Bachelor's Thesis

LABELING COMMITS AS BUG FIXING AND FIX INDUCING: STRATEGIES AND EXPERIMENTS

SIMON FEDICK

Monday 21st June, 2021

Advisor:
Florian Sattler   Chair of Software Engineering

Examiners:
Prof. Dr. Sven Apel   Chair of Software Engineering
Prof. Dr. Jens Dittrich   Big Data Analytics Group

Chair of Software Engineering
Saarland Informatics Campus
Saarland University

Simon Fedick: *Labeling Commits as Bug Fixing and Fix Inducing: Strategies and Experiments,* © June 2021

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,______________________ ______________________________
(Datum/Date) (Unterschrift/Signature)

ABSTRACT

Automatically identifying known software bugs and their causes is one of the most crucial topics in software engineering research, as it enables the development of bug prediction models. Many software engineering researchers have attempted to solve this problem using the Sliwerski-Zimmermann-Zeller (SZZ) approach and proposed many improvements to the original structure since then. However, most SZZ implementations are restricted to projects using certain version control and issue tracking systems or even restricted to certain programming languages. The VaRA-Tool-Suite (VaRA-TS), as a framework that offers research tools available for any given project, represents a use case of SZZ where exactly this problem becomes obvious: Being able to extract the changes that fix or introduce bugs from a project is essential, but being able to provide bug detection strategies independent of a project's language or issue tracking system also is.

In this thesis, we implement an approach to SZZ that satisfies the needs of VaRA-TS and features improvements over the original SZZ approach where possible using two different fix detection strategies and corresponding introduction detection strategies. While one strategy offers more accurate data for projects using the issue tracking system integrated in GitHub, the other strategy always applies independently of the language or issue tracking system used. In the end, we evaluate our implementation by comparing it to other known fix detection strategies, such as the keyword matching strategy in the original SZZ approach, as well as to the open-source implementation SZZUnleashed. As a result, we see that our approach manages to generate more accurate bug data than the default SZZUnleashed version and that it is justified to only keyword match the first line of a commit message during the process.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1

INTRODUCTION

Automatically identifying known software bugs and their causes is one of the most crucial topics in software engineering research. The motivation behind automatically detecting the exact changes to a project that fixed or introduced bugs to the system is enabling bug prediction, and consequentially partial automation of code review. Many approaches to the automated classification of code changes have already been made, but fixing and introducing changes are very challenging to exactly pin down since intended behavior of another software system is nearly impossible to grasp in an automated context. Additionally, since manually labeling thousands of code changes with whether they actually fixed or introduced bugs is extremely infeasible, the correctness of such algorithms also is hard to confirm. Therefore, automated bug detection needs to put trust in the developer's own documentation on unwanted behavior that occurred during development. Although this is prone to produce inaccurate results, it is also a necessity to understand what the bugs of a project truly are.
State-of-the-art research mostly relies on the rule-based bug detection algorithm SZZ (Sliwerski-Zimmermann-Zeller) developed by Sliwerski et al. [12]. This algorithm relies on scanning the meta-data of code changes for keywords that indicate fixes and, starting from there, traces the code locations of fixes back to their original introduction. As the most well-known approach to this problem, many researchers have reproduced SZZ's results and found improvements to the original detection for fix-inducing changes. For instance, Williams et al. [15] and Kim et al. [3] proposed improvements to the tracing of modified lines across an entire history of changes, while Williams et al. [15] and Neto et al. [7] refined the approach for certain languages by identifying changes that do not alter the behavior of the system and hence should be neglected when looking for introducing changes. Remeli [10] also introduced an SZZ variant that is able to find introducing changes for bug fixes that do not modify the previous state of the code but only add entirely new lines to the structure.
Knowledge about problematic revisions of a software system that entailed one or even multiple fixes as well as the revisions that fixed a bug can also be decisive for explaining unusual code interactions throughout the history of changes. The framework VaRA-TS (Variability-aware Region Analyzer Tool-Suite) offers various tools to analyze such interactions between code regions through experiments, as well as visualize the results of those experiments. However, VaRA-TS employs no means to determine the exact changes that introduced or fixed bugs thus far. Using already established SZZ variants to resolve this is infeasible since most SZZ approaches only work for certain programming languages or on certain platforms that track the issues of a project, whereas VaRA-TS features a wide variety of issue tracking systems and languages. Implementing an easy method to detect these changes can significantly enhance the analysis of code interactions that previously missed context during future experiments. Hence, this motivates us to implement a method for VaRA-TS to access these changes conveniently, using an own variant of SZZ that fits the needs of the Tool-Suite.

1.1 GOALS

Our goal is to provide VaRA-TS with an SZZ approach that can be applied to all projects and provides bug data for all tools in the Tool-Suite. We aim to include improvements over the original algorithm where possible, yet our main priority is for the bug data to be available on any project with any issue tracking system, so we dismiss improvements that would otherwise restrict us in that aspect. Thus, we want to be able to detect bugs using only the information given by commit messages, since not every VaRA-TS project uses the same issue tracking system, and not every issue tracking system provides us with data regarding reported bugs in the same way. Furthermore, we want to evaluate our bug provider implementation in comparison to other alternatives using different fixing and introducing change detection methods.

1.2 CONTRIBUTIONS

We implement a bug provider for VaRA-TS, which is an interface that enables easy access to all bug fixing changes and their respective introducing changes on any given project. It employs a strategy to detect bug fixing changes by matching keywords in the description of changes, a non-mandatory fix detection strategy that detects fixes by the resolution of GitHub issues, in case the given project uses that API, and a strategy to detect introducing changes depending on the strategy that their fix has been found by. In addition to this, we carry out an analysis of our bug provider implementation in comparison to both the original fix detection method of the first SZZ approach as well as the introducing change detection of the open source project SZZUnleashed.

1.3 OVERVIEW

First, we establish essential background knowledge on version control systems like Git, issue tracking systems, the most relevant SZZ approaches introduced so far and the VaRA-Tool-Suite itself in Chapter 2. In Chapter 3, we go into detail about the implementation of our bug provider. After that, we evaluate our approach by comparing it to other fix detection approaches and the open source implementation SZZUnleashed by performing experiments on three VaRA-TS projects. The setup for these experiments is explained in Chapter 4, while we present and discuss the results in Chapter 5. In the end, we summarize our findings and propose ideas for future research in Chapter 6.

2 BACKGROUND

In this chapter, we explain the necessary background knowledge needed to understand the complexities and difficulties behind detecting the fixing and fix-inducing commits of a project. First off, we introduce basic concepts of version control and issue tracking in Section 2.1, so that we get a basic understanding of what exactly a change or a commit means in the context of versioning. As soon as we know those concepts, we talk about the most relevant approach to detecting the bug fixing and bug introducing changes called SZZ in Section 2.2. This approach has been reviewed and refined many times over, so we go into detail about some key problems of the original SZZ algorithm and how to deal with them in Section 2.3. In the end, we introduce the VaRA-Tool-Suite, the framework for which we implement our language- and issue tracking system independent bug detection approach in Section 2.4.

2.1 VERSION CONTROL AND ISSUE TRACKING SYSTEMS

If we want to extract the bug-related changes from an arbitrary project, we need sustainable information about each and every change that has been made to that project. If we lost information about changes once their modified lines have been touched again by a more recent change, we would be unable to reconstruct a lot of useful information, e.g. bug data, and reverting unwanted changes would also pose an impossible challenge. In order to meet these requirements, developers often make use of version control systems, the most well-known and commonly used example being Git.
Apart from the fact that we want to keep track of different versions of a project, we also need to be able to report and monitor known software errors, functionality that is yet to be implemented, insufficient documentation and other problems with the software, since a lot of these problems can occur at once and not every single one can be taken care of immediately and by the exact person that discovered the issue in the first place. This is what we need issue tracking systems for.
Issue tracking and version control systems are powerful tools that even are able to cooperate with each other in some ways. In this section, we get an understanding of these system's core functionalities and why they are such an important basis for any fully automated bug detection algorithm.

2.1.1 *Version Control Systems and the Usage of Git*

Version Control Systems (VCS) are a widely used method of managing projects. Seeing that bug detection research heavily relies on the assumption that we have access to different versions of a project at all times, we need to introduce some basic versioning concepts to understand how we can meet these requirements. The main VCS we use in this thesis is Git. Working directories where changes to any file are tracked by Git are called *repositories*.

```
    $ git log
commit 734713bc047d87bf7eac9674765ae793478c50d3
Author: Scott Chacon <schacon@gmail.com>
Date:   Fri Jan 2 18:32:33 2009 -0800

    Fix refs handling, add gc auto, update tests

commit d921970aadf03b3cf0e71becdaab3147ba71cdef
Merge: 1c002dd... 35cfb2b...
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 15:08:43 2008 -0800

    Merge commit 'phedders/rdocs'

commit 1c002dd4b536e7479fe34593e72e6c6c1819e53b
Author: Scott Chacon <schacon@gmail.com>
Date:   Thu Dec 11 14:58:32 2008 -0800

    Add some blame and merge stuff
```

Listing 2.1: Example Git Log (from chapter 7.1 in the GitHub docs[1])

Changes consist of an arbitrary number of modified lines in their respective files, where each modification is either an *insertion* or a *deletion*. In order to persist modifications, we can select files that contain modifications and add them to the current repository state to create a new snapshot of the repository state. These snapshots are called *commits* in Git terminology. Since each commit is a complete snapshot, each represents a revision of the project and therefore is assigned a distict *hash* consisting of alphanumeric characters as a means to uniquely identify different revisions. We call the insertions and deletions made between two such revisions the *diff*, and the diff of one specific commit usually refers to the diff between the most recent commit made before this commit, which we call the *parent commit*, and the commit itself. The most recent revision of the project is most commonly called the *HEAD (commit)*.
Commits provide various additional meta-data apart from their insertions and deletions, for example, they attach a corresponding *time stamp* of the commit's creation time and a committing *author*, which specifies the developer who added the commit to the project. All changes made to a project are registered inside the commit *history*, which can be visualized using the `git log` command. The log displays relevant information about the respective commits, such as the hash, the committing author and the date of the commit. (An example output of the Git log command can be found in Listing 2.1).
Clean documentation of a project is key to distinguish wanted from unwanted behavior and hence to be able to identify the bugs of a project. Thus, it is the responsibility of a committing author to add a respective *commit message* describing the purpose and contents of the modifications being made. There are barely any general conventions regarding the

structure of these messages, so bug detection heavily relies on the purpose of commits being properly documented.

