kind. The quality of the dataset is then further increased during the subsequent step of manual analysis.

3.2. Manual Filtering

Four of the authors (from now on, evaluators) manually inspected the 4,585 commits produced by the previous step. The evaluators have different backgrounds (graduate student, faculty member, junior and a senior researcher with two years of industrial experience). The goal of the manual validation was to verify (i) whether the commit was an actual bug-fix, and (ii) if it included in the commit message a non-ambiguous sentence clearly indicating the commit(s) in which the fixed bug was introduced. For both steps the evaluators mostly relied on the commit message and, if available, on possible references to the issue tracker. Those references could be issue IDs or links that the evaluators inspected to (i) ensure that the fixed issue was a bug, and (ii) store for each commit the links to the mentioned issues and, for each issue, its opening date.

The latter is an information that may be required by an SZZ implementation (*e.g.,* SZZ Unleashed (Borg et al., 2019) and OpenSZZ (Lenarduzzi et al., 2020b) require the link to the issue) to exclude from the candidate list of bug-inducing commits those performed after the opening of the fixed issue.

Indeed, if the fixed bug has been already reported at date d_i, a commit performed on date $d_j > d_i$ cannot be responsible for its introduction. Since the commits to inspect come from a variety of software systems, they rely on different issue trackers. When an explicit link was not available, but an issue was mentioned in the commit message (*e.g.,* see the commit message shown in the introduction), the evaluators searched for the project's issue tracker, looking on the GitHub repository for documentation pointing to it (in case the project did not use the GitHub issue tracker itself). If no information was found, an additional Google search was performed, looking for the project website or directly searching for the issue ID mentioned in the commit message.

The manual validation was supported by a web-based application we developed that assigns to each evaluator the candidate commits to review, showing for each of them its commit message and a clickable link to the commit GitHub page. Using a form, the evaluator indicated whether the commit was relevant for the oracle (*i.e.,* an actual bug-fix documenting the bug-inducing commit) or not, and listing mentioned issues together with their opening date. Each commit was assigned by the web application to two different evaluators, for a total of 8,231 evaluations. To be more conservative and to have higher confidence in our oracle, we decided to not resolve conflicts (*i.e.,* cases in which one evaluator marked the commit as relevant and the other as irrelevant): we excluded from our oracle all commits with at least one "irrelevant" flag.

3.3. The Resulting SZZ Oracle

Out of the 4,585 manually validated commits, 2,304 (50%) passed our manual filtering, of which 212 include references to a valid issue (*i.e.,* an issue labeled as a bug that can be found online). For these, we also automatically checked if the issue date is valid considering the extracted bug commit (*i.e.,* the bug commit date must be before the issue date). This indicates that SZZ implementations that rely on information from issue trackers can only be run on a minority of bug-fixing commits. Indeed, the 2,304 instances we report have been manually checked as true positive bug-fixes, and only 212 of these (13%) mention the fixed issue. The dataset is available in our replication package (Rosa et al., TBD).

These 2,304 commits and their related bug-inducing commits impact files written in many different languages. All the implementations of the SZZ algorithm (except for B-SZZ) perform some language-specific parsing to ignore changes performed to code comments.

In our study (Section 4.1) we experimented several versions of the SZZ including those requiring the parsing of comments. We implemented support for the top-8 programming languages present in our oracle (*i.e.,* the ones responsible for more code commits): C, C++, C#, Java, JavaScript, Ruby, PHP, and Python. This led to the creation of the dataset we use in our experimentation, only including bug-fixing/inducing commits impacting files written in one of the eight programming languages we support. This dataset is also available in our replication package (Rosa et al., TBD). Table 3 summarizes the main characteristics of the *overall* dataset and of the *language-filtered* one. Note that the *language-filtered* dataset contains a lower number of instances also for repositories having as a main language one of the eight supported ones because some of their commits were related to unsupported languages (*e.g.,* fixing a bug in a Maven `pom` file).

	Overall			*Language-filtered*		
Language	**#Repos**	**#Commits**	**#Issues**	**#Repos**	**#Commits**	**#Issues**
C	406	520	62	343	430	43
Python	311	348	43	276	307	29
C++	187	223	25	159	189	19
JS	186	207	29	138	155	16
Java	92	106	14	74	83	8
PHP	65	73	6	57	64	3
Ruby	47	52	6	40	42	5
C#	31	38	3	25	32	1
Others	833	1077	99	0	0	0
Total	1,854	2,364	246	1,059	1,258	119

Table 3: Features of the *language-filtered*/*overall* datasets.

It is worth noting that a repository or even a commit can involve several programming languages: for this reason, the *total* may be lower than the sum of the per-language values (*i.e.,* a repository can be counted in two or more languages).

Besides sharing the datasets as JSON files, we also share the cloned repositories from which the bug-fixing commits have been extracted. This enables the replication of our study and the use of the datasets for the assessment of future SZZ improvements.

4. Study 1: Evaluating SZZ Variants

In this section we report the updated results of our first study, in which we use the oracle we built to evaluate state-of-the-art SZZ variants and tools.

4.1. Study Design

The *goal* of this study is to experiment different variants of the SZZ algorithm. The *perspective* is that of researchers interested in assessing the effectiveness of the state-of-the-art implementations and identify possible improvements that can be implemented to further improve the accuracy of the SZZ algorithm. To achieve such a goal, we aim to answer the following research question:

Table 4: Characteristics of the SZZ implementations we compare in the context of RQ$_{\mathbf{1}}$. We mark with a "◇" our re-implementations.

Acronym	**Fix Line Filtering**	**BIC Identification Method**	**BIC Filtering**	**BIC Selection**	**Differences w.r.t. the original paper**
B-SZZ	//	Annotation Graph(Kim et al., 2006)	//	//	We use git `blame` instead of the CVS `annotate`, *i.e.*, we implicitly use an annotation graph (Kim et al., 2006). We do not filter BICs based on the issue creation date.◇
AG-SZZ	Cosmetic changes(Kim et al., 2006)	Annotation Graph(Kim et al., 2006)	//	//	No differences.◇
MA-SZZ	Cosmetic changes(Kim et al., 2006)	Annotation Graph(Kim et al., 2006)	Meta-Changes(Da Costa et al., 2016)	//	No differences.◇
L-SZZ	Cosmetic Changes(Kim et al., 2006)	Annotation Graph(Kim et al., 2006)	Meta-Changes(Da Costa et al., 2016)	Largest (Davies et al., 2014)	We filter meta-changes (Da Costa et al., 2016).◇
R-SZZ	Cosmetic Changes(Kim et al., 2006)	Annotation Graph(Kim et al., 2006)	Meta-Changes(Da Costa et al., 2016)	Latest (Davies et al., 2014)	We filter meta-changes (Da Costa et al., 2016).◇
RA-SZZ*	Cosmetic Changes(Kim et al., 2006) Refactorings(Neto et al., 2019)	Annotation Graph(Kim et al., 2006)	Meta-Changes(Da Costa et al., 2016)	//	We use Refactoring Miner 2.0 (Tsantalis et al., 2020).◇
SZZ@PYD	Cosmetic Changes(Kim et al., 2006)	Annotation Graph(Kim et al., 2006)	//	//	We implement a wrapper for PyDriller (Spadini et al., 2018).
SZZ@UNL	Cosmetic Changes(Kim et al., 2006)	Line-number Mapping(Williams and Spacco, 2008a)	Issue-date(Śliwerski et al., 2005)	//	We implement a wrapper for SZZ Unleashed (Borg et al., 2019).
SZZ@OPN	//	Annotation Graph(Kim et al., 2006)	//	//	We implement a wrapper for OpenSZZ (Lenarduzzi et al., 2020b).

> ***RQ$_1$**: How do different variants of SZZ perform in identifying bug-inducing changes?* With this research question we want to compare the various state-of-the-art SZZ implementations using our dataset.

4.1.1. SZZ Implementations Compared

We used for our experiment different variants of the SZZ algorithm. Specifically, re-implemented all the main approaches available in the literature (presented in Section 2) in a publicly available tool named `pyszz`[6]

[6] `https://github.com/grosa1/pyszz`

which also includes an adapted version of the PyDriller SZZ implementation (Spadini et al., 2018). Moreover, we adapted existing Open Source tools (*i.e.,* SZZ Unleashed (Borg et al., 2019), and OpenSZZ (Lenarduzzi et al., 2020b)) to work with our dataset. We provide a replication package (Rosa et al., TBD) containing all the tools involved in the experiment with instructions on how to run them.

