Evaluating SZZ Implementations Through a Developer-informed Oracle

Giovanni Rosa*, Luca Pascarella†, Simone Scalabrino*, Rosalia Tufano†, Gabriele Bavota†,
Michele Lanza†, and Rocco Oliveto*
*University of Molise, Italy
†Software Institute @ USI Università della Svizzera italiana, Switzerland

***Abstract*—The SZZ algorithm for identifying bug-inducing changes has been widely used to evaluate defect prediction techniques and to empirically investigate when, how, and by whom bugs are introduced. Over the years, researchers have proposed several heuristics to improve the SZZ accuracy, providing various implementations of SZZ. However, fairly evaluating those implementations on a reliable oracle is an open problem: SZZ evaluations usually rely on (i) the manual analysis of the SZZ output to classify the identified bug-inducing commits as true or false positives; or (ii) a golden set linking bug-fixing and bug-inducing commits. In both cases, these manual evaluations are performed by researchers with limited knowledge of the studied subject systems. Ideally, there should be a golden set created by the original developers of the studied systems.**

We propose a methodology to build a "developer-informed" oracle for the evaluation of SZZ variants. We use Natural Language Processing (NLP) to identify bug-fixing commits in which developers explicitly reference the commit(s) that introduced a fixed bug. This was followed by a manual filtering step aimed at ensuring the quality and accuracy of the oracle. Once built, we used the oracle to evaluate several variants of the SZZ algorithm in terms of their accuracy. Our evaluation helped us to distill a set of lessons learned to further improve the SZZ algorithm.

***Index Terms*—SZZ, Defect Prediction, Empirical Study**

I. Introduction

The SZZ algorithm, proposed by Śliwerski, Zimmermann, and Zeller [1] at MSR 2005, identifies, given a bug-fixing commit C_{BF}, the commits that likely introduced the bug fixed in C_{BF}. These commits are termed "bug-inducing" commits. In essence, given C_{BF} as input, 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}.

SZZ has been widely adopted to (i) design and evaluate defect prediction techniques [2]–[6], and to (ii) run empirical studies aimed at investigating under which circumstances bugs are introduced [7]–[10]. The relevance of the SZZ algorithm was recognized a decade later with a MIP (Most Influential Paper award) presented at the 12th Working Conference on Mining Software Repositories (MSR 2015).

Several researchers have proposed variants of the original algorithm, with the goal of boosting its accuracy [11]–[16].

For example, one issue with the basic SZZ implementation is that it considers changes to code comments and whitespaces like any other change. This means that if a comment is modified in C_{BF}, the latest change to that comment is mistakenly considered as a bug-inducing commit. An improvement by Kim *et al.* [11] was therefore to ignore changes to code comments and blank lines as candidate bug-inducing commits.

Despite the major advances made on the accuracy of SZZ, Alencar da Costa *et al.* [14] highlighted the major difficulties in fairly evaluating and comparing the SZZ variants proposed in the literature. They observed that the studies presenting and evaluating SZZ variants mostly rely on manual analysis of a small sample of SZZ results [1], [11]–[13], with the goal of evaluating its accuracy. Such an evaluation is usually performed by the researchers who—not being the original developers of the studied systems—do not always have the knowledge needed to correctly identify the bug introducing commit. Also, due to the high cost of such a manual analysis, it is usually performed on a small sample of the identified bug-inducing commits. Other researchers built instead a ground truth to evaluate the performance of the SZZ algorithm [16]. However, also in these cases, the ground truth is produced by the researchers. Alencar da Costa *et al.* [14] called for evaluations performed with "*domain experts (*e.g., *developers or testers)*" reporting however that "*such an analysis is impractical*" since "*the experts would need to verify a large sample of bug-introducing changes, which is difficult to scale up to the size of modern defect datasets*" [14].

We present a methodology to build a "developer-informed" oracle for the evaluation of SZZ implementations. To explain its idea, let us take as example commit `a8a97bd` from the `apache/thrift` GitHub project, accompanied by a commit message saying: "*THRIFT-4513: fix bug in comparator introduced by e58f75d*". The developer fixing the bug is explicitly documenting the commit that introduced such a bug. Based on this observation, we defined a number of strict NLP-based heuristics to automatically identify notes in bug-fixing commits in which developers explicitly reference the commit(s) that introduced the fixed bug. We applied these heuristics to a total of 19,603,736 mined through GH Archive [39], which archives all public events on GitHub.

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.