2.1.2 *Issue Data*

Apart from being able to keep track of the version history of a project, developers use *Issue Tracking Systems* along with their Version Control Systems as a way to keep track of software bugs, features that are yet to be implemented and other issues with the software. Issue Tracking Systems are comparable to the concept of To-Do-Lists, since issues help with keeping track of various tasks that are still left to be done and issues can be closed once they have been taken care of, similar to checking off completed tasks on To-Do-Lists.
There is a wide variety of such systems, some popular examples being, e.g., the bug tracking system *BUGZILLA*, which was used in the initial version of the SZZ algorithm (more on this in Section 2.2) and Jira. For the purpose of this thesis, we use the terminology for the native Issue Tracking System implemented in GitHub.
Unlike the hashes used for commits, issues are usually just enumerated in the order of their opening date (i.e., the first issue created would be referred to as #1) in order to uniquely identify them. Issues consist of a *title* and a *description*, which elaborates on the problem and the work that needs to be done to resolve the issue. This can again be split up into a *task list* contained in the issue description and so as to resolve the issue, all tasks on the list need to be taken care of first.
As a means to tell bugs apart from features that are yet to be implemented or missing documentation, issues can be tagged with an arbitrary number of *labels*, whereas these labels can be, e.g., "bug", "important", or the language required to rectify the issue, to only name a few possibilities. Collaborators on a project can be assigned the task to resolve certain issues, making them *assignees* of the issue. As soon as an issue has been resolved, it can then be put in a *closed* state (as opposed to the initial *open* state) as well as *reopened*. Furthermore, issues store their initial *creation date* which also informs us about when the issue first was reported.
All relevant changes made to an issue are logged by GitHub as so-called *issue events*, which are an important concept to take note of for the implementation details described in Section 3.2.1. Issue events contain all information about the issue they refer to, as well as the actual type of *event* (e.g., "closed", "labeled" or "reopened"), a *date* of when the event occurred and an optional *commit hash*. As issue tracking systems are mostly linked to the version control system of the same project, issue numbers can be referred to inside of a commit message, triggering a certain type of event. The most interesting event is produced by closing an issue with a commit, which marks, given that the issue represented a bug, the version that *fixed* a bug. A similar non-automated version of this concept has already been made use of in the initial SZZ implementation with BUGZILLA, see Section 2.2.1.

1 https://git-scm.com/doc

2.2 THE ORIGINAL SZZ ALGORITHM

The SZZ Algorithm has initially been introduced by Sliwerski et al. [12] and acts as the most common basis for fully and partially automated bug data extraction algorithms to date. It aims to identify and provide information about the commits that are related to documented bugs. These commits always consist of one *fixing commit* that fixed the corresponding bug, i.e., the first revision that does not contain the bug, and arbitrarily many *fix-inducing commits*, or alternatively *introducing commits*. The latter type of changes led to their respective fix being authored, and therefore are considered to be the changes introducing the bug to the system.
In this section, we will start by taking a look at the original approach introduced by Sliwerski et al. The algorithm's concept consists of two essential steps. In the first step, the algorithm extracts all fixing commits, as described in Section 2.2.1. The code locations of the modifications made by fixing commits can be traced back to the roots of the fixed problem, which is what we do during the second phase described in Section 2.2.2.
The bug detection algorithms that are based on the original SZZ approach mostly keep the structure of these two steps and implemented various improvements to the steps themselves. We take a further look into the problems with this original approach and how other versions have dealt with them in Section 2.3.

2.2.1 *Classifying Changes as Fixes*

The multiple methods used to classify a commit as a *fixing commit* essentially boil down to two main strategies: Keyword matching and verifying cross-references to issues. Both of these strategies heavily rely on the project being thoroughly documented.
A very important thing to note here is that we identify each bug with one single and unique fix. Since bug reports using issue or bug tracking systems usually do not specify an associated location in the code, we use the insertions and deletions made by a fixing commit to find the problematic code regions. Our assumption also means that if a fixing commit fixed more than a single bug at once, we are not able to distinguish between these fixes in the associated code regions, thus we assume a bijective mapping between bugs and fixing commits.
The original SZZ approach made use of *confidence levels* as a heuristic to decide whether a commit is to be considered a fix. To be precise, we talk about two different types: *syntactic* and *semantic* confidence levels. When we define the semantic confidence level as *sem* and the syntactic confidence level as *syn*, we consider a commit a fixing commit if and only if the condition $(sem > 1) \lor (sem = 1 \land syn > 0)$ is satisfied, i.e., we want a sufficiently large semantic confidence level or a sufficiently large syntactic confidence level with a semantic confidence of 1. We discuss the details on how confidence levels were determined in the following sections.

KEYWORD MATCHING Keyword Matching is the idea of matching regular expressions to commit messages as a means to find certain descriptive terms indicating the intention of the respective commit. If certain words are found to be common practice amongst developers to use in their commit messages when they fixed a bug, we can be more confident that a

commit is a fix if its commit message contains these keywords. This technique even allows us to use stemming, which describes the process of reducing words to their word stem. That way, we are able to match regular expressions to all morphologically deductible forms of a word stem. This is a convenient method to deal with the fact that the tense in which developers write their commit messages are mostly up to their personal preference. For example, the regular expression `fix(e[ds])?` would match to the commit message "fixes bug #1" as well as "fix bug #1" and "fixed bug #1" and therefore, fit different commit message styles. Another factor that needs to be accounted for here is capitalized letters, such that commit messages starting off with "Fixed [...]", which are amongst the examples used in Chapter 3.2 of Sliwerski et al.'s research [12], would also match. Sliwerski et al. used the following regular expression to match keywords: `fix(e[ds])?|bugs?|defects?|patch`
Regular expressions also were used to match numbers and bug numbers inside a commit message. One way to raise the syntactic confidence level by one was for the commit message to either contain a keyword listed above or to only contain numbers or bug numbers. Intuitively, we do this since we assume that other keywords most likely imply another commit type than a bug fix, even if they are used in relation to numbers and bug numbers. However, these confidence level rules suggest that commit messages only containing numbers are most likely an enumeration of bugs fixed by the respective commit. This technique may even be applied to projects without needing to know the issue tracking system it made use of, as matching regular expressions to commit messages looking for certain keywords does not need additional external sources to verify that references refer to actual bugs.

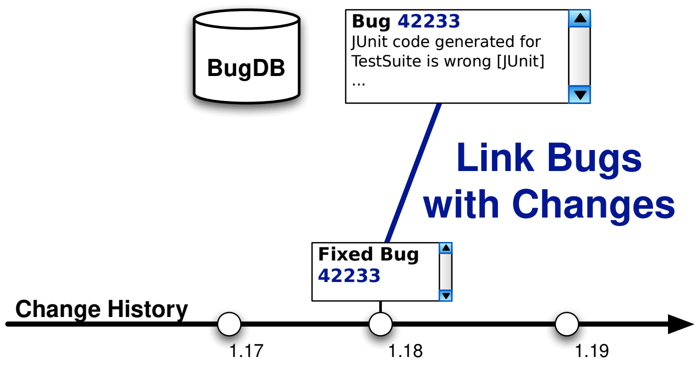

Figure 2.1: Change Referencing a Bug Number (taken from [12])

ISSUE CROSS-REFERENCES Just like we match regular expressions to commit messages in order to find certain keywords, regular expressions can be used to match issue or bug numbers in the syntax of the issue tracking system used. In Figure 2.1, we see how a commit message refers to a bug number that can be found in BUGZILLA's bug database. We want to verify these links that can be drawn by extracting the respective bug report data from the bug database.
Regarding syntactic confidence, Sliwerski et al. used multiple regexes to match different notations for bug numbers:

- `bug[# \t]*[0-9]+`

- `pr[# \t]*[0-9]+`
- `show\_bug\.cgi\?id=[0-9]+`
- `\[[0-9]+\]`

If either of these regular expressions found a match, it would be considered a bug number and raise the syntactic confidence level by one.
In order to make sure bug numbers actually referred to bugs inside the BUGZILLA database, Sliwerski et al. implemented semantic confidence levels. For semantic analysis, commits are linked to their respective bug report and form a tuple. Each of these tuples gets its semantic confidence level raised by one in each of the following cases:

- The bug report has been closed at least once.
- The committing author was assigned to the bug report.
- The short description of the bug report (≈ GitHub issue title) is contained inside the commit message.
- At least one of the committed files was linked inside the bug report.

These rules for semantic confidence, just like the rules for syntatic confidence, are subject to potential modifications to match the syntax and functionality, for example, of different issue tracking systems. Especially the last point refers to BUGZILLA functionality not implemented for GitHub issues, expressing the need for other rules if we implement this approach for different issue tracking systems.
Also, note that closing bugs via commit messages is not a feature implemented in BUGZILLA, so this could not be included as a confidence-raising condition in the original SZZ implementation while it promises to be a useful tool for our work.

RELIANCE ON DOCUMENTATION Without provided documentation in a given software system, we would be almost fully unable to distinguish expected from unexpected behavior. Although tests come close to defining the semantics of a project, it is unfeasible to find bugs by running tests for each revision. On one hand, we cannot assume tests to be defined around every reported bug, and on the other hand, very few projects reach full test coverage. Moreover, not all bug fixes are expected to also let the system pass more tests. In some cases, it can even be the opposite, e.g. if defect code segments could not be reached during test execution before a certain fix. Thus, we have to rely on what developers document to be bug fixes if we want to automatically extract bug data.
Bugs being documented in a respective issue tracking system also provides us with more information in comparison to bug fixes found only through keyword matching, e.g., the exact date when a bug was reported, which also helps us filter candidates for introducing commits (cf. Section 2.2.2). That way, we are able to rule out more false positives.

2.2.2 *How Changes Can Induce Fixes*

Each fix to a bug comes alongside changes that caused the fix to be necessary, i.e. fix inducing changes. The identification of fixes represents the first phase of the SZZ algorithm,

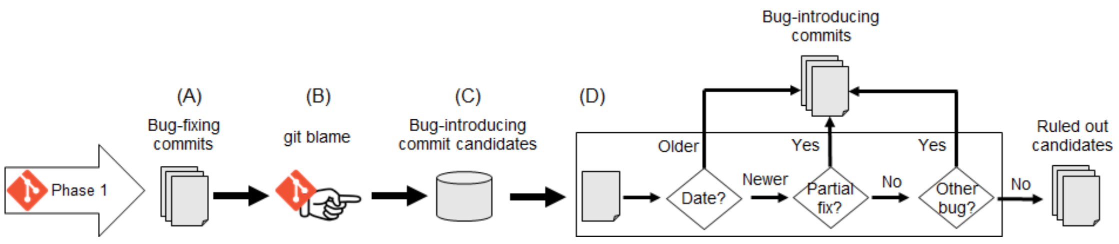

Figure 2.2: Second Phase of SZZ (taken from [1])

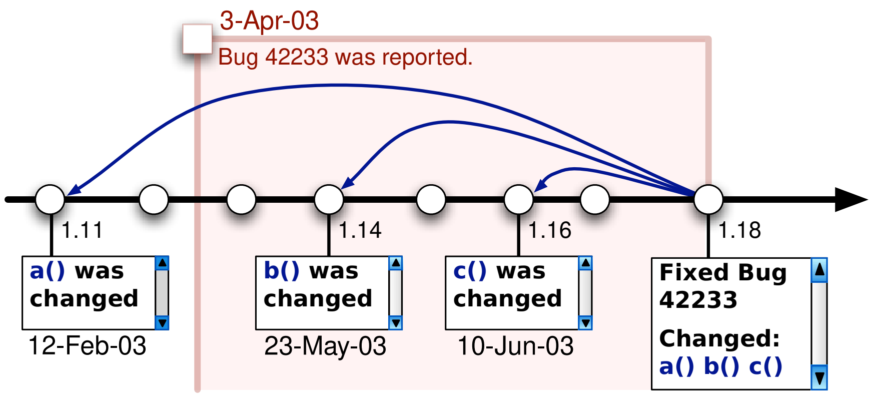

Figure 2.3: Candidates for Fix-Inducing Commits (taken from [12])

while the search for the corresponding fix-inducing changes is referred to as the second phase. Figure 2.2 illustrates a brief overview of the corresponding steps.
For each fix f identified in the first phase, within the code using the diff provided by our VCS. Using these locations, SZZ makes use of annotations, which correspond to the *blame* output in the case of Git. Applying this command to the parent commit of f (i.e., the last version without the fix), it provides us with a revision r that was the last commit to modify line l for each line l that has been touched by the fix itself. These revisions r are called *candidates* for fix-inducing changes and are stored in pairs (r, f) containing the fix f they induced. In the example given in Figure 2.3, the set of such candidate pairs would be $\{(1.11, 1.18), (1.14, 1.18), (1.16, 1.18)\}$. Note that the blame algorithm needs to be applied to the revision r_{pre} that is the last to not contain the fix since otherwise, f itself would be the most recent to touch each line l.