We report the details about all the implementations we compare in Table 4 and, for each of them, we explicitly mention (i) how it filters the lines changed in the fix (*e.g.,* it removes cosmetic changes), (ii) which methodology it uses for identifying the preliminary set of bug-inducing commits (*e.g.,* annotation graph), (iii) how it filters such a preliminary set (*e.g.,* it removes meta-changes), and (iv) if it uses a heuristic for selecting a single bug-inducing commit and, if so, which one (*e.g.,* most recent commit). We also explicitly mention any difference between our implementations and the approaches as described in the original papers presenting them.

As most of the bug-fix pairs in our dataset do not contain the reference to the bug-report (∼91%), all our re-implementations are independent from the issue-tracker systems. This is the reason why we did not set the "Issue-date" as a default BIC filtering technique, despite it is reported in the respective papers (*e.g.,* for B-SZZ). However, since we have extracted this information where present, we experiment all techniques with and without such a filtering applied. Note that git tracks both the *author's date* (*i.e.,* when the commit was performed in the first place) and the *commit's date*, which the latter changing every time the commit is being modified (*e.g.,* due to a rebasing of the branch). For the issue date filter we use the author's date since the commit's date might make SZZ erroneously filter out some legit bug-inducing commits. For example, let us consider an issue I reported at a date d_I, and its bug-inducing commit C having an author's date $da_C < d_I$ and a commit's date $dc_C > d_I$. This indicates a situation in which the issue was reported after the change was performed in the first place, but before C has been modified due, for example, to a rebase. If we considered the commit's date, we would have discarded C as a bug-inducing commit as performed after the issue was reported.

For the Open Source tools, instead, we did not modify their implementation of the BIC-finding procedures: *e.g.,* we did not remove the filtering by issue date from SZZ Unleashed. However, our wrappers for such tools allow to run them with our dataset. For example, SZZ Unleashed depends on a specific issue-tracker system (*i.e.,* Jira) for filtering commits done after the bug-report was opened. We made it independent from it by adapting our datasets to the input it expects (*i.e.,* Jira issues in JSON format). It is worth noting that, despite the complexity of such files, SZZ Unleashed only uses the issue opening date in its implementation. For this reason, we only provide such field and we set the others to `null`.

Note that some of the original implementations listed in Table 4 can identify bug-fixing commits. In our study, we did not want to test such a feature: we test a scenario in which the implementations already have the bug-fixing commits for which they should detect the bug-inducing commit(s).

4.1.2. Study Context

To evaluate the described implementations, we defined two version of the datasets extracted from the *language-filtered* dataset: (i) the $oracle_{all}$ dataset, featuring 1,258 bug-fixes, which includes both the ones with and without issue information, and (ii) the $oracle_{issues}$ dataset, featuring 119 instances, which includes only instances with issue information. Moreover, we defined two additional datasets, $oracle^{J}_{all}$ (81 instances) and $oracle^{J}_{issues}$ (8 instances), obtained by considering only Java-related commits from the $oracle_{all}$ and $oracle_{issues}$, respectively. We did this because two implementations, *i.e.,* RA-SZZ*[7] and OpenSZZ, only work on Java files.

4.1.3. Experimental Procedure

To answer $\mathrm{RQ}_{\mathbf{1}}$, we ran all the implementations on all the datasets on which they can be executed. This means that we run all the state-of-the-art SZZ implementations and tools (Table 4) on $oracle_{all}$ and $oracle_{issues}$, except for RA-SZZ* and OpenSZZ that are executed on the datasets including Java files only.

Another exception is for SZZ Unleashed, that requires the issue date in order to work. Since it would not be possible to run it on the $oracle_{all}$ dataset, we simulated the best-case-scenario for such commits: we pretended that an issue about the bug was created few seconds after the last bug-inducing commit was done. Consider the bug-fixing commit BF without issue information and its set of bug-inducing commits BIC; we assumed that the issue mentioned in BF had $max_{b \in BIC}(date(b)) + \delta$ as opening date, where δ is a small time interval (we used 60 seconds).

[7] It relies on Refactoring Miner (Tsantalis et al., 2020) which only works on Java files.

Such an experimental design allows us to compare all the implementations in two scenarios: (i) the *realistic* scenario ($oracle_{issues}$), in which the issue date is real, *i.e.,* it may be quite far from the BIC dates; (ii) the *best-case* scenario (*i.e.,* $oracle_{all}$) in which real issue information would be available only for a very small percentage of the bug-fixes instances, while the others are simulated. Thus, when experimenting the variants of the techniques not using the issue opening date, the results we achieve are those one would achieve in reality. Instead, when testing the approaches exploiting the issue opening date information, we are showing what would be the hypothetical effectiveness of such techniques in the best case scenario in which all commits refer to an issue having an identifiable opening date and, for most of the commits, the opening of the related issue immediately follows the bug introduction.

In the end, we obtained a set of bug-inducing commits detected by the experimented implementations. Based on the oracle from our datasets, we evaluated their accuracy by using three widely-adopted metrics: recall, precision, and F-measure (Baeza-Yates and Ribeiro-Neto, 1999).

In detail, we computed the such metrics using the following formulas:

$$recall = \frac{|correct \cap identified|}{|correct|}\% \qquad precision = \frac{|correct \cap identified|}{|identified|}\%$$

where *correct* and *identified* represent the set of true positive bug-inducing commits (those indicated by the developers in the commit message) and the set of bug-inducing commits detected by the experimented algorithm, respectively. As an aggregate indicator of precision and recall, we report the F-measure (Baeza-Yates and Ribeiro-Neto, 1999), defined as the harmonic mean of precision and recall. Such metrics were also used in previous works for evaluating SZZ variants (*e.g.,* Neto et al. (2019)).

Given the set of experimented SZZ variants/tools $SZZ_{exp} = \{v_1, v_2, \dots v_n\}$, we also analyze their complementarity, by computing the following metrics for each v_i (Oliveto et al., 2010):

$$correct_{v_i \cap v_j} = \frac{|correct_{v_i} \cap correct_{v_j}|}{|correct_{v_i} \cup correct_{v_j}|}$$

$$correct_{v_i \setminus (SZZ_{exp} \setminus v_i)} = \frac{|correct_{v_i} \setminus correct_{(SZZ_{exp} \setminus v_i)}|}{|correct_{v_i} \cup correct_{(SZZ_{exp} \setminus v_i}|}$$

where $correct_{v_i}$ represents the set of correct bug-inducing commits detected by v_i and $correct_{(SZZ_{exp} \setminus v_i)}$ the correct bug-inducing commits detected by all other techniques but v_i. $correct_{v_i \cap v_j}$ measures the overlap between the set of correct bug-inducing commits identified by two given implementations: we computed it between the pairs of experimented SZZ variants and present the results using a heatmap to better visualize the overlap metrics. $correct_{v_i \setminus (SZZ_{exp} \setminus v_i)}$, instead, measures the correct bug-inducing commits identified only by technique v_i and missed by all others experimented in $RQ_{\mathbf{1}}$. It is worth clarifying that, when we compute the overlap metrics, we compare all the implementations among them on the same dataset. This means, for example, that we do not compute the overlap between a variant tested on $oracle_{all}$ and another variant tested on $oracle_{issues}$.

As a last step, we compute the set of bug-fixing commits for which none of the experimented techniques was able to correctly identify the bug-inducing commit(s). Then, we qualitatively discuss these cases to understand (i) the weak points of the applied heuristics and (ii) if it is possible to refine these heuristics to cover particular cases.

4.2. Study Results

Table 5: Precision, recall, and F-measure calculated for all SZZ algorithms in the context of $RQ_{\mathbf{1}}$. † means Java only files.