Approach name	Reference	Based on	Used by	Oracle type	# Projects	# Bug Fixes
B-SZZ	Śliwerski *et al.* [1]		[3], [4], [17]–[24]	//	//	//
AG-SZZ	Kim *et al.* [11]	B-SZZ	[2], [8], [25]–[31]	Manually defined (researchers)	2	301
DJ-SZZ	Williams and Spacco [12]	AG-SZZ	[6], [7], [32]–[37]	Manually defined (researchers)	1	25
L-SZZ & R-SZZ	Davies *et al.* [13]	AG-SZZ	[14]	Manually defined (researchers)	3	174
MA-SZZ	da Costa *et al.* [14]	AG-SZZ	[6], [9], [10], [15], [16], [38]	Automatically computed metrics	10	2,637
RA-SZZ	Neto *et al.* [15]	MA-SZZ	[5], [6], [15]	Manually defined (researchers)	10	365
RA-SZZ*	Neto *et al.* [16]	RA-SZZ	None	Manually defined (researchers)	10	365

TABLE I: 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).

We mined the time period between March 2011 and April 2020, obtaining 3,585 commits. To further increase the intrinsic quality of the dataset, we manually validated the 3,585 commits, to (i) verify if, from the commit message, it was clear that the developer was documenting the bug-inducing commit; and (ii) taking note of any issue referenced in the commit message (*e.g.,* issue THRIFT-4513 in the previous example). Information from the issue tracker is exploited by some of the SZZ implementations and we wanted our dataset to include it.

As output of this process, we obtained a dataset of 1,930 validated bug-fixing commits in which developers documented the commit(s) that introduced the bug, with 212 also including information about the fixed issue(s). To the best of our knowledge, our work is the first presenting a dataset for the SZZ evaluation built by using information about the bug-inducing commit(s) explicitly reported by the bug fixer.

We tested nine variants of SZZ on our dataset. Besides reporting their precision and recall, we analyzed their complementarity and focused on the set of bug-fixes where all SZZ variants fail. A qualitative analysis of those cases allowed to distill lessons learned useful to further improve the SZZ algorithm in the future. Summarizing, our contributions are:

1) A methodology to build a "developer-informed" oracle for the evaluation of SZZ implementations, which does not require major manual efforts as compared to the classical manual identification of bug-inducing commits.
2) A first, easily extensible dataset built using our methodology and featuring 1,930 validated bug-fixing commits.
3) An empirical study comparing the effectiveness of several SZZ implementations.
4) A comprehensive replication package featuring (i) the dataset, and (ii) the implemented SZZ variants [40].

II. Background and Related Work

We start by presenting several variants of the SZZ algorithm [1] proposed in the literature over the years. Then, we discuss how those variants have been used in SE research community.

A. SZZ and its variants

Table I 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.,* 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.* [1] (we refer to it as B-SZZ, following the notation provided by da Costa *et al.* [14]). 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.* [11] 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 [12] improved the AG-SZZ algorithm in two ways: first, they use a line-number mapping approach [41] instead of the annotation graph introduced by Kim *et al.* [11]; second, they use DiffJ [42], 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.* [13] 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.* [14], 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 [15] and RA-SZZ* [16]. 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 [43], while RA-SZZ* uses Refactoring Miner [44], with the second one being more effective [16].

The original SZZ was not empirically evaluated [1]. Instead, all its variants, except MA-SZZ, were manually evaluated by their authors. One of them, RA-SZZ* [16], used an external dataset, *i.e.,* Defect4J [45]. MA-SZZ was evaluated using automated metrics, namely *earliest bug appearance*, *future impact of a change*, and *realism of bug introduction* [14].

In Table II we list the open-source implementations of SZZ.

Tool name	Approach	Public repository
SZZ Unleashed [33]	∼DJ-SZZ [12]	https://github.com/wogscpar/SZZUnleashed
OpenSZZ [46]	∼B-SZZ [1]	https://github.com/clowee/OpenSZZ
PYDRILLER [47]	∼AG-SZZ [1]	https://github.com/ishepard/pydriller

TABLE II: Open-source tools implementing SZZ.

SZZ Unleashed [33] partially implements DJ-SZZ: it uses line-number mapping [12] but it does not rely on DiffJ [42] for computing diffs, also working on non-Java files. It does not take into account meta-changes [14] and refactorings [16].

OpenSZZ [46] implements the basic version of the approach, B-SZZ. Since it is based on the git `blame` command, it implicitly uses the annotated graph [11].

PYDRILLER [47], 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.* [11]. We do not report in Table II a comprehensive list of all the SZZ implementations that can be found on GitHub, but only the ones presented in papers.

B. 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.* [37], featuring 187 papers.

SZZ has been used to run several empirical investigations having different goals [7]–[10], [17], [18], [20], [22]–[25], [27]–[31], [35], [37]. For example, Aman *et al.* [9] studied the role of local variable names in fault-introducing commits and they used SZZ to retrieve such commits, while Palomba *et al.* [17] 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 [2]–[6], [19], [21], [26], [34], [38].