Having found all candidates, we assume all candidates to be true fix-inducing changes in case of a missing corresponding issue link due to the lack of sufficient information, as implied in Section 2.2.1. While this would not happen in the original SZZ approach by Sliwerski et al. since the semantic confidence level needs to be at least 1 for a commit to be considered a fix, the criteria for a fix regarding confidence levels can largely differ among different SZZ approaches. We need to be aware of this fact when identifying fixes by their syntactic confidence level only.
However, given the additional information of a linked issue, we are able to further categorize candidates by comparing their committer date to the report date of the linked issue. In case a candidate was committed before the bug was reported, we accept the candidate as a true

introducing commit. In Figure 2.3, this applies to revision 1.11 for the pair $(1.11, 1.18)$. We declare the remaining candidates as *suspects* since it is unclear whether these revisions are related to the bug at all if they happened only after the bug became known. The original SZZ algorithm divided these suspects into *weak suspects*, *hard suspects* and *partial fixes* using the following criteria:

- (r, f) is called a partial fix, iff r is a fix.
- (r, f) is called a weak suspect, iff there is a fix f' such that (r, f') is a non-suspect candidate.
- (r, f) is called a hard suspect, iff it can neither be classified a weak suspect nor a partial fix.

Each commit r in a pair (r, f) that is not considered a hard suspect at the end of the suspect classification is considered a true introducing commit. Weak suspects get redeemed since the candidate has other evidence of being fix-inducing, while partial fixes are considered previous attempts at fixing the bug or a related bug, and therefore also get redeemed. Hard suspects, however, have no redeeming factor and thus, we assume them to be unrelated to the fix.

2.3 ENHANCEMENTS TO THE SZZ ALGORITHM

The original SZZ algorithm provides a first approach to mine Git repositories for meaningful bug data. However, the algorithm had some significant weaknesses to be improved upon, some of them have already been suggested in the paper by Sliwerski et al. [12] itself. Since the classification criteria for introducing and fixing commits are purely dependent on documentation and do not consider the modified code itself, there are multiple ways to improve upon the original SZZ algorithm. In this section, we elaborate on the problems of SZZ and enhancements that can counter them.

2.3.1 *Line Annotation Strategies*

The annotation feature by CVS used in phase 2 of the original SZZ approach can be implemented in entirely different ways for other version control systems, as not all version control systems compute the diff of commits in the same way. Even more problematic is that the diff feature of version control systems does not always compute an accurate mapping of lines across revisions. While the diff and blame tools implemented in Git provide options that are sensitive to lines being moved across a file, less modern annotate features by version control systems like CVS, which was used in the original SZZ algorithm, only used to observe the history of certain line numbers and can lose track of moved lines that way. For example, when annotating a commit r_a, the diff of other revisions $r_0, ..., r_{a-1}$ with r_a does not consider the history of particular lines in between revisions $r_0, ..., r_{a-1}$. Therefore, there are various approaches aiming to deal with this problem and provide improved line number mappings.
Kim et al. [3] developed a solution called *Annotation Graphs*. These graphs introduce each line in a certain revision as a separate node, and compute how lines are being moved

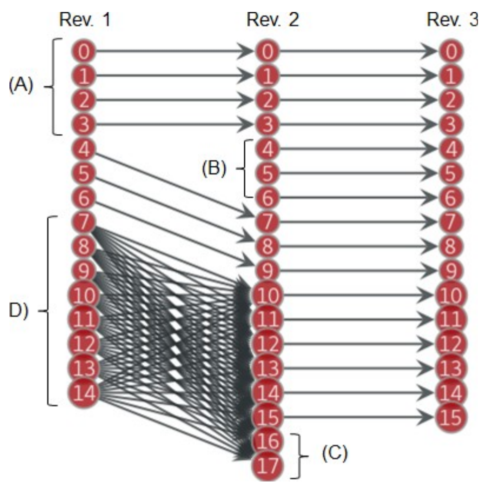

Figure 2.4: Annotation Graph Example (taken from [1])

between consecutive changes, such that in a revision pair (r_a, r_b) where r_b is the next change based on r_a, we add an edge between lines l_a of revision r_a and l_b of revision r_b if l_b resulted from modifications being made to l_a or if $l_a = l_b$. An example of such annotation graphs can be seen in Figure 2.4. Here, while lines 0-3 in hunk (A) stay at the same position across all three revisions, the lines of hunk (B) added in revision 2 have no parent node and therefore are assumed to be newly inserted. The lines 4-6 of revision 1 are moved accordingly in revision 2. If new lines 16 and 17 were added after revision 3, they also would not be associated with the lines deleted in hunk (C). Using this data structure, the inverse direction of these edges can be used to find the history of individual lines across revisions, improving the accuracy of search algorithms for fix-inducing changes.
The key problem with annotation graphs, however, is that it can be inaccurate at keeping track of moved lines in larger hunks of modifications even between two consecutive revisions, which is why Williams et al. later introduced a different approach called *probabilistic line number mapping* [14] [15]. This approach tracks lines per-class and per-method, assigning each pair of lines between consecutive revisions an edge weight determined by a heuristic method. These edge weights are computed using the normalized Levenshtein [5] edit distance, which is computed by dividing the actual edit distance by the maximum edit distance that would be possible. Since there are edge weights for each respective pair of lines, the mapping can then be computed by applying any algorithm that solves the minimum weight bipartite matching problem, as we want to minimize the edit distance for lines to be as similar as possible; Williams et al., in particular, used the Kuhn-Munkres approach to compute the perfect matching [4]. In order for entirely different lines not to get mapped to each other in case no better matching is available, for example to avoid deleted lines of the first revision getting matched with newly inserted lines in the second revision, only edges under a certain edge weight threshold are considered for the graph on which to compute the matching.

2.3.2 *Cosmetic Changes*

Another problem that arises when ignoring the meaning of the lines we analyze in the second phase of SZZ is that we sometimes consider changes fix-inducing that were purely cosmetic, while the true culprit was a non-cosmetic change that happened to the same line earlier. A change is for example considered a cosmetic change if it satisfies at least one of these conditions:

- The commit refactored function names, e.g., renaming the definition and each usage of function `foo()` to `bar()`.
- The commit refactored variable names, in the same manner as function names.
- The commit reordered or refactored function parameters, in the same manner as variable names.
- The commit modified a comment.
- The commit modified whitespaces or indentations.
- The commit modified import statements.

Note that this list is not necessarily exhaustive, as cosmetic changes can be defined in different ways. For instance, Fowler [9] defines refactoring as any kind of change that improves the design, but does not temper with the external behavior of the software system. Moreover, he suggests there are 63 different types of refactorings and provides an elaborate list of all types.
In the SZZ approach by Williams et al. [15], they use a tool specialized in recognizing these patterns inside Java projects called DiffJ[2]. This tool can produce a diff that is not purely textual, but rather detects Java syntax patterns and categorizes each modification into various modification types, producing a Java-aware diff. When using this diff algorithm, the blame algorithm that is used to detect the most recent change being made to each line in the fixing commit can blame the most recent commit that was not a cosmetic change according to DiffJ.
A different approach to this problem was introduced by Neto et al., who named it *refactoring-aware* SZZ (RA-SZZ) [7]. Similar to the method by Williams et al., RA-SZZ relies on RefDiff [11], which also is coined as a tool that produces refactoring-aware diff outputs. The tool supports the languages Java, JavaScript and C and claims to detect 13 different types of cosmetic changes according to the refactoring types of Fowler [8]. During later research, Neto et al. suggest that RefDiff produces an insufficiently accurate diff output and therefore, they include the tool RefactoringMiner, which was developed by Tsantalis et al. [13], into RA-SZZ [8]. RMiner claims to detect the 15 types of refactoring changes that are most common in practice and proves higher accuracy when compared to RefDiff. In the updated RA-SZZ method, Neto et al. incorporate both RefDiff and RMiner into their introducing commit pruning and manage to improve the accuracy of their second phase approach that way.
Many kinds of refactoring operations are claimed to be rather infeasible to detect in fully

2 https://github.com/jpace/diffj

```
1 int foo(E e){                   - int foo(E e){
2                                 +     if(e != nullptr)
3     e.bar();                    -         e.bar();
4 }                               - }
```

Listing 2.2: Example of an Insertion Only Fix

automated bug detection, such as semantically equivalent changes. For example, changing a line such as `int x = 2 + 2;` into `int x = 4;` would be considered a semantically equivalent change, however without any means to run a deep semantical analysis, researchers are yet to find a reliable filtering method for these types of refactoring operations.

2.3.3 *History of Insertion-Only Fixes*

The original second phase of the SZZ algorithm bases on the assumption that a fix entails only modifications to faulty lines since a change r_i is only considered fix-inducing if the fix r_f modified lines that revision r_i previously modified. However, there are fixes that require only insertions of new lines. In the example illustrated in Listing 2.2, we assume a parameter `e` of Type `E` that can potentially be a `nullptr`, and therefore not accounting for that fact when calling the function `e.bar()` is a defect. The fix only needs to add an if-statement to the previous line to account for this fact, though. Bugs of this kind do not need to be fixed in the same line in which they were introduced.
While in such a case, the original SZZ approach would blame commits that modified the line that previously had the same line number as the newly inserted line, the line number mappings described in Section 2.3.1 would most likely not detect a predecessor to the newly inserted line and thus, the fix would have an empty set of fix-inducing changes. The thesis by Remeli [10] examines this phenomenon and finds that for the project HIVE, around 28.91% of bug-fixing commits are insertion-only fixes and as such, they have an empty set of fix-inducing commits. As a solution, they propose an SZZ variant called *Enhanced SZZ*. This approach, while being one of the more modern SZZ approaches working with Git as VCS (which provides a cross-revision line number mapping similar to the approaches in Section 2.3.1 by default), also incorporate RefDiff to filter out cosmetic changes. In order to deal with insertion-only fixes and track the last changes to lines surrounding the fixing insertions, they also include one line above and below inserted lines to the blame algorithm. In Listing 2.2, this would mean to also look into the last revisions touching line 1 and 3, since the insertion was in line 2.