	Algorithm	$oracle_{all}$			$oracle_{issue}$		
		Recall	**Precision**	**F1**	**Recall**	**Precision**	**F1**
No issue date filter	B-SZZ	0.68	0.39	0.49	0.69	0.37	0.48
	AG-SZZ	0.60	0.45	0.52	0.62	0.45	0.52
	L-SZZ	0.45	0.52	0.49	0.43	0.50	0.46
	R-SZZ	0.57	0.66	0.61	0.55	0.63	0.59
	MA-SZZ	0.63	0.36	0.46	0.66	0.35	0.46
	†RA-SZZ*	0.49	0.22	0.31	0.50	0.22	0.31
	SZZ@PYD	0.67	0.39	0.49	0.69	0.39	0.50
	SZZ@UNL	0.67	0.09	0.15	0.71	0.06	0.11
	†SZZ@OPN	0.20	0.33	0.25	0.12	0.50	0.20
With date filter	B-SZZ	0.68	0.42	0.52	0.69	0.38	0.49
	AG-SZZ	0.60	0.49	0.54	0.62	0.46	0.53
	L-SZZ	0.47	0.55	0.51	0.45	0.51	0.48
	R-SZZ	0.62	0.73	0.67	0.57	0.66	0.61
	MA-SZZ	0.63	0.39	0.49	0.66	0.36	0.47
	†RA-SZZ*	0.49	0.26	0.34	0.50	0.22	0.31
	SZZ@PYD	0.67	0.42	0.52	0.69	0.41	0.51
	SZZ@UNL	0.67	0.09	0.15	0.71	0.06	0.11
	†SZZ@OPN	0.20	0.34	0.25	0.12	0.50	0.20

Table 5 reports the results achieved by the experimented SZZ variants and tools. The top part of the table shows the results when the issue date filter has not been applied, while the bottom part relates to the application of the date filter. With "issue date filter" we refer to the process through which we remove from the list of candidate bug-inducing commits returned by a given technique all those performed after the issue documenting the bug has been opened. Those are known to be false positives. For this reason, such a filter is expected to never decrease recall (since the discarded bug-inducing commits should all be false positives) while increasing precision. The left part of Table 5 shows the results achieved on $oracle_{all}$, while the right part focuses on $oracle_{issue}$.

R-SZZ achieves the highest F-Measure (61%) when not using the issue date filtering (top part). Our implementation of R-SZZ uses the annotation graph, ignores cosmetic changes and meta-changes (as MA-SZZ), and only considers as bug-inducing commits the latest change that impacted a line changed to fix the bug. Thanks to that combination of heuristics, R-SZZ also achieves the highest precision on both oracles, achieving a precision score of 66% on $oracle_{all}$ and 63% on $oracle_{issue}$.

B-SZZ, the simplest SZZ version, exhibits the highest recall score of 68% on $oracle_{all}$ and 69% on $oracle_{issue}$, followed by PyDriller and SZZ@UNL. Nonetheless, B-SZZ pays in precision, making it the fourth algorithm together with the PyDriller implementation for $oracle_{all}$ and the sixth for $oracle_{issue}$. Due to the similarity between B-SZZ and the PyDriller implementation, also their performances are quite similar.

Despite the recall/precision tradeoff, R-SZZ and B-SZZ are not heavily affected in terms of recall score compared to SZZ@UNL (SZZ Unleashed). It achieves 66% of recall on $oracle_{all}$ and 67% on $oracle_{issue}$ datasets, with a very low precision of 9% and 6%, respectively. We investigated the reasons behind such a low precision, finding that it is mainly due to a set of outlier bug-fixing commits for which SZZ@UNL identifies a high number of (false positive) bug-inducing commits. For example, three bug-fixing commits are responsible for 72 identified bug-inducing commits, out of which only three are correct. We analyzed the distribution of bug-inducing commits reported by SZZ@UNL for the different bug-fixing commits. Cases for which more than 20 bug-inducing commits are identified for a single bug-fix can be considered outliers. By ignoring those cases, the recall and precision of SZZ@UNL are 66% and 17%, respectively on $oracle_{all}$, and 71% and 16% on $oracle_{issue}$. By lowering the outlier threshold to 10 bug-inducing, the precision grows in both datasets to 22%. We believe that the low precision of SZZ@UNL may be due to misbehavior of the tool in few isolated cases.

Two implementations (*i.e.,* RA-SZZ* and SZZ@OPN) only work with Java files. In this case, we compute their recall and precision by only considering the bug-fixing commits impacting Java files. Both of them exhibit limited recall and precision. While this is due in part to limitations of the implementations, it is also worth noting that the number of Java-related commits in our datasets is quite limited (*i.e.,* 81 in $oracle_{all}$ and only 8 in $oracle_{issue}$). Thus, failing on a few of those cases penalizes in terms of performance metrics.

AG-SZZ, L-SZZ, and MA-SZZ exhibit, as compared to others, good performance for both recall and precision. These algorithms provide a good balance between recall and precision, as also shown by their F-Measure (∼50%).

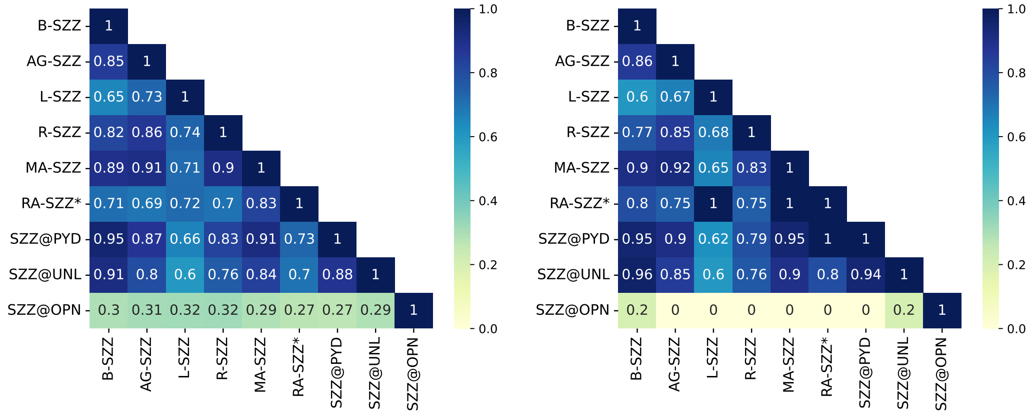
The bottom of Table 5 shows the results achieved by the same algorithms when using the issue data filter.

As expected, the recall remains, for the most of the cases, equal to the previous scenario with marginal improvements in precision (thanks to the removal of some false positives). While most of the algorithms improve their precision by 1%-4%, R-SZZ obtain substantial improvements in the $oracle_{all}$ dataset R-SZZ (+6%). This boosts the latter to a very good 73% precision on $oracle_{all}$, and 66% on $oracle_{issue}$ (+3%).

To summarize the achieved results: We found that R-SZZ is the most precise variant on our datasets, with a precision ∼70% when the issue date filter is applied. Thus, we recommend it when precision is more important than recall (*e.g.,* when a set of bug-inducing commits must be mined for qualitative analysis). If the focus is on recall, the current recommendation is to rely on B-SZZ, using, for example, the implementation provided by `PyDriller`. Finally, looking at Table 5, it is clear that there are still margins of improvement for the accuracy of the SZZ algorithm.

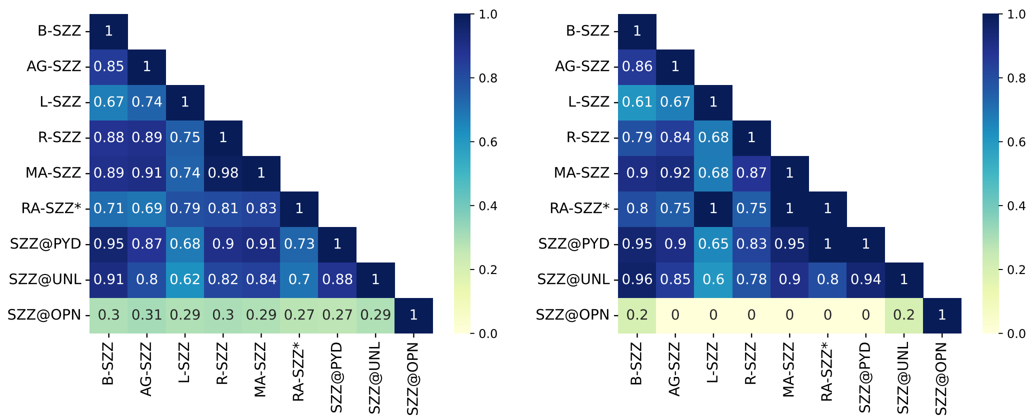
Table 6 shows the $correct_{v_i \setminus (SZZ_{exp} \setminus v_i)}$ metric we computed for each SZZ variant v_i. This metric measures the correct bug-inducing commits identified only by technique v_i and missed by all the others.