Looking at Table I it is worth noting that, despite its clear limitations [11], many studies, even recent ones, still rely on B-SZZ [3], [4], [17]–[24] (the approaches that use git implicitly use the annotation graph defined by Kim *et al.* [11]). 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 [33], [46].

The studies most similar to ours are the one by da Costa *et al.* [14] and the one by Rodríguez-Pérez *et al.* [36]. Both report a comparison of different SZZ variants. Da Costa *et al.* [14] 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 [14], which motivated our study. Rodríguez-Pérez *et al.* [37] 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.

III. BUILDING A DEVELOPER-INFORMED DATASET OF BUG-INDUCING COMMITS

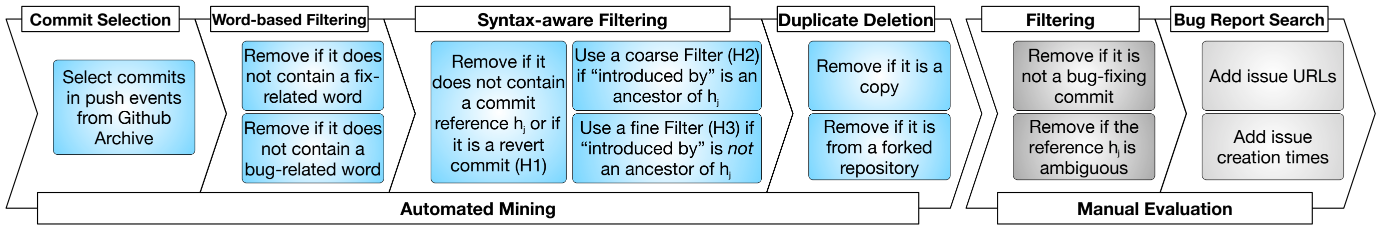
We present a methodology to build a dataset of bug-inducing commits by exploiting information provided by developers when fixing bugs. Our methodology reduces the manual effort required for building such a dataset and more important, does not assume technical knowledge of the involved source code on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

A. 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 [48]: issues labeled with "bug", "defect", etc. are mined from the issue tracking system, storing their issue ID (*e.g.,* THRIFT-4513). 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.



Commit Selection
Select commits in push events from Github Archive
Word-based Filtering
Remove if it does not contain a fix-related word
Remove if it does not contain a bug-related word
Syntax-aware Filtering
Remove if it does not contain a commit reference h_j or if it is a revert commit (H1)
Use a coarse Filter (H2) if "introduced by" is an ancestor of h_j
Use a fine Filter (H3) if "introduced by" is not an ancestor of h_j
Duplicate Deletion
Remove if it is a copy
Remove if it is from a forked repository
Automated Mining
Filtering
Remove if it is not a bug-fixing commit
Remove if the reference h_j is ambiguous
Bug Report Search
Add issue URLs
Add issue creation times
Manual Evaluation

Fig. 1: Process used for building the dataset.

As a preliminary step, we mined GH ARCHIVE [39] 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 April 30^{th} 2020, extracting 19,603,736 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.

1) Word-Based Bug-Fixing Selection: To identify bug-fixing commits, we first apply a lightweight regular expression on all the commits we gathered, as done in previous work [49], [50]. We mark as potential bug-fixes all commits accompanied by a message including at least a fix-related word[1] and a bug-related word[2]. 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).

2) Syntax-Aware Filtering: We needed to select from the set of candidate bug-fixing commits only the ones in which developers likely documented the bug-inducing commit(s). 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[3] 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*".

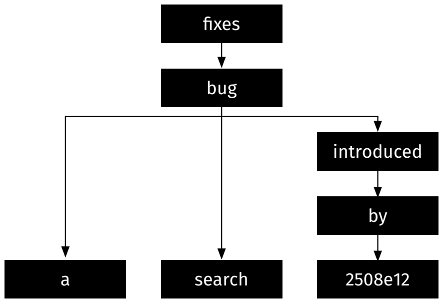


Fig. 2: Example of word dependency tree built by SPACY.

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 a regular expression[4]. 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[1] **and** one of the bug-related words[2] 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.

[1] *fix* or *solve* [2] *bug*, *issue*, *problem*, *error*, or *misfeature* [3] https://spacy.io/ [4] [0-9a-f]{6,40}

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 it is a false positive.

3) Duplicate Deletion: 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 19,603,736 parsed commits, the automated filtering selected 3,585 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.

B. Manual Analysis

Four of the authors (from now on, evaluators) manually inspected the 3,585 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 [33] and OpenSZZ [46] 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 7,170 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.