2.4 VARA-TOOL-SUITE

The Variability-aware Region Analyzer Tool-Suite (or short: VaRA-TS) is a framework for researchers that provides a selection of various pre-configured projects that can be automatically compiled, as well as a wide variety of tools that enable running experiments on these projects. VaRA-TS makes running experiments more convenient for researchers by offering helpful tools to automate different configuring, compiling and analyzing steps and working them into a single experiment abstraction. VaRA-TS is a public project that can be found on GitHub[3]. The implementation that we developed during this thesis, as it is described in Chapter 3, was contributed to VaRA-TS and can also can be found in the repository.

2.4.1 *The Project Interface*

The `Project` interface is an abstract representation of a software project that contains information which we need to be able to use in experiments and encapsulates the possibilities how VaRA-TS can interact with a project. As such, the interface offers information on how to checkout, configure and build the project behind it. If we want to run experiments on an arbitrary project, it needs a corresponding `Project` decleration in the Tool-Suite that satisfies the specifications of the interface first.
Furthermore, the project interface offers additional information about the concrete software project, such as a link to the repository, the repository's name and the group behind that project, so researchers can access this information during experiments. This information can then be used by the provider abstraction to compute additional meta-information that can then again be used for experiments.

2.4.2 *The Provider Interface*

Results of experiments run by VaRA-TS can be visualized in many different ways, such as graphs and tables, depending on the information that the researcher needs and how they need it to be presented. Oftentimes, extraordinary data points when analyzing code interactions can be explained further when annoted with meta-information, such as CVE and bug data. In this case, the provider interface corresponding to the exact type of data needed can offer this information to the experiment that lacks it.
The main purpose of a concrete `Provider` implementation is to provide additional information about a given project. For example, the `CVEProvider` can give us information about all publically known CVEs (i.e., cybersecurity vulnerabilities) of a project. Such a provider can also be realized for bug data, so the `BugProvider` that we want to implement offers data about all bugs and the revisions that introduced and fixed them. We elaborate on the exact functionality offered by the BugProvider interface in Section 3.1.1.

3 https://github.com/se-sic/VaRA-Tool-Suite

3 IMPLEMENTATION

In this section, we describe and explain our approach to incorporating an implementation of SZZ into the VaRA-Tool-Suite using the Provider interface. Our main goal for the `BugProvider` class implementation is for it to be a reusable interface that can be used by all kinds of experiments to provide bug data for any given project. We elaborate on the different options this interface offers in Section 3.1, as well as on how exactly we represent the bug datatype in the Tool-Suite. The exact methods that we use to find bugs by their fixing commits, i.e., the first phase of SZZ will be expanded on in Section 3.2, while we explain the second phase of our algorithm in Section 3.3. With the integration of our implementation into the VaRA-Tool-Suite and the reusability for multiple purposes come some limitations regarding the introducing commit filtering, which is why we were not able to include all SZZ improvements into our bug provider yet. Therefore, we included the open-source implementation SZZUnleashed into the Tool-Suite as an additional research tool. However, there were some adjustments to the original implementation that we had to make in order to get the tool to work with the Tool-Suite, which is what we cover in Section 3.4.

3.1 THE INTERFACES

First off, we want to take a look at the possibilities how we can interact with our provider and what information we can obtain from it and take a deeper look at the implementation realizing these interfaces afterward.

3.1.1 *The Bug Provider*

The `BugProvider` offers various functions that grant access to bug data depending on the use case, similar to other existing `Provider` classes in the Tool-Suite, such as the `CVEProvider`. After the provider has been created using the class of the project to analyze, various functions can be called that offer two different types of output formats for bug data each, depending on the information needed. Apart from the possibility to output all bugs of a project, we provide filter functions that enable searching for certain bugs by introducing or fixing commits. A sketch of the interface can be found in Listing 3.1.
Furthermore, when calling the `get_provider_for_project` method to create the bug provider, it will automatically check whether it uses Git as version control system, as well as if is a GitHub project in order to ensure whether the `IssueEvent` list offered by the GitHub API can be used to find bugs that are related to a respective GitHub issue.

Listing 3.1: Interface of the Bug Provider

```
class BugProvider(Provider):
        def find_all_pygit_bugs(self) -> tp.FrozenSet[bug.PygitBug]:
                [...]

        def find_all_raw_bugs(self) -> tp.FrozenSet[bug.RawBug]:
                [...]

        def find_pygit_bug_by_fix(self, fixing_commit: str)
                -> tp.FrozenSet[bug.PygitBug]:
                [...]

        def find_raw_bug_by_fix(self, fixing_commit: str)
                -> tp.FrozenSet[bug.RawBug]:
                [...]

        def find_pygit_bug_by_introduction(self, introducing_commit: str)
                -> tp.FrozenSet[bug.PygitBug]:
                [...]

        def find_raw_bug_by_introduction(self, introducing_commit: str)
                -> tp.FrozenSet[bug.RawBug]:
                [...]
```

3.1.2 *PygitBugs and RawBugs*

The output format for bugs in the bug provider divides itself into two classes `PygitBug` and `RawBug`, which mainly differ in the information they provide about the fixing and fix inducing commits. While a `RawBug` only provides either of these as string representations of commit hashes, a `PygitBug` provides them as `Commit` objects from the Pygit2 library[1], which contain more details about the commit, such as the commit message, the author and the committer date. Aside from the fixing commit and a set of associated introducing commits, both bug interfaces also contain the optional attributes `issue_id` representing the number of the associated issue, `creationdate` representing the date when the issue has been created and `resolutiondate` representing the date when the issue has been closed, in case there is a corresponding GitHub issue. These attributes are `None` if the bug has been found through commit messages only.
Since the `RawBug` class is the more abstract interface version of a `PygitBug`, a `PygitBug` additionally offers a method that converts itself to a `RawBug`, in case the supplementary information from the Pygit2 library is not needed or should be capsulated.
Note that since in this context, we understand a bug as a structure that maps one fixing commit to all of its introducing commits, our algorithm is structurally slightly different to the original SZZ algorithm, which outputs pairs of one fixing and one introducing commit

1 https://www.pygit2.org/

each, but allows fixing commits to occur multiple times. This implies not considering bug fixes without introducing commits as bugs, which our implementation allows.

3.2 FINDING FIX-INDUCING COMMITS

The `BugProvider` implements the first phase of the SZZ algorithm by combining the results of multiple fix detection approaches. First, in case the project repository is a GitHub repository, we download all `IssueEvent` s from the GitHub API and filter them for the events that closed issues representing a bug, as described in Section 3.2.1. Otherwise, we can always additionally find fixing commits by matching regular expressions to all messages in the commit history, even if the project has no remote repository on GitHub, which means that we are unable to work with issue data. This is what we want to show in Section 3.2.2. In the end, the results of both approaches are merged by adding all bugs stemming from issue events into the resulting set, but only adding commit message bugs if their fixing commit is not already associated with an issue event bug.

3.2.1 *Searching by Issue Events*

Our first approach to finding fixing commits is by analyzing issue data rather than the commit history, since the GitHub API offers an openly accessible method to obtain a full log of a project's issue events. As opposed to the approach in the original SZZ algorithm, our implementation does not need to cross-check issue links in the commit messages against actual issue data, but we rather profit off the way in which issue events are logged by GitHub. The `IssueEvent` class provides not only the type of event and the issue it is related to, but also an associated commit, in case the behavior has been caused by a certain commit. That way, if a commit closed an issue that was labeled as a bug, we can classify this commit as bug-fixing purely by analyzing issue data. However, it is important to note that in case the event type is `"closed"`, but there is no commit associated with it, we are unable to link any corresponding bug-fixing commit to the potential bug indicated by the issue event, and therefore have to neglect it.

Listing 3.2: Issue Event Filtering

```python
def _has_closed_a_bug(issue_event: IssueEvent) -> bool:
    if issue_event.event != "closed" or issue_event.commit_id is None:
        return False
    for label in issue_event.issue.labels:
        if label.name == "bug":
            return True
    return False
```

The code we use to realize this behavior for some given Issue Event, as it is displayed in Listing 3.2, is rather simple since the GitHub API provides us with exactly the data that we need. This method is made use of when creating `PygitSuspectTuple` objects associated with the fixing commit, which we explain in detail in Section 3.3.2.

3.2.2 *Searching by Commit Messages*

As our second fix detection approach, we want to detect bugs using keyword matching on commit messages, similar to the original SZZ approach [12]. The VaRA-Tool-Suite features many projects using many different issue tracking systems, so we would need to incorporate a separate algorithm for each system separately if we would like to include issue tracking data for each project. Moreover, not every issue tracking system provides an API like GitHub that we can take advantage of. Thus, we cannot be as strict on semantic confidence levels as the original SZZ implementation is. As such, the regular expression we use for identifying bugs cannot be the same as in the original implementation. Commits that reference bug numbers using GitHub syntax are already covered by the first identification method and cross-checking references to issues that stem from unknown issue tracking systems is infeasible, hence we only use regular expressions for keyword matching.
We say that we classify a commit as bug-fixing if its commit message contains the words "fix", "fixed" or "fixes" (in either lower-case or capitalized notation), which covers the same commit messages as a part of the regular expression used in the original SZZ notation, i.e., `fix(e[ds])?`. As suggested in the original SZZ implementation [12], we also just look into the first line of commit messages, assuming that line sums up the main purpose of the commit and therefore should contain the keyword in case it is a true fix.
By iterating over the whole commit history ordered by time, we check for each commit whether we want to classify it as bug-fixing using the method shown in Listing 3.3, which implements the behavior described above.

Listing 3.3: Commit Message Filtering

```
def _is_closing_message(commit_message: str) -> bool:
    first_line = commit_message.partition('\n')[0]

    return any([
        keyword in first_line.split()
        for keyword in ['fix', 'Fix', 'fixed', 'Fixed', 'fixes', 'Fixes']
    ])
```

3.3 FINDING INTRODUCING COMMITS

For the second phase of the SZZ algorithm, we rely on the help of PyDriller[2] to obtain the diff of a commit and blame the commits which last modified the deleted lines in the fix (Note that the modification of a line consists of the deletion of the old version and the insertion of the new version). Thus, the PyDriller library already implements the original second phase of SZZ aside from suspect classification[3]. Suspect classification is infeasible for bugs that we detected using commit messages only, but we are able to do so for bugs that we found using the issue event method in a similar manner to the original SZZ version.