Fig. 3a and Fig. 3b depict the $correct_{v_i \cap v_j}$ metric computed between each pair of SZZ variants when not filtering based on the issue date, while Fig. 4a and Fig. 4b show the same metric when the issue filter has been applied. Given the metric definition, the depicted heatmaps will be symmetric. To improve the readability, we keep only the lower triangular matrix



(a) $oracle_{all}$ (b) $oracle_{issue}$

Figure 3: Overlap between SZZ variants, evaluated in $\text{RQ}_\mathbf{1}$, when no issue date filter is applied.



(a) $oracle_{all}$ (b) $oracle_{issue}$

Figure 4: Overlap between SZZ variants, evaluated in $\text{RQ}_\mathbf{1}$, when the issue date filter is applied.

(*i.e.*, $correct_{v_i \cap v_j} = correct_{v_j \cap v_i}$). The only technique able to identify bug-inducing commits missed by all others SZZ implementations is SZZ@UNL (19 on $oracle_{all}$ and 2 on $oracle_{issue}$) – Table 6. This is not surprising considering the high SZZ@UNL recall and the high number of bug-inducing commits it returns for certain bug-fixes. The main difference with the other evaluated SZZ variants is the BIC identification method used (*i.e.*, Line-number Map-

Table 6: Bug inducing commits correctly identified exclusively by the v_i algorithm. † Java only files.

Algorithm	No date filter		With date filter	
	$oracle_{all}$	$oracle_{issue}$	$oracle_{all}$	$oracle_{issue}$
B-SZZ	1/898	0/86	1/898	0/86
AG-SZZ	0/898	0/86	0/898	0/86
L-SZZ	0/898	0/86	0/898	0/86
R-SZZ	0/898	0/86	0/898	0/86
MA-SZZ	0/898	0/86	0/898	0/86
†RA-SZZ*	0/56	0/5	0/56	0/5
SZZ@PYD	0/898	0/86	0/898	0/86
SZZ@UNL	19/898 (2%)	2/86 (2%)	19/898 (2%)	2/86 (2%)
†SZZ@OPN	0/56	0/5	0/56	0/5

ping(Williams and Spacco, 2008a)). This can be the reason why none of the other implementations identifies such bug-inducing commits: Given 898 as cardinality of the intersection of the true positives identified by all SZZ techniques, SZZ@UNL correctly retrieves 842 of them.

Looking at the overlap metrics in Fig. 3 and Fig. 4, two observations can be made. First, the overlap in the identified true positives is substantial. Excluding SZZ@OPN, 24 of the 28 comparisons have an overlap in the identified true positives ≥70% and the lower values are still in the range 60-70%. The low overlap values observed for SZZ@OPN are instead due to the its low recall. Second, the complementarity between the different SZZ variants is quite low, which indicates that there is a set of bug-fixing commits for which all of the variants fail in identifying the correct bug-inducing commit(s). We manually analyzed those cases to derive possible improvements to the SZZ that we distill in the following.

The buggy line is not always impacted in the bug-fix. In some cases, fixing a bug introduced in line l may not result in changes performed to l. An example in Java is the addition of an `if` guard statement checking for `null` values before accessing a variable. In this case, while the bug has been introduced with the code accessing the variable without checking whether it is `null`, the bug-fixing commit does not impact such a line, it just adds the needed `if` statement. An example from our dataset is the bug-fixing commit from the *thcrap* repository[8] in which line 289 is modified to fix a bug

[8] `https://github.com/thpatch/thcrap/commit/29f1663`

introduced in commit `b67116d`, as pointed by the developer in the commit message. However, the bug was introduced with changes performed on line 290. Thus, running git blame on line 289 of the fix commit will retrieve a wrong bug-inducing commit. Defining approaches to identify the correct bug-inducing commit in these cases is far from trivial. Also, in several bug-fixing commits we inspected, the implemented changes included both added and modified/deleted lines. SZZ implementations focus on the latter, since there is no way to blame a newly added line. However, we found cases in which the added lines were responsible for the bug-fixing, while the modified/deleted ones were unrelated. An example is commit `ca11949` from the *snake* repository[9], in which two lines are added and two deleted to fix a bug. The deleted lines, while being the target of SZZ, are unrelated to the bug-fix, as clear from the commit message pointing to commit `315a64b`[10] as the one responsible for the bug introduction. In the bug-inducing commit, the developer refactored the code to simplify an `if` condition. While refactoring the code, she introduced a bug (*i.e.,* she missed an `else` branch). The fixing adds the `else` branch to the sequence of `if/else if` branches introduced in the bug-inducing commit. In this case, by relying on static analysis, it should be possible to link the added lines, representing the `else` branch, to the set of `if/else if` statements preceding it. While the added lines cannot be blamed, lines related to them (*e.g.,* acting on the same variable, being part of the same "high-level construct" like in this case) could be blamed to increase the chances of identifying the bug-inducing commit.

SZZ is sensible to history rewriting. Bird et al. (2009b) highlighted some of the perils of mining git repositories, among which the possibility for developers to rewrite the change history. This can be achieved through rebasing, for example: using such a strategy can have an impact on mining the change history (Kovalenko et al., 2018), and, therefore, on the performance of the SZZ algorithm. Besides rebasing, git allows to partially rewrite history by reverting changes introduced in one or more commits in the past. This action is often performed by developers when a task they are working on leads to a dead end. The revert command results in new commits in the change history that turn back the indicated changes. Consequently, SZZ can improperly show one of these commits as candidate bug-inducing. For

[9] `https://github.com/krmpotic/snake/commit/ca11949`
[10] `https://github.com/krmpotic/snake/commit/315a64b`

example, in the message of commit `5d8cee1` from the *xkb-switch* project[11], the developer indicates that the bug she is fixing has been introduced in commit `42abcc`. By performing a blame on the fix commit, git returns as a bug-inducing commit `8b9cf29`[12], which is a revert commit. By performing an additional blame step, the correct bug-inducing commit pointed by the developer can be retrieved[13].

5. New Heuristics for Improving SZZ

Based on the discussed limitations, we propose two new heuristics aimed at improving SZZ. In the first one, H_{DU}, we use data flow analysis to process added lines in bug-fixing commits in order to identify unchanged lines that might be the actual buggy lines on which the blame must be performed to correctly retrieve the bug-inducing commits. In the second one, H_R, we propose a heuristic that allows SZZ to be aware of reverted changes, *i.e.,* changes that result in new commits that undo previous changes. While both heuristics can be combined with any SZZ variant, we experiment them with MA-SZZ and R-SZZ, providing four new variants that we implement in our `pyszz` tool.

5.1. H_{DU}: Handling Added Lines

As outlined in Section 4.2, developers might add new lines to fix bugs, but such lines are ignored by all SZZ variants. To overcome such a limitation, it would be necessary to (i) identify the instructions functionally impacted by the added lines and (ii) run the SZZ on those lines, assuming that some of them induced the bug.

To achieve this goal, we define H_{DU}, a heuristic that relies on Definition-Use Chains (DUCs) to process added lines. We report below the steps for running H_{DU}:

Step 1: Building Definition-Use Chains. A Definition-Use Chain (DUC) is a data structure that links the definition of a variable to all its uses. DUCs can be statically extracted from source code. To extract the DUCs from a given file, we first identify all the declared functions or methods. Then, for each of them, we parse each line and we assign the label def_v if it assigns

[11] `https://github.com/grwlf/xkb-switch/commit/5d8cee1`
[12] `https://github.com/grwlf/xkb-switch/commit/8b9cf29`
[13] `https://github.com/grwlf/xkb-switch/commit/42abcc0`