C. The Obtained SZZ Oracles

Out of the 3,585 manually validated commits, 1,930 (55.6%) 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). 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 1,930 instances we report have been manually checked as true positive bug-fixes, and only 212 of these (11.0%) mention the fixed issue. The dataset is available in our replication package [40].

These 1,930 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 IV) 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 [40]. Table III summarizes the main characteristics of the *overall* dataset and of the *language-filtered* one.

Language	Overall #Repos	Overall #Commits	Overall #Issues	Language-filtered #Repos	Language-filtered #Commits	Language-filtered #Issues
C	350	433	52	297	366	41
Python	271	304	36	249	279	35
C++	198	241	31	138	162	20
JS	169	180	26	127	135	18
Java	88	101	14	72	80	10
PHP	63	71	6	56	64	5
Ruby	43	47	5	36	37	4
C#	25	32	3	20	27	1
Others	498	588	48	0	0	0
Total	1,625	1,930	212	951	1,115	129

TABLE III: 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.

IV. Study Design

The *goal* of this study is to experiment several implementations of the SZZ algorithm on the previously defined *language-filtered* dataset (*context* of our study). 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:

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

A. Data Collection

We focused our experiment on several variants of the SZZ algorithm. Specifically, we (i) re-implemented all the main approaches available in the literature (presented in Section II) in a new tool, and (ii) adapted three existing tools (PyDriller [47], SZZ Unleashed [33], and OpenSZZ [46]) to work with our dataset. We provide in our replication package [40] both our tool and the adapted versions of the other tools, including detailed instructions on how to run them.

We report the details about all the implementations we compare in Table IV 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.

It is worth noting that we intentionally made all our re-implementations optionally independent from the issue-tracker systems: we did this because most of the instances of our dataset do not provide links to the bug-report ($\sim$88%). This is the reason why we did not implement the "Issue-date" as a BIC filtering technique by default, despite it is reported in the respective papers (*e.g.,* for B-SZZ). However, we experiment all techniques with and without such a filtering applied.

As for the 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. On the other hand, we implemented wrappers for such tools that allowed us to run them with our dataset. 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 IV 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).

To evaluate the previously described implementations, we defined two datasets extracted from the *language-filtered* dataset: (i) the $oracle_{all}$ dataset, featuring 1,115 bug-fixes, which includes both the ones with and without issue information, and (ii) the $oracle_{issues}$ dataset, featuring 129 instances, which includes only instances with issue information. Also, we defined two additional datasets, $oracle^{J}_{all}$ (80 instances) and $oracle^{J}_{issues}$ (10 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*[5] and OpenSZZ, only work on Java files.

We ran all the implementations on all the datasets on which they can be executed (*i.e.,* we did not run RA-SZZ* and OpenSZZ on the datasets including non-Java files). It is worth noting that SZZ Unleashed requires the issue date in order to work, so it would not be possible to run it on the $oracle_{all}$ dataset. To avoid this problem, 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).

B. Data Analysis

Given the defined oracle and the set of bug-inducing commits detected by the experimented implementations, we evaluated its accuracy by using two widely-adopted Information Retrieval (IR) metrics, namely recall and precision [52].

[5] It relies on Refactoring Miner [51] which only works on Java files.

TABLE IV: Characteristics of the SZZ implementations we compare in our study. 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 [11]	//	//	We use git `blame` instead of the CVS `annotate`, *i.e.,* we implicitly use an annotation graph [11]. We do not filter BICs based on the issue creation date.◇
AG-SZZ	Cosmetic changes [11]	Annotation Graph [11]	//	//	No differences.◇
MA-SZZ	Cosmetic changes [11]	Annotation Graph [11]	Meta-Changes [14]	//	No differences.◇
L-SZZ	Cosmetic Changes [11]	Annotation Graph [11]	Meta-Changes [14]	Largest [13]	We filter meta-changes [14].◇
R-SZZ	Cosmetic Changes [11]	Annotation Graph [11]	Meta-Changes [14]	Latest [13]	We filter meta-changes [14].◇
RA-SZZ*	Cosmetic Changes [11] Refactorings [16]	Annotation Graph [11]	Meta-Changes [14]	//	We use Refactoring Miner 2.0 [51].◇
SZZ@PYD	Cosmetic Changes [11]	Annotation Graph [11]	//	//	We implement a wrapper for PyDriller [47].
SZZ@UNL	Cosmetic Changes [11]	Line-number Mapping [12]	Issue-date [1]	//	We implement a wrapper for SZZ Unleashed [33].
SZZ@OPN	//	Annotation Graph [11]	//	//	We implement a wrapper for OpenSZZ [46].

We computed them 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 [52], defined as the harmonic mean of precision and recall. Such metrics were also used in previous work for evaluating SZZ variants (*e.g.,* Neto *et al.* [16]).