2 https://pydriller.readthedocs.io/en/latest/index.html
3 https://pydriller.readthedocs.io/en/latest/reference.html?#pydriller.git.Git.get_commits_last_modified_lines

Therefore, the algorithm for finding introducing commits works very differently depending on the method we found the respective fixing commit with.
While many suggestions by other researchers to optimize the second phase of the SZZ algorithm do help with getting a more realistic estimation of bug-introducing commits, we were not able to include all enhancements so far. Although we already gain access to smarter line number mappings by using Git as version control system[4], improvements such as refactoring-aware diff algorithms are challenging to realize in the VaRA-TS environment, as these improvements are strongly language-dependent and can not be easily implemented for all languages featured in the Tool-Suite. Moreover, choosing which lines in the diff we want to blame is not an option when using PyDriller, which means we would need to work with a more customizable alternative if we wanted to implement the suggestion regarding insertion-only fixes by Remeli [10].

3.3.1 *Commit Message Bugs*

Fixing commits that we found through commit messages have a quite straightforward algorithm for finding their fix-inducing commits, as we are unable to apply further filtering without sufficient information provided by a corresponding issue. The function `create_corresponding_pygit_bug` searches, given the name of the project as well as the fixing commit, all commits that PyDriller can find blaming the diff of the fixing commit. The output by PyDriller yields a mapping of deleted lines to the respective commit hashes of the commits that last modified the deleted line. As shown in Listing 3.4, all commits corresponding to these hashes are added as introducing commits of the bug regardless of their corresponding line, as the explicit lines of the fixing commit they belong to do not concern us in this case.

Listing 3.4: PyDriller Blame Dictionary

```
blame_dict = pydrill_repo.get_commits_last_modified_lines(
    pydrill_repo.get_commit(closing_commit)
)

for _, introducing_set in blame_dict.items():
    for introducing_id in introducing_set:
        introducing_pycommits.add(
            project_repo.revparse_single(introducing_id)
        )
```

3.3.2 *Issue Event Bugs*

In case there is a GitHub issue that the fixing commit relates to, we can apply additional introducing commit filtering using suspects. While we gather the candidates for introducing commits in the same manner as shown in Listing 3.4, we do not assume all these commits

4 https://git-scm.com/docs/git-blame/2.31.0

to be fix-inducing immediately, but rather divide these commits into suspects and non-suspect candidates. After that, we can use this information to create *suspect tuple* objects as introduced in the following paragraph. As such, the code responsible for finding all candidates, although structurally similar to the commit message bugs, needs to check for each candidate whether it is a suspect or not by comparing the committer date of the blamed commit to the creation date of the issue. We can see how this is implemented in Listing 3.5.

Listing 3.5: Classifying Candidates Found by PyDriller

```
blame_dict = pydrill_repo.get_commits_last_modified_lines(
    pydrill_fixing_commit
)

non_suspect_commits = set()
suspect_commits = set()
for introducing_set in blame_dict.values():
    for introducing_id in introducing_set:
        issue_date = issue_event.issue.created_at
        introduction_date = pydrill_repo.get_commit(
            introducing_id
        ).committer_date
        if introduction_date > issue_date:
            suspect_commits.add(
                pygit_repo.revparse_single(introducing_id)
            )
        else:
            non_suspect_commits.add(
                pygit_repo.revparse_single(introducing_id)
            )
```

SUSPECT REPRESENTATION Similar to the structure of our bug classes, we do not interpret candidates as tuples of a fixing commit and an associated fix-inducing candidate. Instead, we create `PygitSuspectTuple` objects uniquely identified by their fixing commit, and we distinguish between three candidate sets for the introducing commits: *non-suspects* representing candidates that were committed before the issue has been created, *uncleared suspects* representing suspects for which we do not know yet whether they are hard suspects or not and *cleared suspects* for which we clearly know they are no hard suspects. Hard suspects get permanently removed from a suspect tuple once identified. This suspect class offers methods that help with suspect classification, i.e., it offers a method `extract_next_uncleared_suspect` that removes one uncleared suspect and returns it, a method `clear_suspect` that adds a given suspect which is confirmed to not be a hard suspect to the cleared suspects, a method `is_cleared` verifying whether all suspects inside the suspect tuple have been cleared yet and lastly, a method `create_corresponding_bug` that automatically uses its data to create a `PygitBug` once all suspects have been cleared or declared as hard suspects.

SUSPECT FILTERING In order to filter the suspects of each suspect tuple after we identified them, we implement the approach of Sliwerski et al.[12] by checking whether they are weak suspects or partial fixes. Checking for partial fixes is straightforward in this structure since we just need to check whether the currently inspected suspect is the fixing commit of any other suspect tuple. The case for weak suspects is also quite simple to implement, as we check for each suspect tuple whether the currently inspected suspect is contained in the set of non-suspects of this tuple.
The implementation details of this filtering method are displayed in Listing 3.6. After a suspect tuple has been fully cleared, we convert it to a `PygitBug` (or potentially a `RawBug` for output format purposes).

Listing 3.6: Suspect Classification

```python
for suspect_tuple in suspect_tuples:
    while not suspect_tuple.is_cleared():
        suspect = suspect_tuple.extract_next_uncleared_suspect()
        partial_fix = False
        weak_suspect = False

        # partial fix?
        for other_tuple in suspect_tuples:
            if suspect.hex == other_tuple.fixing_commit.hex:
                partial_fix = True
                break

        # weak suspect?
        if not partial_fix:
            for other_tuple in suspect_tuples:
                if suspect.hex in (
                    non_suspect.hex
                    for non_suspect in other_tuple.non_suspects
                ):
                    weak_suspect = True
                    break

        if partial_fix or weak_suspect:
            suspect_tuple.clear_suspect(suspect)
```

3.4 SZZUNLEASHED AS RESEARCH TOOL

While we are interested in comparing our SZZ approach to other SZZ implementations, finding suitable reference implementations by other researchers poses quite a challenge. Most times, we lack the source code or open-source implementations are infeasible to use due to the fact that their improved diff algorithms only can be used for certain languages, such as Java. There are few open-source SZZ implementations that are language-independent and therefore compatible with all projects provided by VaRA-TS, so we cannot use SZZ

variants relying on RefDiff[5] or DiffJ[6]. Hence, we settle for the open-source implementation SZZUnleashed developed by Borg et al. [1] that pledges to implement the improvements suggested by Williams et al. [15] except for the usage of DiffJ.
Apart from incorporating SZZUnleashed into VaRA-TS as a research tool, we correct some of its more severe implementation errors, e.g., there are instances where the original SZZUnleashed version available on GitHub[7] uses `==` to compare string objects, while in Java, this operator is used to compare the references of strings to each other rather than their values. Furthermore, SZZUnleashed can be run with a *depth parameter*. This parameter aims to represent the depth of the line mapping method proposed by Williams et al. [15] that we explained in Section 2.3.1. Due to an implementation error, though, SZZUnleashed does not only track the lines modified by the original fixing commit but also every line touched by the commits found on each layer of the search, including many lines not touched by the fix itself. With the parameter's default value set to 3, this results in a high number of commits supposedly unrelated to the fixing commit being blamed. Setting this parameter to 1 should cause it to behave similar to the original SZZ approach [12], but we are interested in observing the behavior of the default parameter, as well.

5 https://github.com/aserg-ufmg/RefDiff
6 https://github.com/jpace/diffj
7 https://github.com/wogscpar/SZZUnleashed

4 EXPERIMENTS

In this chapter, we describe the structure and setup of our experiments with our now complete bug provider implementation. Our main interest here is evaluating the precision and applicability of our implementation on Tool-Suite projects and comparing the results to other bug detection approaches.
First, it is important to get an idea about what kind of metrics we can use to compare different approaches in a meaningful way. Since it is difficult to find ground truth at least on what introducing commits are, we explore different possibilities for this in Section 4.1. After that, we take a look at the projects that we run our experiments on and briefly discuss the decision behind using these Tool-Suite projects in particular, which is what we do in Section 4.2. In the end, we discuss the comparison of different regular expressions for fix detection in Section 4.3 and the comparison between our bug detection approach and the approach of SZZUnleashed in Section 4.4.

4.1 EVALUATION METRICS

Evaluating how well bug detection algorithms actually perform is not a straight-forward task since for most projects, we are missing ground truth on what exactly the fixing and introducing commits of each bug in the project are. While a *manual inspection* of the commit history can be done for fixing commits if we agree to only examine a certain portion of the commits (note that VaRA-TS projects can reach from a size of around 600 commits to about 23,000 commits total), introducing commits are nearly impossible to retrace by hand. However, we are able to make use of the metrics introduced by Costa et al. [2]. In this paper, the authors suggest three different metrics to estimate how well a bug detection algorithm performs without the need of ground truth.

4.1.1 *Earliest Bug Appearance*

Using *earliest bug appearance*, we measure how much SZZ disagrees with the estimated affected versions by the development team. This metric profits off the fact that when using JIRA as issue tracking system, bugs can get reported along with the versions the bug affects. The earliest version mentioned in this field would get compared to the earliest introducing commit of a fixing commit and the introducer found by SZZ would be marked as incorrect if the introducing commit happened after the earliest affected version given by the development team. This is infeasible to use in our case since while GitHub allows referencing commits in an issue, there is no dedicated affected version field that we could use to analyze this in a meaningful way.

4.1.2 *Future Impact of Changes*

The idea behind the *future impact of changes* metric is to analyze the bug-introducing commit data in regards to what fixes were introduced by each single introducing commit and how realistic it is that the change is actually a bug-introducing change. The core idea here opens up two possibilities to measure the future impact of a change: The *count of future bugs* and the *time span of future bugs*, where the term *future bugs* refers to the fixes that a fix-inducing commit induced later on.

First off, the count of future bugs metric bases on the assumption that introducing commits are unlikely to introduce many fixes. The findings of a manual verification of SZZ-generated data by Williams et al. [15] suggest that around 93% of introducing commits only introduce between 1 and 3 fixes. We, therefore, assume that a bug introducing change that caused more than three fixes is either an exceptionally problematic change or an error in the SZZ-generated data. When we analyze the count of future bugs for an entire project, we measure the percentage of non-suspicious introducing changes, i.e., the changes that are bug introducing but do not introduce more than 3 bugs.

The time span of future bugs refers to all time spans between the introducing change that we are analyzing and all the fixes that the change introduced. Here, the idea is that if bugs introduced to a project usually get fixed rather quickly, strong outliers regarding the time span between introduction and fix suggest that the introducing change possibly did not introduce a bug. In order to approximate how likely bug introducing changes are to be correctly identified by the SZZ algorithm regarding this date difference, we observe how far it deviates from the median of time spans across the project. As previous work by Leys et al. [6] could show, computing the *median absolute deviation (MAD)* around the median usually yields better results than computing the standard deviation around the mean. Hence, Costa et al. [2] suggest that an introducing change should be considered realistic regarding the time span of future bugs if the time span between the introduction and all of its fixes does not exceed the upper MAD, which is the sum of the median on all of the data and its MAD. When applying this metric to the whole project, we are interested in the percentage of bug introducing changes that do not exceed this threshold.