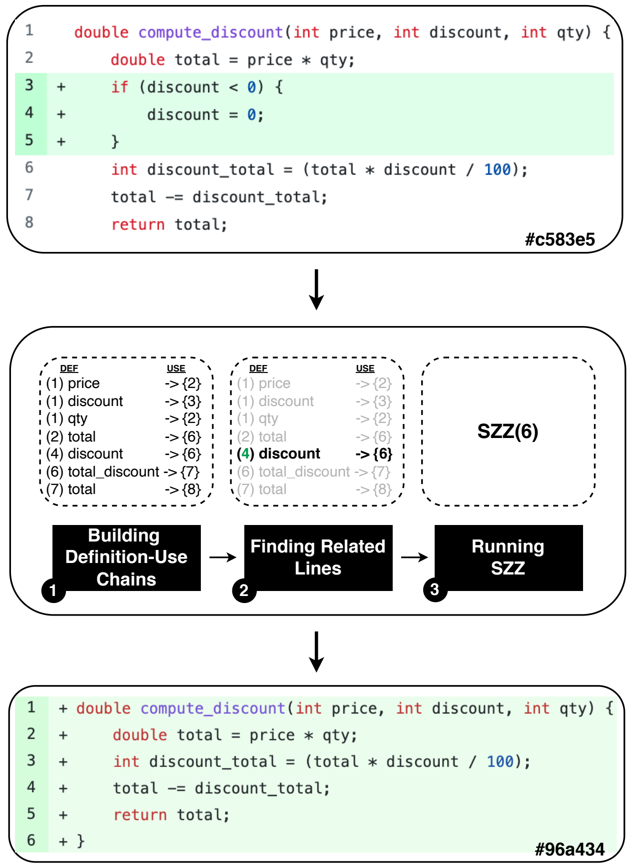


Figure 5: Example workflow of H_{DU} heuristic.

the variable v and the labels use_v if it uses variable v. For example, the line `int a = b + c` is marked with the labels def_a, use_b, and use_c. Finally, for each variable v, we link all the instruction that use v (marked with use_v) to the nearest instruction that precedes and defines it (*i.e.*, marked with def_v). It is worth noting that for each instruction we keep the line number in which it appears. Therefore, we transform the instructions into line numbers, and determine which lines are related by definition-use relationships. The output of this step is a map DUM that associates each def line with its respective use lines.

Step 2: Finding Related Lines. Given the list of added lines L_a in the bug-fixing commit, we aim at finding related lines in DUM. We find for all the line numbers L_a their reference in DUM, where we extract the

DUCs containing L_a. From the selected DUCs, for each def, we select the use line at distance $k = 1$. As a result, we obtain a set of $def - use$ pairs, from which we extract the referenced line numbers. Pairs involving the lines added in the bug-fixing commits are ignored, since it would not be possible to run SZZ on them due to the lack of a change history.

Step 3: Running SZZ. As a final step, we use SZZ on all the lines identified in the previous step, as if they were modified in the commit. The assumption is that the commit that introduced/modified such lines was probably responsible for the introduction of the bug.

Fig. 5 shows an example of our H_{DU} heuristic. We implemented a prototype implementation of H_{DU} for the C programming language, given the need to perform language-dependent static analysis. We choose C because it is the programming language with the largest number of instances in our dataset. It is worth noting, however, that our methodology can be adapted to other languages. We used `SrcML`[14] to parse the input files and convert them in XML-like format to support the static analysis.

5.2. H_R: Filtering Revert Commits

The second heuristic that we introduce is a filter for reverting changes. As we found in our first study, SZZ is sensible to history rewritings: Rebase operations and revert commits might be erroneously selected as bug-inducing commits.

When a rebase operation is performed, the change history is entirely wiped up to a specific commit. In such cases, it is impossible to go back to the previous version of the history. In other words, rebase operations can not be treated. Revert commits, instead, are additional commits that apply inverse changes up to a given point. Therefore, revert commit explicitly appear in the revision history. Similarly to what done in MA-SZZ, we filter the SZZ output to ignore revert commits and reduce the number of false positives. Therefore, we implemented H_R, a heuristic that leverages the commit message to identify reverted commits and ignore them. Such a filter consists in a simple string match using two patterns. With the first one, we skip commit that contain the sequence *“This reverts commit”* in the message. With the second pattern, we skip commits that start with the sequence *“Revert”*. We define these two pattern taking into account the

[14] `https://www.srcml.org/doc/c_srcML.html`

default reverting commit message provided by git. This means that H_R can not identify reverting commits having a customized commit message.

6. Study 2: Evaluating the Proposed SZZ Heuristics

In this section we report our second study, in which we evaluate the two novel heuristics we introduced.

6.1. Study Design

The *goal* of this study is to evaluate whether the two new heuristics we propose, H_{DU} and H_R, allow to improve the accuracy of the SZZ algorithm. In particular, we aim to answer the following research questions:

- ***RQ$_2$**: Does H_{DU} improve the accuracy of SZZ?* With this research question, we want to evaluate the effectiveness of the heuristic we defined for handling added lines.
- ***RQ$_3$**: Does H_R improve the accuracy of SZZ?* In this research question, we aim to experiment our heuristic that allows SZZ to be aware of reverting commits.

6.1.1. Study Context

We reply on the previously described $oracle_{all}$ and $oracle_{issues}$ dataset. Since the implementation of our H_{DU} heuristic performs data flow analysis for functions written in C, we defined two additional datasets: $oracle^C_{all}$ (397 instances) and $oracle^C_{issues}$ (40 instances), obtained by considering only C-related commits from the $oracle_{all}$ (1,258 instances) and $oracle_{issues}$ (119 instances), respectively. That means we selected all the bug-fix commits impacting only *.c* and *.h* source files.

6.1.2. Experimental Procedure

To answer $\mathrm{RQ}_{\mathbf{2}}$, we compare H_{DU} with the approach defined by Sahal and Tosun (2018). As reported in Section 2, such a heuristic runs SZZ on all the lines belonging to the same blocks of the added lines. Since no tool implementing such a heuristic is available, we re-implemented the approach by Sahal *et al.*. Similarly to H_{DU}, we implemented such a heuristic to work on C code. To understand if H_{DU} allows to improve the accuracy of SZZ, we combine it (and also the baseline heuristic) with two SZZ variants: MA-SZZ (*i.e.,* the implementation adopting the most complete set of filtering heuristics, excluding RA-SZZ that only works for Java code), and R-SZZ (*i.e.,* the one that achieved the best results in our first study).

In total, we define four new variants: MA-SZZ@DU, MA-SZZ@A, R-SZZ@DU, and R-SZZ@A. Note that the variants starting with "DU-" are those adopting our H_{DU} heuristic, while those starting with "A-" are those using the approach defined by Sahal and Tosun (2018). We run such variants on the $oracle^{C}_{all}$ and the $oracle^{C}_{issues}$ datasets. As a reference baseline, we also run the original SZZ implementation on these datasets. We use the same experimental design and performance metrics adopted in our first study.

To answer $RQ_{\mathbf{3}}$, similarly to $RQ_{\mathbf{2}}$, we combine H_R with MA-SZZ and R-SZZ. Thus, we define two new variants: MA-SZZ@REV and R-SZZ@REV. Since such an implementation supports any programming language, we run it on $oracle_{all}$ and $oracle_{issues}$. Again, as a reference, we compare the results with the ones obtained on MA-SZZ, R-SZZ, and B-SZZ.

As a last step, we compute the set of bug-fixing commits for which none of the experimented techniques was able to correctly identify the bug-inducing commit(s). Then, we qualitatively discuss these cases to understand (i) the weak points of the applied heuristics and (ii) if it is possible to further refine these heuristics to cover corner cases we did not consider.

6.2. Study Results

*6.2.1. **RQ₂**: Does H_{DU} improve the accuracy of SZZ?*

Table 7 reports the resulting metrics for the six variants we compare based on R-SZZ and MA-SZZ.

When no issue date filter is applied, R-SZZ@DU is the best performing on $oracle^{C}_{all}$, followed by R-SZZ. Considering $oracle^{C}_{issues}$, both R-SZZ@DU and R-SZZ achieve an F-measure score of 53%. The same is true for Precision. R-SZZ@A is the worst performing variant, with an F-measure of 53% on $oracle^{C}_{all}$, which goes down to 40% for $oracle^{C}_{issues}$. However, MA-SZZ remains the best compared to its two variants regarding Recall and F-measure score. MA-SZZ@A have the lowest F-measure and Precision, obtaining the highest Recall of 73% and 68% on the two datasets. This is a consequence of the selection heuristic used where the entire code block encapsulating the added lines is returned.

The observed differences are related to the underlying BIC selection heuristic behind R-SZZ. With R-SZZ@A, the resulting BICs are filtered, selecting, for each instance, only the most recent commit, thus effectively reducing the disadvantage it has with MA-SZZ in terms of Precision, which,

Table 7: Precision, recall, and F-measure calculated for the SZZ algorithms evaluated in the context of $RQ_{\mathbf{2}}$.

