Tracking the Evolution of Static Code Warnings: the State-of-the-Art and a Better Approach

Junjie Li

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of

Master of Computer Science (Computer Science) at

Concordia University

Montréal, Québec, Canada

May 2021

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Junjie Li**

Entitled: **Tracking the Evolution of Static Code Warnings: the State-of-the-Art and a Better Approach**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the Final Examining Committee:

_____ Chair
Dr. Juergen Rilling

_____ External Examiner
Dr. Weiyi Shang

_____ Examiner
Dr. Juergen Rilling

_____ Supervisor
Dr. Jinqiu Yang

Approved by _____
Dr. Lata Narayanan, Chair
Department of Computer Science and Software Engineering

13 May 2021 _____
Dr. Mourad Debbabi, Dean
Faculty of Engineering and Computer Science

Abstract

Tracking the Evolution of Static Code Warnings: the State-of-the-Art and a Better Approach

Junjie Li

Static bug detection tools help developers detect problems in the code, including bad programming practices and potential defects. However, it is known that static bug detectors remain underutilized due to various reasons. Recent advances to incorporate static bug detectors in modern software development workflows, such as in code review and continuous integration, are shown capable of better motivating developers to fix the reported warnings on the fly. A proper mechanism to track the evolution of the reported warnings can better support such integration. Moreover, tracking the static code warnings will benefit many downstream software engineering tasks, such as learning the fix patterns for automated program repair and learning which warnings are of more interest, so they can be prioritized automatically. Hence, precisely tracking the warnings by static bug detectors is critical to improve the utilization of static bug detectors further.

In this thesis, we study the effectiveness of the state-of-the-art (SOA) solution in tracking the warnings by static bug detectors and propose a better solution based on our analysis of the insufficiencies of the SOA solution. In particular, we examined over 2000 commits in four large-scale open-source systems (i.e., JClouds, Kafka, Spring-boot, and Guava) and crafted a dataset of 3,452 static code warnings by two static bug detectors (i.e., Spotbugs and PMD). We manually uncover the ground-truth evolution status of the static warnings: persistent, resolved, or newly-introduced. Moreover, upon manual analysis, we identified the main reasons behind the insufficiencies of the SOA solution. Finally, we propose a better approach to improve the tracking of static warnings over software development history. Our evaluation shows that our proposed approach provides a significant improvement in terms of the precision of the tracking, i.e., from 66.9% to 90.0%.

Acknowledgments

Foremost, I would like to express my greatest gratitude to my supervisor Dr. Jinqiu Yang. Her guidance, knowledge, and mentorship helped me to overcome research obstacles, and more importantly, to grow to become a better person. Without her supervision and support, nothing of this would have been possible.

Apart from my supervisor, I would like to sincerely thank my thesis examiners, Dr. Shang and Dr. Rilling, for their extremely valuable and constructive suggestions. Furthermore, I appreciate Dr. Peter Chen for his valuable guidance in my research.

I am very lucky to have lively communications and fruitful discussions with all the members of SE group. I learned so much from all of you, it is my honor and pleasure to work with you all. My special thanks go to my friends, Bo Yang, Zehao Wang, and Triet Pham who were always there for me, in both fun and difficult times.

Nothing can express my gratitude to my parents, for their love and constant encouragement, and unconditional support to keep me going.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Static bug detection tools have been widely applied in practice to detect potential defects in software. To name a few, both Google and Facebook adopt static bug detectors in their large codebases on a daily basis (Sadowski, Aftandilian, Eagle, Miller-Cushon, & Jaspan, 2018). However, static bug detectors are known to be underutilized due to various reasons. First, static bug detectors detect an overwhelming number of warnings, which may be far beyond what resources are allowed to resolve. For example, Spotbugs (*Spotbugs latest version*, 2019), i.e., the spiritual successor of *Findbugs*, detects thousands of or more static code warnings in one version of *JClouds*. Second, static bug detectors are known to detect many false positive warnings. The existence of a large number of false positives discourages developers from actively working on resolving the reported warnings. As a result, a significant portion of static code warnings remain unresolved by developers and can hinder software quality.

There have been efforts from a variety of directions to improve the utilization of static bug detection tools, e.g., prioritizing and recommending actionable static warnings and identifying false positive warnings. For example, researchers have been working on techniques to identify the actionable warnings and reduce the false static code warnings, such as recommending actionable warnings by learning from past development history (Hanam, Tan, Holmes, & Lam, 2014; S. Kim & Ernst, 2007). On the other hand, recent studies show that by better integrating static bug detectors in software development workflows, such as code review and continuous integration, developers demonstrated a higher response rate in resolving the reported static warnings (Sadowski et al.,

2018; Sadowski, van Gogh, Jaspan, Soederberg, & Winter, 2015). Developers are presented with much fewer warnings, which are introduced by a new commit, and encounter fewer context switch problems in fixing the warnings.

Making static bug detectors more frequent in workflows such as code review requires proper management of the evolving static code warnings. Such proper management is not straightforward. One way is to adapt differential static analysis to only analyze modified code files, yet to achieve satisfactory performance. However, it requires algorithm innovation and non-trivial engineering effort for every static bug-finder. Alternatively, we advocate for management that tracks the evolution of static code warnings in the commit history, i.e., *diff* the static code warnings from two consecutive software revisions. Tracking the evolution of static code warnings reveals that given a commit, which warnings remain unresolved by developers, which warnings are resolved, and which warnings are newly-introduced in the commit.