Since it is compatible with bugs found by both commit messages and GitHub issue events, we are using this metric to measure what fraction of all fix-inducing changes found we can consider to be realistic.

4.1.3 *Realism of Bug Introduction*

The *realism of bug introduction* can be used to analyze the realism of the introducing changes that were found for a certain fixing commit, i.e., all introducing changes of the same bug. We assume that for each bug, the sequence of commits that introduced this same bug is not likely to have large time spans in-between. Thus, we analyze for each fixing commit whether its set of introducing commits is likely to belong to the same fix or not. In order to measure this, we consider the median of date differences among the introducing commits of each fix and once again use the upper MAD to find outliers. In particular, we assume that the sequence of introducing commits for a certain fix is most likely an error in the SZZ-generated data if the date difference among the introducing commits exceeds the upper

MAD.
Just like the future impact of changes, this metric is independent of issue tracking data (especially independent of JIRA issue tracking data) and can therefore also be used for our cause.

4.1.4 *Criteria for the Manual Inspection*

In order to evaluate the detection of fixing commits, it is feasible for us to perform a manual inspection on at least a portion of each project that we analyze. Using this inspection as ground truth, we can find the number of true positive, true negative, false positive and false negative fixing commits that our SZZ implementation generates.
In order to establish a few core rules to follow for the inspection, we first agree on only considering commits as fixes if they change the project's behavior and mention a problem in their commit message (or issue description in case the bug stems from the GitHub issue interface). Moreover, only if an issue is closed during the time window due to a linked commit that happened during the time window, we can associate a fixing commit with that issue. Meta changes such as merge commits also are not considered to be fixing commits in our inspection.
From now on, we refer to the fixing commits found through this manual inspection as *true fixing commits* and compare these true fixes to the fixes that were detected by our bug detection approaches and were committed during the time window of our manual inspection.

4.2 PROJECTS USED

As we do not want to generalize the acquired data over all projects of the Tool-Suite, we are rather looking to analyze three projects more in-depth in regards to their different properties. In this section, we discuss the three projects that we settled for in regards to their size, how they differ from each other and especially what portion of the commit history we were able to manually inspect for bug fixes. A small overview of the projects used can be seen in Table 4.1.

	Begin of Inspection	Inspected Commits	True Fixes
Gravity	01/01/2018	325	65
OpenVPN	01/01/2020	459	85
Gzip	01/01/2013	181	52

Table 4.1: Statistics of Inspected Projects

GRAVITY The first project that we are using for our analysis is GRAVITY[1], which is a programming language using a parser written in C. This repository currently includes around 740 commits. After performing a manual inspection on the commit history starting from January 1st, 2018, we find that around 65 commits of the 320 total commits that we

1 https://github.com/marcobambini/gravity

performed the inspection on are fixing commits, considering both the commit messages and issues on GitHub. Gravity uses GitHub as issue tracking system, which makes it interesting to consider for the analysis as we implemented a fix detection approach via GitHub issue events.

OPENVPN For the second project, we take a deeper look at OpenVPN[2], which is a large-scale open source tunneling daemon with considerably rapid development. In total, the project currently counts around 2900 commits, so we are only able to manually inspect a small portion of the total project regarding fixing commits. However, even though the project does not use the GitHub interface for issue tracking, the descriptive multi-line commit messages offered in this project allow deep insights into the purpose of each commit. Nevertheless, we can only inspect bugs found by commit messages for this project. A manual inspection lets us find 85 fixing commits among the 459 commits that have been authored since January 1st, 2020.

GZIP The last project we want to analyze is the compression utility GNU Gzip[3], which is a smaller-scale project with around 600 commits in total at the time of our analysis. This project is rather interesting to analyze since the average date difference between each commit largely deviates from the other two projects, so we expect potentially strongly different results regarding the date-related metrics we introduced compared to the other projects. The repository features only about 181 commits since January 1st, 2013, the beginning of our analysis. Over this time span, we find around 52 fixing commits, most of which are reported on its DEBBUG bug tracker[4] instead of issue tracking on GitHub, but consistent linking in commit messages to the corresponding bug pages makes double-checking our manual inspection less error-prone.

4.3 COMPARING KEYWORD MATCHING STRATEGIES

In the first part of our evaluation, we test our own keyword matching strategy against other fix detection approaches on the three previously selected Tool-Suite projects.

RESEARCH QUESTIONS First off, we evaluate how the regular expression used in the original SZZ, `fix(e[ds])?|bugs?|defects?|patch`, compares to our bug provider fix detection which assumes that the regular expression `fix(e[ds]?)` suffices to detect most fixes. Our manual inspection also leads us to believe that the original convention to only match keywords in the first line of the commit message might lead us to miss some essential bug fixes since it is not necessarily a common convention to always put the fact that a commit fixed an issue in the first line. The commit message might also just summarize the exact change that has been made in the first line and elaborate on what the issue was and which bug this change fixes in the further description. Based on this observation, we also decide to analyze the two keyword matching strategies mentioned before in relation to the

2 https://github.com/OpenVPN/openvpn
3 https://github.com/vulder/gzip
4 https://debbugs.gnu.org/

bug provider strategy when applied to the entire commit message.

RQ 1.1: *Can our approach that reduced the original SZZ regular expression to only the* `fix`*-stemming can also reduce our false positive rate? Also, does the restriction to only keyword match in the first line of commit messages let us miss out on true fixes, or does scanning the whole message increase our false positive rate too much?*

RQ 1.2: *Do different fix detection strategies also influence the estimated realism of the associated introducing commit sequences, i.e., does detected introducing commit data become more inaccurate along with its corresponding fix detection strategies when consulting the Costa et al. [2] metrics?*

EXPERIMENT SETUP As we are asking two separate questions here, this experiment divides itself into two parts.
For the first question, we implement the *fix evaluation tables*, which reports the absolute number of true positive, false positive, true negative and false negative detected fixing commits that happened during the same time frame that the manual inspection covers. Apart from these measurements, we also extract the number of total commits during the given time window, the total number of true fixes and the total number of fixes detected by our bug provider. We run this experiment three times, one time each for our own regular expression, the regular expression used in the original SZZ approach and our regular expression applied to the entire commit messages.
In order to answer the second question, we implement *introduction evaluation tables*. The exact values we are interested in here are the time span of future bugs and the realism of introduction each given as median of days along with the portion of commits that find themselves within the respective thresholds, as well as the portion of commits that manage to stay below the count of future bugs threshold of not more than 3 induced fixes. This experiment is also run once for each of the three fix detection strategies.

EXPECTED RESULTS Assuming that the actions mentioned in a commit message (e.g., "fixes", "adds", "improves", "implements", "cleans up") are most likely to describe the true purpose of a commit, keywords such as "patch" or "defect" might not show an equally strong correlation to the commit actually being a fix. Hence we run the first experiment in expectation for the original regular expression to match more commits that are no true fixes, but we also suspect that it might miss true fixes due to the first line constraint that the keyword matching strategy for the entire commit message should find.
As for the introduction evaluation, we expect more strict fix detection methods to also slightly lower the realism of introduction and time span of future bugs values, since we do not expect non-fixing changes to show any correlation in terms of time spans with the changes that last touched the same lines. Therefore, more wrongly classified fixing commits could cause some noise in the data.

4.4 BUG PROVIDER VS. SZZUNLEASHED

To investigate how the bug provider approach performs compared with other SZZ implementations, our second step is to run an experiment on the bug data laid out by each the provider and the SZZUnleashed research tool [1].

RESEARCH QUESTIONS As we already focussed on the first phase of SZZ in the first experiment and the main difference between our bug provider and SZZUnleashed is the approach to the second phase, we are mainly interested in comparing the introducing commit detection methods to each other consulting the metrics suggested by Costa et al. [2] While SZZUnleashed is coined to implement the improvements suggested by Williams et al. [15], the customizable depth parameter mentioned in Section 3.4 is not correctly implemented. We, therefore, consider both the bug data generated when using the default parameter of 3 and the data generated when using a depth of 1 for our analysis.

RQ 2: *Does the depth-of-1 SZZUnleashed version let us find more accurate introducing commit data than the default parameter? Also, is our bug provider implementation able to compete with either of the two versions despite not being coined to incorporate the same improvements?*

EXPERIMENT SETUP For this experiment, we review the entire repositories in regards to both fixing and introducing commits as we are not restricted to manually inspected portions of the commit history here. However, this also means that this setup purely relies on the metrics that we can use without any ground truth.
We can reuse the introduction evaluation tables implemented for the first experiment here, running the experiment for the SZZUnleashed implementation with default depth parameter, SZZUnleashed with a depth of 1 and our bug provider each. Note that we are using the slightly modified version explained in Section 3.4 for all SZZUnleashed runs.

EXPECTED RESULTS As SZZUnleashed claims to apply stricter pruning to the introducing commit candidates found, we usually would expect it to be able to rule out more introducing commits than the bug provider approach and therefore perform better in regards to the count of future bugs, but also in regards to the time span of future bugs and realism of bug introduction assuming that suspicious introducing commits are more likely to be ruled out. However, due to the flawed implementation of the depth parameter, the depth-of-1 version should behave similarly to the bug provider implementation.
Nevertheless, we expect three layers of introducing commits to contain too many candidates that are unrelated to the original fix and as such, we anticipate both SZZUnleashed with a depth of 1 and the bug provider to outperform the default parameter version.

5 EVALUATION

In the previous chapter, we elaborated on our experimental setup to evaluate our bug provider implementation. This chapter is dedicated to presenting our results in Section 5.1 and discussing them afterward in Section 5.2. Additionally, we discuss the potential problems of our experiments and how they could impair the validity of our work in Section 5.3.

5.1 RESULTS

Before being able to answer the research questions we formulated in the previous chapter, we present the exact values of the SZZ-generated data we acquired. We first present the keyword matching results relating to RQ 1.1 and RQ 1.2, and then the results of the comparison between SZZUnleashed and our bug provider relating to RQ 2 for each of the three projects, respectively. Additionally, we want to keep in mind here that when evaluating the introduction metrics, we prefer lower median values for each the time span of future bugs and the realism of bug introduction, while we prefer a higher number of commits falling under the upper MAD threshold. For the count of future bugs, we prefer a higher number of bugs falling below the threshold of 3.

5.1.1 *Keyword Matching Strategies*

First, we run both our fix evaluation and our introduction evaluation algorithm for all three fix detection approaches, i.e., the bug provider regular expression, the original SZZ regular expression and the bug provider regular expression applied to the whole commit message, on each Gravity, OpenVPN and Gzip. In the following, we present our results for each project separately.