Given the set of experimented SZZ variants/tools $SZZ_{exp} = \{v_1, v_2, \ldots v_n\}$, we also analyze their complementarity by computing the following metrics for each v_i [53]:

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

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

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 all the pairs of SZZ variants and present the results using a heatmap. $correct_{v_i \setminus (SZZ_{exp} \setminus v_i)}$, instead, measures the correct bug-inducing commits identified only by technique v_i and missed by all others.

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 in our analysis, 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). We qualitatively discuss these cases to understand their peculiarities and point to future improvements of the SZZ algorithm.

V. Results Discussion

Table V reports the results achieved by the experimented SZZ variants and tools.

TABLE V: Precision, recall, and F-measure calculated for all SZZ algorithms. † Java only files.

	Algorithm	$oracle_{all}$ Recall	$oracle_{all}$ Precision	$oracle_{all}$ F1	$oracle_{issue}$ Recall	$oracle_{issue}$ Precision	$oracle_{issue}$ F1
No issue date filter	B-SZZ	0.69	0.39	0.50	0.69	0.38	0.49
	AG-SZZ	0.60	0.45	0.52	0.62	0.43	0.51
	L-SZZ	0.45	0.52	0.48	0.43	0.49	0.46
	R-SZZ	0.57	0.66	0.61	0.56	0.64	0.60
	MA-SZZ	0.64	0.36	0.46	0.65	0.36	0.47
	†RA-SZZ*	0.45	0.35	0.39	0.40	0.57	0.47
	SZZ@PYD	0.67	0.39	0.49	0.68	0.39	0.50
	SZZ@UNL	0.72	0.09	0.16	0.72	0.06	0.12
	†SZZ@OPN	0.19	0.32	0.24	0.10	0.50	0.17
With date filter	B-SZZ	0.69	0.42	0.53	0.69	0.39	0.50
	AG-SZZ	0.60	0.49	0.54	0.62	0.44	0.52
	L-SZZ	0.45	0.54	0.49	0.43	0.50	0.46
	R-SZZ	0.57	0.73	0.64	0.56	0.67	0.61
	MA-SZZ	0.64	0.39	0.48	0.65	0.37	0.47
	†RA-SZZ*	0.45	0.43	0.44	0.40	0.57	0.47
	SZZ@PYD	0.67	0.42	0.52	0.68	0.41	0.51
	SZZ@UNL	0.72	0.09	0.16	0.72	0.06	0.12
	†SZZ@OPN	0.19	0.33	0.24	0.10	0.50	0.17

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 not have any impact on recall (since the discarded bug-inducing commits should all be false positives) while increasing precision. The left part of Table V shows the results achieved on $oracle_{all}$, while the right part focuses on $oracle_{issue}$.

The first result to extrapolate from Table V is the general trend concerning the performance of the SZZ implementations.

When not using the issue date filtering (top part), the highest achieved F-Measure is 61% (R-SZZ). R-SZZ uses the annotation graph, ignores cosmetic changes and meta-changes, and only considers as bug-inducing commits the latest change that impacted a line changed to fix the bug.

Such a combination of heuristics make the R-SZZ the most precise on both oracles, achieving a 66% precision on $oracle_{all}$ and 64% on $oracle_{issue}$. With respect to recall/precision tradeoff, there is a price to pay in terms of recall that, however, it is not dramatically worse compared to the best approach in terms of recall: SZZ@UNL (SZZ Unleashed). The latter achieves a 72% recall on both $oracle_{all}$ and $oracle_{issue}$ datasets, with, however, a 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 67% and 18%, respectively on $oracle_{all}$, and 67% and 17% on $oracle_{issue}$. By lowering the outlier threshold to 10 bug-inducing, the precision grows in both datasets to 24%. 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 assuming 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.,* 80 in $oracle_{all}$ and only 10 in $oracle_{issue}$). Thus, failing on a few of those cases penalizes in terms of performance metrics. Still, we found the low precision of RA-SZZ* surprising, considering the expensive mechanism it uses to limit false positives (*i.e.,* ignoring lines impacted by refactoring operations detected by Refactoring Miner [51].

B-SZZ, the simplest SZZ version, exhibits a good recall of 69% on both datasets, making it the second-best algorithm after SZZ@UNL. Nonetheless, B-SZZ pays in precision, making it the fourth algorithm together with the PyDriller implementation for $oracle_{all}$ and the fifth for $oracle_{issue}$. The similarity between B-SZZ and the PyDriller implementation results in very similar performances.

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 V shows the results achieved by the same algorithms when using the issue data filter.