	Algorithm	$oracle^{C}_{all}$			$oracle^{C}_{issue}$		
		Recall	**Precision**	**F1**	**Recall**	**Precision**	**F1**
No filter	R-SZZ@A	0.51	0.54	0.53	0.40	0.40	0.40
	R-SZZ@DU	**0.55**	**0.64**	**0.59**	**0.50**	**0.57**	**0.53**
	R-SZZ	0.54	0.63	0.58	**0.50**	**0.57**	**0.53**
	MA-SZZ@A	**0.73**	0.06	0.12	**0.68**	0.03	0.06
	MA-SZZ@DU	0.62	0.28	0.38	0.57	0.20	0.29
	MA-SZZ	0.60	**0.35**	**0.44**	0.57	**0.25**	**0.35**
Issue date filter	R-SZZ@A	**0.68**	**0.73**	**0.70**	0.42	0.42	0.42
	R-SZZ@DU	0.60	0.72	0.66	**0.53**	**0.60**	**0.56**
	R-SZZ	0.59	0.72	0.65	**0.53**	**0.60**	**0.56**
	MA-SZZ@A	**0.73**	0.07	0.12	**0.68**	0.03	0.06
	MA-SZZ@DU	0.62	0.33	0.43	0.57	0.23	0.32
	MA-SZZ	0.60	**0.37**	**0.46**	0.57	**0.26**	**0.35**

instead, does not filter the BICs. The same is true for R-SZZ@DU and MA-SZZ@DU, where the BIC filtering procedure used in R-SZZ (most recent commit) gives the same advantage to R-SZZ@A. However, as H_{DU} is more conservative than the heuristic by Sahal and Tosun (2018), the impact on Precision is always acceptable. For example, considering $oracle^{C}_{all}$, MA-SZZ identifies a total of 688 bug-inducing changes against the 883 of MA-SZZ@DU and 4575 of MA-SZZ@A.

When the issue date filter is applied, similarly to $RQ_{\mathbf{1}}$, there is a general improvement in the Precision score due to the reduced number of false-positive BICs.

In general, combining SZZ with heuristics that can process added lines improves SZZ. Therefore, both the heuristics work well when combined with R-SZZ and less well when combined with MA-SZZ.

6.2.2. $\boldsymbol{RQ_3}$: Does H_R improve the accuracy of SZZ?

We report in Table 8 the resulting metrics of our experiment. Both MA-SZZ@REV and R-SZZ@REV perform similar to MA-SZZ and R-SZZ, achieving a small improvement (~ 1%) with and without the issue date filter. When the issue date filter is applied, there is a general improvement in terms of Precision, as seen for $RQ_{\mathbf{1}}$.

Table 8: Precision, recall, and F-measure calculated for the SZZ algorithms evaluated in the context of RQ_3.

	Algorithm	$oracle_{all}$			$oracle_{issue}$		
		Recall	**Precision**	**F1**	**Recall**	**Precision**	**F1**
No filter	MA-SZZ	0.63	0.36	0.46	0.66	0.35	0.46
	MA-SZZ@REV	0.64	0.36	0.46	0.66	0.36	0.47
	R-SZZ	0.57	0.66	0.61	0.55	0.63	0.59
	R-SZZ@REV	0.58	0.66	0.62	0.57	0.65	0.61
With filter	MA-SZZ	0.63	0.39	0.48	0.66	0.36	0.47
	MA-SZZ@REV	0.64	0.39	0.49	0.66	0.37	0.47
	R-SZZ	0.62	0.73	0.67	0.57	0.66	0.61
	R-SZZ@REV	0.63	0.74	0.68	0.59	0.67	0.63

We can conclude that H_R only has a positive effect when combined with R-SZZ, where the BIC selection heuristic picks only one commit as a BIC candidate. As a consequence, the effectiveness of the revert commit filter is concrete only for some SZZ variants. Another point to consider is that the effectiveness of the heuristic directly depends on the presence of cases where there are revert commits. However, our heuristic never reduced the efficacy of the baselines: This means that H_R can be safely used on top of any SZZ variant, and we found no drawbacks in including it.

7. Results Discussion

In summary, our first and second studies show that (i) R-SZZ generally achieves the best results, and (ii) by considering added lines and revert commits, the accuracy of SZZ improves. Interestingly, however, we found such an advantage (mostly, the ones related to added lines) dependent on the context. Some variants might work better in some cases, while some others in other cases. To check this intuition, we measure, for each commit, what is the best performing SZZ variant in terms of correctly identified BICs. To do this, for each variant v_j and commit C_i, we compute the precision score for each bugfix commit as follows:

$$F_{C_i}^{v_j} = \frac{|identified_{C_i}^{v_j} \cap correct_{C_i}|}{|identified_{C_i}^{v_j}|}$$

where $identified_{C_i}^{v_j}$ is the set of BICs returned by v_j for commit C_i, and

Table 9: Correctness ratio computed among all evaluated SZZ approaches.

Algorithm	No issue date filter		With issue date filter	
	$oracle^C_{all}$	$oracle^C_{issues}$	$oracle^C_{all}$	$oracle^C_{issues}$
B-SZZ	19/397 (0.05)	4/40 (0.10)	17/397 (0.04)	3/40 (0.08)
AG-SZZ	17/397 (0.04)	2/40 (0.05)	2/397 (0.01)	2/40 (0.05)
MA-SZZ	2/397 (0.01)	0/40	0/397	0/40
L-SZZ	4/397 (0.01)	0/40	0/397	0/40
R-SZZ	2/397 (0.01)	**20/40 (0.50)**	1/397 (0.00)	**21/40 (0.53)**
MA-SZZ@A	10/397 (0.03)	2/40 (0.05)	3/397 (0.01)	2/40 (0.05)
R-SZZ@A	32/397 (0.08)	1/40 (0.03)	**269/397 (0.68)**	1/40 (0.03)
MA-SZZ@DU	0/397	0/40	0/397	0/40
R-SZZ@DU	**218/397 (0.55)**	0/40	12/397 (0.03)	0/40
MA-SZZ@REV	0/397	0/40	0/397	0/40
R-SZZ@REV	0/397	0/40	0/397	0/40

$correct_{C_i}|$ is the set of BICs correctly identified by v_j for the commit C_i. The higher the score, the more the given variant is suitable for the commit. For each commit C_i, we award a point to the SZZ variant(s), achieving the highest score for C_i. Then, we sum such scores. In case there are more SZZ implementations with the same score, we assign the point to the one that also achieves the highest *F-measure* score on the entire dataset. We identify the final resulting score as *correctness ratio*. In Table 9 we report the correctness ratio score. When the issue date filter is not applied, R-SZZ@DU achieves the highest score for $oracle^C_{all}$, while for $oracle^C_{issues}$ the best performing is R-SZZ. The SZZ variants that are less effective, without earning any points on both datasets, are MA-SZZ@DU, R-SZZ@REV, and MA-SZZ@REV. When the issue date filer is applied, R-SZZ@A achieves the highest correctness ratio score (68%)on $oracle^C_{all}$, while looking at $oracle^C_{issues}$ the top performer is still R-SZZ (53%). This confirms what we stated in RQ$_\mathbf{2}$, that the best combination of line processing heuristic, BIC selection techniques and filters for SZZ depend on a specific bug-fixing context (*i.e.,* fix pattern). As the proposed heuristics give the best improvement to R-SZZ, we can also conclude that not all the SZZ heuristics are compatible, but some work better in combination with others. To verify this, for each commit, we pick only the best performing SZZ implementation to compare the resulting *F-measure* scores to the highest achieved in the context of RQ$_\mathbf{2}$.Thus, we obtain an overall score of 0.71 (+0.12) for the dataset $oracle^C_{all}$ and 0.63 (+0.10) for $oracle^C_{issues}$, without applying the issue date filter. When the issue date filter is applied, we achieve 0.75 (+0.05) and 0.65 (+0.09), respectively. Surprisingly, both R-SZZ@REV and MA-SZZ@REV does not gain any points with and without filtering by issue date. This because the uniquely identified commits, looking at the results from $RQ_{\mathbf{3}}$, do not impact C source files. Thus, the H_R does not give any advantage over the other SZZ implementations considering the C-only dataset.