GRAVITY We ran these experiments on all commits that have been committed from January 1st, 2018 up until June 11th, 2021.
Upon taking a look at the fixing commit evaluation presented in Table 5.1, what instantly catches our attention here is that the original SZZ regular expression gives us exactly the same number of correct predictions as our bugprovider regular expression. In addition to that, we see in the introducing commit evaluation shown in Table 5.2 that the introduction evaluation also produces the exact same results on both approaches, indicating that they both ended up finding the exact same fixing commits and consequentially the same introducing commits.
While applying our approach to the entire commit message slightly reduced the number of false negatives for the fixing commits from 22 to 17, it increased our false positive count significantly from 7 to 19. As for the introduction evaluation, the median realism of introduction, as well as the median time span of future bugs, slightly rose compared to the other approaches. A slight decrease in commits that stay below the count of future bugs

threshold is also noticeable. In addition to that, a very slight rise in the commit portion under the upper MAD can be discovered as well, but this is most likely due to the upper MAD threshold itself rising.

	Fixes Found	True Positive	False Positive	True Negative	False Negative
Bug Provider	50	43 (66.15%)	7 (2.69%)	253 (97.31%)	22 (33.85%)
Original SZZ	50	43 (66.15%)	7 (2.69%)	253 (97.31%)	22 (33.85%)
Bug Provider (entire msg)	67	48 (74.85%)	19 (7.31%)	241 (92.69%)	17 (25.15%)

Commits total: 325, True Fixes: 65

Table 5.1: Fix Evaluation on Gravity (2018-2021)

	Realism of Introduction		Time Span of Future Bugs		Count of Future Bugs
	Median (Days)	Below Upper MAD (%)	Median (Days)	Below Upper MAD (%)	Below Threshold (%)
Bug Provider	167.0	64%	453.5	73.85%	96,92%
Original SZZ	167.0	64%	453.5	73.85%	96,92%
Bug Provider (entire msg)	181	64.17%	489	76.4%	92.13%

Table 5.2: Introduction Evaluation on Gravity (2018-2021)

OPENVPN These experiments have been run on all commits to OpenVPN between January 1st, 2020 and June 11th, 2021.
Similar to the results of both our bug provider and SZZ original fix detections on Gravity, the results for both also are exactly identical for this time frame of the OpenVPN commit history, as we can see in the fix evaluation in Table 5.3 and the introduction evaluation in Table 5.4.
The previous observation regarding the bug provider regular expression applied to the entire commit message appears in an even more extreme manner here. Even though we can see the false negative rate slightly dropping, the false positive rate strongly rises in return and way more fixes are found than there are actual true fixes. Interestingly enough, the way more inaccurate fix detection lowers the median realism of introduction and also makes the portion of commits below the MAD threshold drop significantly but this can at least partially be explained by the MAD threshold dropping. Even so, the median time span of future bugs experiences a sharp rise compared to the other methods by 440 days, which is a difference of way over a year. This large rise of the MAD most probably explains the portion of commits below the threshold slightly rising. An insignificantly small drop of 2.06% for the introducing commits below the count of future bugs threshold also can be observed.

GZIP Note that we ran this experiment over around 8 years worth of commit history, from January 1st, 2013 to June 11th, 2021.
Contrary to Gravity and OpenVPN, there is a slight variation in the data between the regular expression of our bug provider and the original SZZ regular expression in the fix evaluation illustrated in Table 5.5. While the original SZZ manages to detect a bug that

	Fixes Found	True Positive	False Positive	True Negative	False Negative
Bug Provider	82	67 (78.82%)	15 (4.01%)	359 (95.99%)	18 (21.18%)
Original SZZ	82	67 (78.82%)	15 (4.01%)	359 (95.99%)	18 (21.18%)
Bug Provider (entire msg)	141	77 (90.59%)	64 (17.11%)	310 (82.89%)	8 (9.41%)

Commits total: 459, True Fixes: 85

Table 5.3: Fix Evaluation on OpenVPN (2020-2021)

	Realism of Introduction		Time Span of Future Bugs		Count of Future Bugs
	Median (Days)	Below Upper MAD (%)	Median (Days)	Below Upper MAD (%)	Below Threshold (%)
Bug Provider	968	85.37%	770	64%	97%
Original SZZ	968	85.37%	770	64%	97%
Bug Provider (entire msg)	863	70.21%	1210	73%	94.94%

Table 5.4: Introduction Evaluation on OpenVPN (2020-2021)

	Fixes Found	True Positive	False Positive	True Negative	False Negative
Bug Provider	24	21 (40.38%)	3 (2.33%)	126 (97.67%)	31 (59.62%)
Original SZZ	26	22 (46.43%)	4 (3.1%)	125 (96.9%)	30 (53.57%)
Bug Provider (entire msg)	33	26 (50%)	7 (5.43%)	122 (94.57%)	26 (50%)

Commits total: 181, True Fixes: 52

Table 5.5: Fix Evaluation on Gzip (2013-2021)

	Realism of Introduction		Time Span of Future Bugs		Count of Future Bugs
	Median (Days)	Below Upper MAD (%)	Median (Days)	Below Upper MAD (%)	Below Threshold (%)
Bug Provider	283	70.83%	1117	64.29%	100%
Original SZZ	283	65.38%	1163	67.74%	96.77%
Bug Provider (entire msg)	1416	72.73%	1886	66.07%	98.21%

Table 5.6: Introduction Evaluation on Gzip (2013-2021)

our approach did not find, it also classifies one more commit as a fix that is not truly a fix. In the introduction evaluation displayed in Table 5.6, we see that for our bug provider implementation, every introducing commit stays below the threshold of 3 introduced bugs. Apart from that, we can see very slight deviations between the bug provider and the original SZZ approach regarding every metric but there is no clear trend for either of these two to deliver significantly better results.
While we experience a rather small increase of each true positive and false positive fixes for our regular expression being applied to whole commit messages, the introduction evaluation table shows a very huge increase of almost three years in the realism of introduction median, as well as a considerably large increase of almost two years in the time span of future bugs.

5.1.2 *Bug Provider vs. SZZUnleashed*

For this experiment, we run the introduction evaluation for each Gravity, OpenVPN and GNU Gzip on each our bug provider, the SZZUnleashed implementation with a depth parameter of 1 and the SZZUnleashed implementation with its default depth parameter of 3. This time, we run the experiment on the entire commit history of our projects to get the true median values of the entire project for our introducing commit metrics. Once again, we present our results for each project in a separate section.

GRAVITY When comparing the SZZUnleashed version with a depth of 1 to our bug provider, we can see in Table 5.7 that while our bug provider implementation achieves better results regarding the realism of introduction and the count of future bugs, SZZUnleashed outperforms it regarding the time span of future bugs.
The most striking difference however is the difference between these two SZZ versions and SZZUnleashed with its default depth parameter, since it yields a way higher value for the realism of introduction and time span of future bugs median each, as well as an expectedly very low portion of introducing commits that manages to stay within the count of future bugs threshold.

	Realism of Introduction		Time Span of Future Bugs		Count of Future Bugs
	Median (Days)	Below Upper MAD (%)	Median (Days)	Below Upper MAD (%)	Below Threshold (%)
Bug Provider	29.0	82.26%	83.0	42.57%	95.05%
SZZUnleashed (detph=1)	45.0	71.96%	73.0	53.28%	91.8%
SZZUnleashed (depth=3)	126.0	66.36%	610.5	57.08%	53.21%

Table 5.7: SZZ Comparison on Gravity

OPENVPN As we can see in Table 5.8, SZZUnleashed with a depth of 1 slightly outperforms the bug provider in terms of the time span of future bugs, but the difference in terms of the count of future bugs is barely recognizable and we can not really see any of the two algorithms doing significantly better in terms of the realism of introduction, since while the median of days is lower for SZZUnleashed, more commits stay within the given threshold

for the bug provider.
Once again, we can see an extremely large dispersion between these variants and the SZZUnleashed version with a depth of 3. The fraction of introducing changes that still falls below the count of future bugs threshold drops significantly below half of the total introducing commits, the realism of introduction median is four times as large and the time span of future bugs is even five times as large as the values gathered for the bug provider and the depth-of-1 version.

	Realism of Introduction		Time Span of Future Bugs		Count of Future Bugs
	Median (Days)	Below Upper MAD (%)	Median (Days)	Below Upper MAD (%)	Below Threshold (%)
Bug Provider	386	81.52%	583	69.77%	94.08%
SZZUnleashed (detph=1)	365	76.4%	447	70.12%	93.7%
SZZUnleashed (depth=3)	1237	70.8%	2012.5	69.94%	44.94%

Table 5.8: SZZ Comparison on OpenVPN

gzip As outlined in Table 5.9, the realism of introduction median has a significantly lower value for our bug provider, while both SZZUnleashed versions obtain the same median. As for the portion of commits below the upper MAD, the bug provider still outperforms the depth-of-1 SZZUnleashed version slightly despite having a way lower median, and it clearly outperforms the depth-of-3 variant by a large amount.
The time span of future bugs is also the lowest for the bug provider implementation and while the depth-of-1 SZZUnleashed version obtains a median that is around 200 days larger, the depth-of-3 version even surpasses it by about 1000 days. The portion below the upper MAD is highest for SZZUnleashed, most likely due to the median being that high.
For the count of future bugs, all three variants do not stray very far away from each other, but the bug provider scores the largest portion below the threshold of three.

	Realism of Introduction		Time Span of Future Bugs		Count of Future Bugs
	Median (Days)	Below Upper MAD (%)	Median (Days)	Below Upper MAD (%)	Below Threshold (%)
Bug Provider	361.5	70.97%	1274.5	60.61%	95.45%
SZZUnleashed (detph=1)	1017.0	67.24%	1470.5	64.2%	92.59%
SZZUnleashed (depth=3)	1017.0	39.65%	2204.0	73.91%	92.75%

Table 5.9: SZZ Comparison on Gzip

5.2 discussion

In this section, we discuss the results we gathered in regards to the research questions that we proposed in the last chapter.

5.2.1 *Keyword Matching Strategies*

Concerning the fix evaluation, a large portion of the false positives and false negatives can be explained by observations made during the manual inspection, while the interpretation of the introduction evaluation is a little more complex. Before trying to answer the two research questions behind this experiment, we prepone a few general remarks about the comparison of true fixes against our fix detection approaches.