As expected, the recall remains equal to the previous scenario in all cases, with marginal improvements in precision (thanks to the removal of some false positives). While most of the algorithms improve their precision by 1%-4%, two algorithms obtain substantial improvements in the $oracle_{all}$ dataset: RA-SZZ* (+8%) and R-SZZ (+7%).

This boosts the latter to a very good 73% precision on $oracle_{all}$, and 67% 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). SZZ@UNL ensures instead a high recall at, however, a high precision cost. 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 V, it is clear that there are still margins of improvement for the accuracy of the SZZ algorithm. We discuss possible directions for future work in Section V-A.

Table VI shows the $correct_{v_i \setminus (SZZ_{exp} \setminus v_i)}$ metric we computed for each SZZ variant v_i.

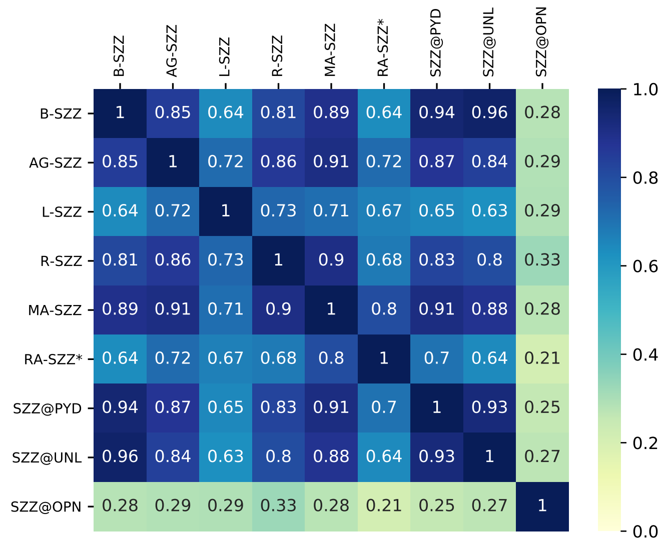
TABLE VI: Bug inducing commits correctly identified exclusively by the v_i algorithm. † Java only files.

Algorithm	No date filter		With date filter	
	$oracle_{all}$	$oracle_{issue}$	$oracle_{all}$	$oracle_{issue}$
B-SZZ	0/804	0/94	0/804	0/94
AG-SZZ	0/804	0/94	0/804	0/94
L-SZZ	0/804	0/94	0/804	0/94
R-SZZ	0/804	0/94	0/804	0/94
MA-SZZ	0/804	0/94	0/804	0/94
†RA-SZZ*	0/56	0/7	0/56	0/7
SZZ@PYD	0/804	0/94	0/804	0/94
SZZ@UNL	20/804 (2.5%)	3/94 (3.2%)	20/804 (2.5%)	3/94 (2.2%)
†SZZ@OPN	0/56	0/7	0/56	0/7

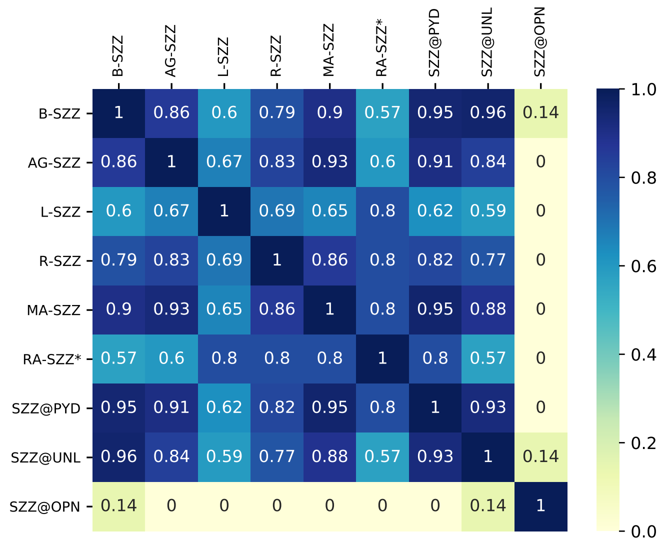
This metric measures the correct bug-inducing commits identified only by technique v_i and missed by all the others.

Fig. 3a and Fig. 3b depicts 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 are symmetric (*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 (20 on $oracle_{all}$ and 3 on $oracle_{issue}$) – Table VI. This is not surprising considering the high SZZ@UNL recall and the high number of bug-inducing commits it returns for certain bug-fixes. It also explains why none of the other implementations identifies bug-inducing commits missed by all the others: Given 804 as cardinality of the intersection of the true positives identified by all SZZ techniques, SZZ@UNL correctly retrieves 800 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, 21 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).