There are still bug-inducing changes that the improved SZZ implementation can not identify. A first example is commit `b0f795` from the *libMesh/libmesh* project[15], where the C file extension is used for a C++ source file and only added lines are present as fixing change. Our SZZ implementations can not correctly process such files as they only work for C source code. Another example is commit `d6ef40` from the repository *gxt/QEMU*[16]. In that case, the bug and the fix impact different files (`cpu-all.h` and `main.c`, respectively). It is interesting to notice that, in such a case, the commit message of the bug-fixing commit contains a reference to the file involved in the bug-inducing commit: *"...but we need to at least define the **`reserved_va`** global so that **`cpu-all.h`**'s **`RESERVED_VA`** macro will work correctly."*. A similar observation can be done for commit `aebda6` from *OpenChannelSSD/linux*[17]: To identify the bug-inducing change, SZZ has to process lines that are not related to those impacted by the fix (*e.g.,* line 548). In this case, the commit message contains information about the method impacted by the fix: *"...to fix the issue, as we have to do is make sure that our **`start_config_issued`** flag gets reset whenever we receive a **`SetInterface`** request."* This shows that it can be possible to use NLP-based techniques to extract information about code artifacts indirectly affected by a commit, using such a piece of information to improve SZZ variants.

8. Threats to Validity

Construct validity. During the manual validation, the evaluators mainly relied on the commit message and the linked issue(s), when available, to confirm that a mined commit was a bug-fixing commit. Misleading information in the commit message could result in the introduction of false positive instances in our dataset. However, all commits have been checked by at least two evaluators and doubtful cases have been excluded, privileging a conservative approach. To build our dataset, we considered all the projects from

[15] `https://github.com/libMesh/libmesh/commit/b0f7953`
[16] `https://github.com/gxt/QEMU/commit/d6ef40b`
[17] `https://github.com/OpenChannelSSD/linux/commit/aebda61`

GitHub, without explicitly defining criteria to select only projects that are invested in software quality. Our assumption is that the fact that developers take care of documenting the bug-introducing commit(s) is an indication that they care about software quality. To ensure that the commits in our dataset are from projects that take quality into account, we manually analyzed 123 projects from our dataset, which allowed us to cover a significant sample of commits (286 out of 1,115, with 95%±5% confidence level). For each of them, we checked if they contained elements that indicate a certain degree of attention to software quality, *i.e.,* (i) unit test cases, (ii) code reviews (through pull requests), (iii) and continuous integration pipelines. We found that in 95% of the projects, developers (i) wrote unit test cases, and (ii) conducted code reviews through pull requests. Also, we found CI pipelines in 75% of the projects.

Internal validity. There is a possible subjectiveness introduced of the manual analysis, which has been mitigated with multiple evaluators per bug-fix. Also, we reimplemented most of the experimented SZZ approaches, thus possibly introducing variations as compared to what proposed by the original authors. We followed the description of the approaches in the original papers, documented in Table 4 any difference between our implementations and the original proposals, and share our implementations (Rosa et al., TBD). Also, note that the differences documented in Table 4 always aim at improving the performance of the SZZ variants and, thus, should not be detrimental for their performance. Another point is that our new implementations of H_{DU} and A-SZZ can have critical point or exceptional cases actually not handled. For example, when construct Definition-Use chains only at method level, thus as discussed in Section 7 there are some cases where our heuristic can not identify the correct BIC. Also, for MA-SZZ@A and R-SZZ@A, currently we do not apply the BICs filter described in the paper, where they select at most 4 commits as BIC. This because we replaced that filter with the filtering heuristic of R-SZZ.

External validity. While it is true that we mined millions of commits to build our dataset, we used very strict filtering criteria that resulted in 2,304 instances for our oracle. Also, the SZZ implementations have been experimented on a smaller dataset of 1,258 instances that is, however, still larger than those used in previous works. Finally, our dataset represents a subset of the bug-fixes performed by developers. This is due to our design choice, where we used strict selection criteria when building our oracle to prefer quality over quantity. It is possible that our dataset is biased towards a specific type of bug-fixing commits: there might be an inherent difference between the bug fixes for which developers document the bug-inducing commit(s) (*i.e.,* the only ones we considered) and other bug fixes.

While, to date, this is the largest dataset to evaluate SZZ implementations, additional mining and different filtering heuristics could help in improving the generalizability of our findings.

9. Conclusion and Future Works

SZZ is a widely studied and adopted algorithm in the context of software engineering research for defect analysis and prediction and also for tasks of Mining Software Repositories (MSR). Exploring new way to improve the effectiveness of SZZ can be always a precious contribution. Also, the creation of a platform to perform a sound and rightful comparison of the various state-of-the-art variant of SZZ is still an issue. The contributions of our work are for first an extensive dataset of developer informed bug-fix commit pairs to evaluate SZZ, where we performed a thorough comparison of the existing SZZ variants including two new heuristics, namely H_{DU} and H_{R}. As a result, the best performing SZZ variant is R-SZZ considering the classical definition of the algorithm. When we consider bug-fixing changes having added lines, one of our new implementation based on Definition-Use chains (R-SZZ@DU) achieves good results together with R-SZZ and R-SZZ@A. Moreover, the new heuristic H_{R}, applied to R-SZZ and MA-SZZ, also gives a slight improvement to SZZ.

The discussion of the results highlights additional points to explore. A first point to explore is to find the optimal combination of filters and heuristics for SZZ considering the bug-fixing pattern in the context of fixing. Moreover, the commit message can help to obtain the missing link between bug and fix, when they impact different locations of the source code. Also, exploring different combinations with static analysis techniques, such as our heuristic H_{DU}, can improve the effectiveness of SZZ.

10. Data Availability

The complete study material, data, and source code of our re-implementations are fully available in our replication package (Rosa et al., TBD).

Acknowledgment

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851720). We are grateful for the support by the Swiss National Science foundation (SNF) and JSPS (Project "SENSOR").

References

Aman, H., Amasaki, S., Yokogawa, T., Kawahara, M., 2019. Empirical study of fault introduction focusing on the similarity among local variable names, in: QuASoQ@ APSEC, pp. 3–11.

Baeza-Yates, R., Ribeiro-Neto, B., 1999. Modern Information Retrieval. Addison-Wesley.

Bavota, G., Russo, B., 2015. Four eyes are better than two: On the impact of code reviews on software quality, in: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE. pp. 81–90.

Bernardi, M.L., Canfora, G., Di Lucca, G.A., Di Penta, M., Distante, D., 2018. The relation between developers' communication and fix-inducing changes: An empirical study. Journal of Systems and Software 140, 111–125.

Bird, C., Bachmann, A., Aune, E., Duffy, J., Bernstein, A., Filkov, V., Devanbu, P., 2009a. Fair and balanced? bias in bug-fix datasets, in: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 121–130.

Bird, C., Rigby, P.C., Barr, E.T., Hamilton, D.J., German, D.M., Devanbu, P., 2009b. The promises and perils of mining git, in: 2009 6th IEEE International Working Conference on Mining Software Repositories, IEEE. pp. 1–10.

Bissyande, T.F., Thung, F., an?d D. Lo, S.W., Jiang, L., Reveillere, L., 2013. Empirical evaluation of bug linking, in: 2013 17th European Conference on Software Maintenance and Reengineering, pp. 89–98.

Borg, M., Svensson, O., Berg, K., Hansson, D., 2019. Szz unleashed: an open implementation of the szz algorithm-featuring example usage in a study of just-in-time bug prediction for the jenkins project, in: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, pp. 7–12.

Çaglayan, B., Bener, A.B., 2016. Effect of developer collaboration activity on software quality in two large scale projects. Journal of Systems and Software 118, 288–296.

Canfora, G., Ceccarelli, M., Cerulo, L., Di Penta, M., 2011. How long does a bug survive? an empirical study, in: 2011 18th Working Conference on Reverse Engineering, IEEE. pp. 191–200.

Chen, B., Jiang, Z.M.J., 2019. Extracting and studying the logging-code-issue-introducing changes in java-based large-scale open source software systems. Empirical Software Engineering 24, 2285–2322.

Da Costa, D.A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., Hassan, A.E., 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Transactions on Software Engineering 43, 641–657.

Davies, S., Roper, M., Wood, M., 2014. Comparing text-based and dependence-based approaches for determining the origins of bugs. Journal of Software: Evolution and Process 26, 107–139.