FIX DETECTION VS. GROUND TRUTH In general, false positives in comparison to the ground truth can mostly be explained by the keyword "fix" and other morphologically deducible forms of it being used in different contexts than we would expect them to be used. For example, "fix" is often used in documentation contexts, while we do not want to consider changes as fixes if they do not influence behavior. Even though there are improvements to SZZ producing refactoring-aware diff outputs which can ignore blamed commits that do not alter behavior, we do not apply the same strategy to incorrectly classified fixes in the first phase so far. Apart from this, multi-line commit messages tend to mention the state of the repository before the change and why this commit has been made, so our keywords often would get mentioned in different contexts, e.g., "added documentation to the fix made by version x". Since the commit history of OpenVPN yields especially long commit messages, this is most likely the reason behind the high count of false positives when mapping our regular expression to entire commit messages.
The false negative rate partly can be cleared up by the fix not being mentioned in the first line of the commit message, as we could see judging by the false negative rate dropping when we mapped our regular expression to entire commit messages. However, a certain portion of false negatives still remains for Gravity and Gzip in particular and the possible explanations depend on the projects, respectively.
On Gravity, for instance, there are many GitHub issues that describe bugs and were fixed by some associated commit, but these commits are mostly just linked in the comments on the issue and did not close the issues themselves. In addition to that, issues on Gravity do barely use the labeling system and as such, even if an issue was closed by the fixing commit directly, we could not be entirely certain whether the issue reported an actual bug, missing documentation or even a feature request.
On Gzip, the project with the highest false negative rate out of the three, we observe that fixes often just contain a description of the fix without labeling itself as fix and link the corresponding report of the resolved bug from either the mailing list or the DEBBUG bug tracker[1]. Since we do not account for this documentation style with our keyword matching approach, we miss many true fixes here.
As we can see looking at the different results for our three chosen projects, the best method to detect exactly the true fixes of a repository very strongly depends on the commit message and issue labeling conventions of the respective development team and is very hard, if not impossible to generalize.

1 https://debbugs.gnu.org/

RQ 1.1: For our first question, we wanted to know whether our `fix`-stemming approach in the bug provider could reduce the false positive rate of the original SZZ's fix detection approach by matching fewer keywords, but at least for the portions of the projects we considered and manually inspected, this was not the case. There is no sign of any of both approaches performing better in our results.
To address the second part of this question, the false positive rate when applying our regular expression to entire commit messages, especially to very long commit messages, appeared to increase our false positive rate considerably more than it reduced our false negative rate. Therefore, our data is consistent with the assumption made by Sliwerski et al. [12] that the first line of a commit message is most descriptive in terms of its true purpose.

RQ 1.2: This question is tough to answer judging by our results. Although there is a slight tendency for the more inaccurate keyword matching approach, i.e., the approach that matches entire commit messages, to score higher (and hence more unrealistic) median values than the other approaches on the introduction metrics, this is not really consistent across all three projects. For example, the entire message approach even scores the lowest median on OpenVPN for the realism of introduction. Thus, a slight correlation can be observed, but we cannot really attach a bigger meaning to it using the little data set that we gathered.

5.2.2 *Bug Provider vs. SZZUnleashed*

When discussing the differences between our bug provider and SZZUnleashed on the chosen projects, we do not have a ground truth that helps us explain the gathered data, but there are some important observations to make here regarding our final research question.

RQ 2: This question essentially boils down to two smaller questions, the first one being whether SZZUnleashed with a depth parameter of 1 provides us with more realistic data when consulting the metrics by Costa et al. [2] compared to our bug provider, since it is coined to prune more introducing commit candidates. While SZZUnleashed tends to perform better in terms of time span of future changes, our bug provider tends to perform best regarding the count of future bugs and the realism of introduction, so our results neither really enable us to prove nor to disprove this hypothesis. In the end, both algorithms produce bug data that we estimate as similarly realistic using our introduction metrics.
Nonetheless, the results support our second claim that both versions outperform the depth-of-3 SZZUnleashed variant. Since it adds two additional layers of introducing commits to the introducing commit set of the original fix, the count of future bugs threshold classifies even more than half of all fix-inducing commits as unrealistic on OpenVPN, which largely deviates from the depth-of-1 approach and our bug provider never falling below 91% on any project. The time-span of several years in-between introducing changes of the same fix or an introducing change and its induced fixes also reflects the intuition that the depth-of-3 variant most likely labels many changes as bug introducing that never touched the same code regions as the fix, to begin with.

5.3 THREATS TO VALIDITY

Although our results at least partially reflect our intuition, there are some other possible explanations for our findings.
Starting off, our manual inspection may be inaccurate. While we can assume with confidence that changes labeled as true fixes by us were seen as fixes by the developers, we have no way of knowing whether commits that modified behavior but were not implied to be fixes in the documentation actually did not fix any errors. Since this can only affect false negatives, our results would otherwise still be accurate.
Furthermore, while the metrics proposed by Costa et al. [2] were verified to be valid metrics for realism of SZZ-generated data, realism does not necessarily imply whether the data is incorrect. Sometimes, the anomalies are caused by strong outliers, e.g., in case of the impact of future changes, the last bug that a single commit introduced could be dormant for several years and let the commit look like an unrealistic introducing change. While we are unable to fully replace ground truth using these metrics, it has been shown that the introductions labeled as unrealistic are more likely to be an error in the SZZ data [2].
Lastly, the number of commits and projects that we analyzed is rather low, threatening the statistical significance. Our aim was to analyze our bug provider implementation in comparison with other fixing and introducing commit detection approaches on only a few projects such that we can provide more in-depth insights. Yet if we wanted to give our claims more statistical support, we would need to run our experiments on more projects and gather more data.

6 CONCLUDING REMARKS

We conclude our thesis by giving a short recap of what we achieved in the previous chapters and discussing some ideas on possibilities to further improve our work in the future.

6.1 SUMMARY

In this thesis, we implemented a convenient method for the VaRA-Tool-Suite to access bug-related revisions in order to help support future experiments with additional context. Our SZZ approach pledged to be accessible for all projects included in the Tool-Suite. To achieve this, we made it fully independent of the programming language and issue tracking system used, it just needs the project to use Git as version control system. In our implementation, we incorporated two fix detection approaches: Fix detection via commit messages and an optional detection method via GitHub issue events on projects that use the native GitHub issue tracking system. We based the regular expression used for our fix detection on the assumption that we do not need to match every keyword contained in the original SZZ regular expression and consequentially only incorporated deducible forms of the "fix" stem. The introducing commit detection that we realized with the help of PyDriller adapts to the fix detection approach used to find the respective fix. It is able to rule out hard suspects when given issue data corresponding to the fixed bug using the SZZ method. Furthermore, we incorporated the open-source SZZ implementation SZZUnleashed into VaRA-TS as a research tool and made some adjustments that fixed issues the original version had.
In the end, we ran experiments in order to test our implementation against other fix detection algorithms and compared our implementation to SZZUnleashed in regards to the realism of the data gathered. We manually inspected certain time windows of the three Tool-Suite projects Gravity, GNU Gzip, and OpenVPN for fixing commits with each of these approaches and found that while the original and our regular expressions produced very similar results, we scored way better than SZZUnleashed on default settings when consulting introduction metrics that can be computed without ground truth. The SZZUnleashed implementation performed very similar to our bug provider, however, when we told it to not mine for a larger introducing commit depth than one. In the end, our experiments showed that we do not need to match for every keyword used in the keyword matching strategy used by the original SZZ approach [12], but its claim that the purpose of a commit can mostly be found in the first line of its commit message is backed up by our results.
In conclusion, we managed to achieve our goal to implement an easily reusable bug detection method for VaRA-TS that can compare to other open-source approaches out there but there is still much room for improvement left. For example, there are many refactoring changes that we still are not able to detect, and fix detection largely depends on different commit message conventions. In the following, we propose a few ideas to address these issues.

6.2 FUTURE RESEARCH

During our experiments, we realized some shortcomings of especially our issue event fix detection algorithm and how we could perform better if we were able to incorporate some established SZZ improvements into VaRA-TS without losing the advantage of language independency.

6.2.1 *Enhancing Fix Detection Based On Issue Events*

There is potential to improve the pruning of introducing commits (since we cannot prune introducing commits for commit message bugs) by supporting more issue tracking systems in our bug provider implementation and make the suspect classification we implemented applicable to as many issue formats as possible. Furthermore, keyword matching the title and description of issues could help us detecting more issue related fixes since only very few projects using the GitHub issue interface make use of the labeling feature. In order to counteract the fact that we can not link closed issues to fixing commits if commits are not directly closing the issue they fixed, we also could classify changes that happened during the same time window as the closure of an issue as fixing commits and analyze how often that assumption turns out to be correct.

6.2.2 *Implementing a Simple Documentation Change Filter*

Refactoring changes so far have been infeasible to detect in the Tool-Suite environment due to the language limitations. Some types of refactoring changes, such as changes made to comments, are not especially challenging to detect, though, since there are very similar and simple syntactic patterns to comments in every language. Apart from that, changes made to file types that are no code files, such as `.txt`-files, also could be feasible to rule out. Pruning introducing candidates that only changed documentation bears the potential to improve our introducing commit detection. Moreover, to our knowledge, no bug detection research so far has ruled out fixing commit candidates that do not change behavior during the first phase of SZZ and this filter could also help us to include this into our bug detection approach.

6.2.3 *Finding Introducers for Insertion-Only Fixes*

As we discussed in Section 2.3.3, Remeli [10] proposed also blaming surrounding lines of addition-only fixes in order to be able to find introducing commits for fixes without deletions, but since our bug detection approach uses PyDriller to blame the diff of fixing commits, we were not able to customize which lines are being blamed, though. By using a more low-level API that allows us to obtain the diff and blame certain lines of a commit manually, we could also incorporate this improvement into our bug provider.

BIBLIOGRAPHY

[1] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel 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: (Mar. 2019).

[2] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. "A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes." In: *IEEE Transactions on Software Engineering* 43.7 (2017).

[3] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. Jr. "Automatic Identification of Bug-Introducing Changes." In: (Sept. 2006).

[4] H. W. Kuhn. "The Hungarian method for the assignment problem." In: *Naval Research Logistics Quarterly* 2.1-2 (1955), pp. 83–97.

[5] V. Levenshtein. "Binary codes capable of correcting deletions, insertions, and reversals." In: *Soviet physics. Doklady* 10 (1965), pp. 707–710.

[6] Christophe Leys, Christophe Ley, Olivier Klein, Philippe Bernard, and Laurent Licata. "Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median." In: *Journal of Experimental Social Psychology* 49.4 (2013).

[7] Edmilson Neto, Daniel Costa, and Uirá Kulesza. "The Impact of Refactoring Changes on the SZZ Algorithm: An Empirical Study." In: Mar. 2018.

[8] Edmilson Neto, Daniel Costa, and Uirá Kulesza. "Revisiting and Improving SZZ Implementations." In: (Sept. 2019).

[9] *Refactoring: Improving the Design of Existing Code*. USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] Mina Remeli. "Enhanced-SZZ: an improved code change labeling algorithm." MA thesis. 2019.

[11] Danilo Silva and Marco Tulio Valente. "RefDiff: Detecting Refactorings in Version Histories." In: *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press, 2017.

[12] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. "When do changes induce fixes?" In: *ACM Sigsoft Software Engineering Notes* 30 (July 2005).

[13] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. "RefactoringMiner 2.0." In: *IEEE Transactions on Software Engineering* PP (July 2020), pp. 1–1.

[14] Chadd Williams and Jaime Spacco. "Branching and merging in the repository." In: (Jan. 2008), pp. 19–22.

[15] Chadd Williams and Jaime Spacco. "SZZ revisited: verifying when changes induce fixes." In: (Jan. 2008).