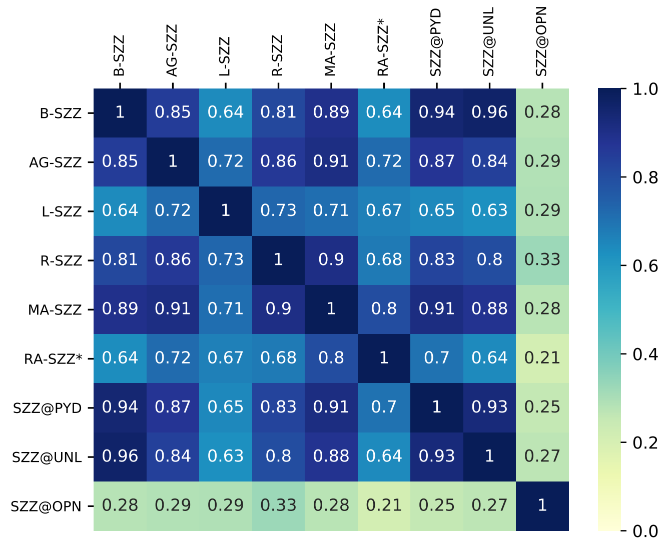


(a) *oracle*$_{all}$

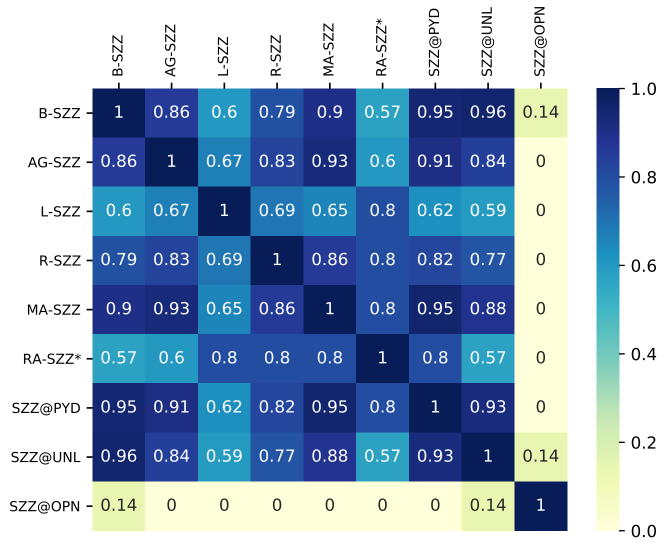


(b) *oracle*$_{issue}$

Fig. 3: Overlap between SZZ variants when no issue date filter is applied.



(a) *oracle*$_{all}$



(b) *oracle*$_{issue}$

Fig. 4: Overlap between SZZ variants when the issue date filter is applied.

We manually analyzed those cases to derive possible future improvements to the SZZ.

A. *Improvements to SZZ*

The manual analysis of 311 bug-fixing commits on which all SZZ variants fail allowed us to identify recurring patterns and distill three possible ways to improve the SZZ algorithm.

1) 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 [54] in which line 289 is modified to fix a bug introduced in commit `b67116d`, as pointed by the developer in the commit message. However, the bug was introduced with changes performed on line 290 [54]. 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.

However, by manually analyzing a large dataset of bug-fixing commits, it should be possible to identify fixing patterns with associated buggy lines. Such a dataset could be used to train a model able, given a bug-fixing commit, to point to the location of the bug.

2) SZZ is sensible to history rewritings: Bird *et al.* [55] highlighted some of the peril 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 [56], 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 dead end. Once run, 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 example, in the message of commit `5d8cee1` from the *xkb-switch* project [57], 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` [58], which is a revert commit. By performing an additional blame step, the correct bug-inducing commit pointed by the developer can be retrieved [59]. Future SZZ variants should handle revert commits, and properly deal with them when analyzing the change history.

3) Looking at the "big picture" in code changes: 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. There have been a recent attempt to address this problem: Sahal and Tosun [60] proposed an SZZ extension that considers past history of all the lines in the block in which the added line appears. However, the research in this aspect is still at the beginning.

An example is commit `ca11949` from the *snake* repository [61], 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` [62] 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. While this would help recall, it would penalize precision without careful heuristics aimed at filtering out false positives.

VI. 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 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 IV any difference between our implementations and the original proposals, and share our implementations [40]. Also, note that the differences documented in Table IV always aim at improving the performance of the SZZ variants and, thus, should not be detrimental for their performance.

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 1,930 instances for our oracle. Also, the SZZ implementations have been experimented on a smaller dataset of 1,115 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.

VII. Conclusion

When an algorithm like SZZ becomes so prominent in software engineering research, it is more than just necessary to explore ways to ameliorate its performance. Still, it is crucial to create a platform that allows for a sound and fair comparison of any new variant.