Eyolfson, J., Tan, L., Lam, P., 2014. Correlations between bugginess and time-based commit characteristics. Empirical Software Engineering 19, 1009–1039.

Fan, Y., Xia, X., da Costa, D.A., Lo, D., Hassan, A.E., Li, S., 2019. The impact of changes mislabeled by szz on just-in-time defect prediction. IEEE Transactions on Software Engineering .

Fischer, M., Pinzger, M., Gall, H.C., 2003. Populating a release history database from version control and bug tracking systems, in: 19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands, p. 23.

Grigorik, I., 2012. GitHub Archive. https://www.githubarchive.org.

Hata, H., Mizuno, O., Kikuno, T., 2012. Bug prediction based on fine-grained module histories, in: 2012 34th international conference on software engineering (ICSE), IEEE. pp. 200–210.

Herbold, S., Trautsch, A., Trautsch, F., Ledel, B., 2022. Problems with szz and features: An empirical study of the state of practice of defect prediction data collection. Empirical Software Engineering 27, 1–49.

Just, R., Jalali, D., Ernst, M.D., 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp. 437–440.

Karampatsis, R.M., Sutton, C., 2020. How often do single-statement bugs occur? the manysstubs4j dataset, in: Proceedings of the 17th International Conference on Mining Software Repositories, MSR, p. To appear.

Kim, S., Whitehead, E.J., Zhang, Y., 2008. Classifying software changes: Clean or buggy? IEEE Transactions on Software Engineering 34, 181–196.

Kim, S., Zimmermann, T., Pan, K., James Jr, E., et al., 2006. Automatic identification of bug-introducing changes, in: 21st IEEE/ACM international conference on automated software engineering (ASE'06), IEEE. pp. 81–90.

Kononenko, O., Baysal, O., Guerrouj, L., Cao, Y., Godfrey, M.W., 2015. Investigating code review quality: Do people and participation matter?, in: 2015 IEEE international conference on software maintenance and evolution (ICSME), IEEE. pp. 111–120.

Kovalenko, V., Palomba, F., Bacchelli, A., 2018. Mining file histories: Should we consider branches?, in: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 202–213.

Lenarduzzi, V., Lomio, F., Huttunen, H., Taibi, D., 2020a. Are sonarqube rules inducing bugs?, in: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 501–511.

Lenarduzzi, V., Palomba, F., Taibi, D., Tamburri, D.A., 2020b. Openszz: A free, open-source, web-accessible implementation of the szz algorithm, in: Proceedings of the 28th IEEE/ACM International Conference on Program Comprehension, p. To appear.

Marinescu, P., Hosek, P., Cadar, C., 2014. Covrig: A framework for the analysis of code, test, and coverage evolution in real software, in: Proceedings of the 2014 international symposium on software testing and analysis, pp. 93–104.

Misirli, A.T., Shihab, E., Kamei, Y., 2016. Studying high impact fix-inducing changes. Empirical Software Engineering 21, 605–641.

Neto, E.C., da Costa, D.A., Kulesza, U., 2018. The impact of refactoring changes on the szz algorithm: An empirical study, in: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 380–390.

Neto, E.C., da Costa, D.A., Kulesza, U., 2019. Revisiting and improving szz implementations, in: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE. pp. 1–12.

Oliveto, R., Gethers, M., Poshyvanyk, D., Lucia, A.D., 2010. On the equivalence of information retrieval methods for automated traceability link recovery, in: Proceedings of the 18th IEEE International Conference on Program Comprehension, IEEE Computer Society. pp. 68–71.

Pace, J., 2007. A tool which compares java files based on content. Http://www.incava.org/projects/java/diffj.

Palomba, F., Bavota, G., Di Penta, M., Fasano, F., Oliveto, R., De Lucia, A., 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. Empirical Software Engineering 23, 1188–1221.

Pascarella, L., Palomba, F., Bacchelli, A., 2019. Fine-grained just-in-time defect prediction. Journal of Systems and Software 150, 22–36.

Posnett, D., D'Souza, R., Devanbu, P., Filkov, V., 2013. Dual ecological measures of focus in software development, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE. pp. 452–461.

Prechelt, L., Pepper, A., 2014. Why software repositories are not used for defect-insertion circumstance analysis more often: A case study. Information and Software Technology 56, 1377–1389.

Rahman, F., Posnett, D., Hindle, A., Barr, E., Devanbu, P., 2011. Bugcache for inspections: hit or miss?, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, pp. 322–331.

Rodríguez-Pérez, G., Robles, G., González-Barahona, J.M., 2018. Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the szz algorithm. Information and Software Technology 99, 164–176.

Rodríguez-Pérez, G., Robles, G., Serebrenik, A., Zaidman, A., Germán, D.M., Gonzalez-Barahona, J.M., 2020. How bugs are born: a model to identify how bugs are introduced in software components. Empirical Software Engineering , 1–47.

Rosa, G., Pascarella, L., Scalabrino, S., Tufano, R., Bavota, G., Lanza, M., Oliveto, R., 2021. Evaluating szz implementations through a developer-informed oracle, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), IEEE. pp. 436–447.

Rosa, G., Pascarella, L., Scalabrino, S., Tufano, R., Bavota, G., Lanza, M., Oliveto, R., TBD. Replication Package for "A Comprehensive Evaluation of SZZ Variants Through a Developer-informed Oracle". `https://figshare.com/s/557bf69cfc3cea64e8ef`.

Sahal, E., Tosun, A., 2018. Identifying bug-inducing changes for code additions, in: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pp. 1–2.

Silva, D., Valente, M.T., 2017. Refdiff: detecting refactorings in version histories, in: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), IEEE. pp. 269–279.

Śliwerski, J., Zimmermann, T., Zeller, A., 2005. When do changes induce fixes? ACM sigsoft software engineering notes 30, 1–5.

Spadini, D., Aniche, M., Bacchelli, A., 2018. PyDriller: Python framework for mining software repositories, in: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018, ACM Press, New York, New York, USA. pp. 908–911. URL: http://dl.acm.org/citation.cfm?doid=3236024.3264598, doi:10.1145/3236024.3264598.

Tan, M., Tan, L., Dara, S., Mayeux, C., 2015. Online defect prediction for imbalanced data, in: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, IEEE. pp. 99–108.

Tóth, Z., Gyimesi, P., Ferenc, R., 2016. A public bug database of github projects and its application in bug prediction, in: Computational Science and Its Applications – ICCSA 2016, Springer International Publishing. pp. 625–638.

Tsantalis, N., Ketkar, A., Dig, D., 2020. Refactoringminer 2.0. IEEE Transactions on Software Engineering doi:10.1109/TSE.2020.3007722.

Tsantalis, N., Mansouri, M., Eshkevari, L., Mazinanian, D., Dig, D., 2018. Accurate and efficient refactoring detection in commit history, in: 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), IEEE. pp. 483–494.

Tu, H., Yu, Z., Menzies, T., 2020. Better data labelling with emblem (and how that impacts defect prediction). IEEE Transactions on Software Engineering .

Tufano, M., Bavota, G., Poshyvanyk, D., Di Penta, M., Oliveto, R., De Lucia, A., 2017. An empirical study on developer-related factors characterizing fix-inducing commits. Journal of Software: Evolution and Process 29, e1797.

Tufano, M., Watson, C., Bavota, G., Penta, M.D., White, M., Poshyvanyk, D., 2019. An empirical study on learning bug-fixing patches in the wild via neural machine translation. ACM Trans. Softw. Eng. Methodol. 28, 19:1–19:29.

Wehaibi, S., Shihab, E., Guerrouj, L., 2016. Examining the impact of self-admitted technical debt on software quality, in: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE. pp. 179–188.

Wen, M., Wu, R., Cheung, S.C., 2016. Locus: Locating bugs from software changes, in: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE. pp. 262–273.

Williams, C., Spacco, J., 2008a. Szz revisited: verifying when changes induce fixes, in: Proceedings of the 2008 workshop on Defects in large software systems, pp. 32–36.

Williams, C.C., Spacco, J.W., 2008b. Branching and merging in the repository, in: Proceedings of the 2008 international working conference on Mining software repositories, pp. 19–22.

Yan, M., Xia, X., Fan, Y., Hassan, A.E., Lo, D., Li, S., 2020. Just-in-time defect identification and localization: A two-phase framework. IEEE Transactions on Software Engineering .