Our goal was to create such a platform, exemplified in a publicly available and extensible oracle of multiple and documented datasets, together with open source re-implementations of a considerable number of variants.

Moreover, as we used our oracle to compare the variants and check our re-implementation validity, we came up with several concrete improvements to the existing SZZ variants.

Given the pivotal role of SZZ for various research endeavors, for example, in the context of defect analysis and prediction, and the whole field of MSR (mining software repositories), we believe our work can set the stage for numerous and, above all, comparable ameliorations of the seminal SZZ algorithm.

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

[1] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

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

[3] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 2. IEEE, 2015, pp. 99–108.

[4] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *Journal of Systems and Software*, vol. 150, pp. 22–36, 2019.

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

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

[7] G. Bavota and B. Russo, "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, 2015, pp. 81–90.

[8] M. Tufano, G. Bavota, D. Poshyvanyk, M. Di Penta, R. Oliveto, and A. De Lucia, "An empirical study on developer-related factors characterizing fix-inducing commits," *Journal of Software: Evolution and Process*, vol. 29, no. 1, p. e1797, 2017.

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

[10] B. Chen and Z. M. J. Jiang, "Extracting and studying the logging-code-issue-introducing changes in java-based large-scale open source software systems," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2285–2322, 2019.

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

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

[13] S. Davies, M. Roper, and M. Wood, "Comparing text-based and dependence-based approaches for determining the origins of bugs," *Journal of Software: Evolution and Process*, vol. 26, no. 1, pp. 107–139, 2014.

[14] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2016.

[15] E. C. Neto, D. A. da Costa, and U. Kulesza, "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, 2018, pp. 380–390.

[16] ——, "Revisiting and improving szz implementations," in *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 2019, pp. 1–12.

[17] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. De Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1188–1221, 2018.

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

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

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

[21] S. Kim, E. J. Whitehead, and Y. Zhang, "Classifying software changes: Clean or buggy?" *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 181–196, 2008.

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

[23] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 179–188.

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

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

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

[27] J. Eyolfson, L. Tan, and P. Lam, "Correlations between bugginess and time-based commit characteristics," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1009–1039, 2014.

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

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

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

[31] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "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*, 2009, pp. 121–130.

[32] P. Marinescu, P. Hosek, and C. Cadar, "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*, 2014, pp. 93–104.

[33] M. Borg, O. Svensson, K. Berg, and D. Hansson, "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*, 2019, pp. 7–12.

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

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

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

[37] G. Rodríguez-Pérez, G. Robles, and J. M. González-Barahona, "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*, vol. 99, pp. 164–176, 2018.

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

[39] I. Grigorik, "GitHub Archive," https://www.githubarchive.org, 2012.

[40] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, *Replication package*, https://github.com/grosa1/icse2021-szz-replication-package.

[41] C. C. Williams and J. W. Spacco, "Branching and merging in the repository," in *Proceedings of the 2008 international working conference on Mining software repositories*, 2008, pp. 19–22.

[42] J. Pace, "A tool which compares java files based on content," 2007, http://www.incava.org/projects/java/diffj.

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

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

[45] R. Just, D. Jalali, and M. D. Ernst, "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*, 2014, pp. 437–440.

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

[47] D. Spadini, M. Aniche, and A. Bacchelli, "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*. New York, New York, USA: ACM Press, 2018, pp. 908–911. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3236024.3264598

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

[49] M. Fischer, M. Pinzger, and H. C. Gall, "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*, 2003, p. 23.

[50] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, "An empirical study on learning bug-fixing patches in the wild via neural machine translation," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 4, pp. 19:1–19:29, 2019.

[51] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020.

[52] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*. Addison-Wesley, 1999.

[53] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. D. Lucia, "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, 2010, pp. 68–71.

[54] *Adjacent fix commit example*, https://github.com/thpatch/thcrap/commit/29f16632f6bedd85e849034b01c3a8e8d4b7d83d.

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

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

[57] *Revert fix commit example*, https://github.com/grwlf/xkb-switch/commit/5d8cee18015b9a64aa3e06a81802f8186a99cc02.

[58] *Revert bug-inducing commit wrong*, https://github.com/grwlf/xkb-switch/commit/8b9cf29bca85076500ae5a2759f86e2042c527d0.

[59] *Revert bug-inducing commit correct*, https://github.com/grwlf/xkb-switch/commit/42abcc0da1c7f1062d069349edf90aa3b8832ca4.

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

[61] *Block bug-inducing commit wrong*, https://github.com/krmpotic/snake/commit/ca119496290f4ba8594c1e298a77336825c71e77.

[62] *Block bug-inducing commit wrong*, https://github.com/krmpotic/snake/commit/315a64b1bd627246b5a2c899ffdd47107d2b7fa6.