More importantly, effective management of static code warnings will benefit many downstream software engineering tasks. To name a few, researchers have been crawling past fixes of static code warnings to provide fix suggestions for new warnings (Bavishi, Yoshida, & Prasad, 2019; Liu, Kim, Bissyande, Yoo, & Le Traon, 2018), which has been shown can further improve the utilization of static bug detectors. Furthermore, such concluded fix patterns are shown to be effective in automated program repair techniques (Liu, Koyuncu, Kim, & Bissyandé, 2019).

Till this end, there has been little effort to systematically review existing solutions to track the evolution of static code warnings and accordingly to propose better solutions. Prior studies rely on simple heuristics to track the static code warnings (Boogerd & Moonen, 2009; S. Kim & Ernst, 2007), i.e., two warnings are identical if they are of the same warning type, in the same file, etc. Avgustinov et al. (Avgustinov et al., 2015) present an algorithm that combines various types of information of one warning, compares two warnings in layers and eventually establishes mappings between two sets of warnings from two software revisions. This algorithm is adopted by recent automated program techniques, and in this thesis, we refer to it as the state-of-the-art (SOA) solution. For example, Liu et al. (Liu et al., 2018) adapt the SOA solution to identify warning-fixing commits in software repositories for automated program repair. However, it remains unknown how accurate the SOA solution is in tracking the static code warnings. An unacceptable performance of the SOA

solution in tracking static code warnings has subsequent negative impacts on the downstream tasks.

Hence, to foster future research in static code warnings, in this thesis, we examine the performance of the SOA solution in tracking static code warnings and propose a better approach after analyzing the insufficiencies of the SOA solution. We answer the following research questions:

RQ1 Is the SOA approach good at tracking the evolution of static code warnings?

RQ2 What are the limitations of the SOA approach?

RQ3 Can our proposed approach perform better than the SOA approach?

RQ4 How accurate is our proposed approach for tracking the evolution of static code warnings?

In particular, our study includes two static bug detectors (i.e., *P*MD and *S*potbugs) and four large-scale open-source software systems (i.e., *J*Clouds, *K*afka, *S*pring-boot, *G*uava) for manual analysis.

To answer **RQ1** and **RQ2**, we crafted a dataset of static code warning and their evolution. In particular, we took statistically significant samples of the reported static code warnings from the entire development history of *JClouds* and *Kafka*, and performed manual analysis to label whether each sampled static code warning is *persistent*, *resolved*, or *newly-introduced* between two consecutive revisions. Eventually we crafted a dataset of **1,715** static code warnings and their evolution status for both manual analysis and future evaluation.

After analyzing the limitations of the SOA solution (**RQ2**), we propose a better approach by addressing the uncovered limitations to answer **RQ3**. Our proposed approach leverages refactoring information and a better matching strategy: **Hungarian algorithm** (Kuhn, 1955), a classic algorithm to solve the assignment problem in bipartite graphs to reduce the impact from the order of the SOA approaches.

In **RQ4**, in addition to *J*Clouds and *K*afka, we also select two other open-source software systems (i.e., *S*pring-boot and *G*uava), which provides a systematic comparative evaluation between our approach and the SOA approach. There are 3,452 labeled warnings we collect from four software systems. Our evaluation based on this dataset shows that our approach provides a significant improvement over the SOA solution, i.e., from 66.9% to 90.0% in terms of the tracking precision. We also take an independent evaluation on the performance of our approach by taking a statistically significant sample of all commits from the four software systems with each static bug detector,

and determine the resolved and newly-introduced warnings. Finally, the tracking precision of our approach is calculated, and our approach achieves 91.8% accuracy.

In summary, this thesis makes the following contributions:

- We collected and manually labeled a dataset of 1,715 static code warnings and uncovered the ground-truth evolution status between two consecutive commits. The static code warnings are detected by two mature and widely used static bug detectors (*PMD* and *Spotbugs*) on two real-world open-source software projects (*JClouds* and *Kafka*).
- We examined the state-of-the-art solution in tracking the evolution of static code warnings in terms of tracking accuracy based on the collected dataset. Our investigation shows that the SOA solution achieves inadequate results.
- We performed a manual analysis to uncover the inaccuracies and the reasons behind the low accuracy of the SOA solution. Our findings offer empirical evidence to further improve the tracking of static code warnings in the development history. We also select two other top-rated projects (*Guava* and *Spring-boot*) with 1,737 static code warnings to evaluate between both approaches.
- We proposed a better approach to track the static code warnings. The evaluation based on the crafted dataset shows that our approach can significantly improve tracking precision. The dataset is available online (*The shared dataset*, 2020).

1.1 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 describes the background, i.e., the relevant knowledge on static code warnings and how the SOA approach works to track the static code warnings in development history. Chapter 3 illustrates the process and results of our manual analysis to understand the problems of the SOA solution, including how the dataset is crafted and what are the insufficiencies of the SOA solution. Chapter 4 shows our proposed approach and its evaluation. Chapter 5 describes the threats to validity. Chapter 6 lists the related work, and Chapter 7 concludes the thesis and proposes future works.

Chapter 2

Background

This chapter will introduce the background information on the basics of static code warnings and how the state-of-the-art (SOA) solution proposed by Avgustinov et al. works to track the evolution of static code warnings in software development history.

2.1 The Metadata of Static Code Warnings

Static bug detectors often represent detected static code warnings using metadata. Examples of such metadata include file path (i.e., the path of the source-code file where one warning is detected) and types of static code warnings. Previous studies (Avgustinov et al., 2015) (Liu et al., 2018) utilize the metadata information that static bug detectors provide for each warning to track the evolution of the detected static code warnings.

Different static bug detectors are different in many aspects, such as the warning types and source code representations (e.g., binary or source-code). Despite the differences, the metadata that are generated by static bug detectors include more or less similar information. Such metadata can distinguish one static code warning from another **in the same revision** (e.g., warning type and code location). When the metadata of the same static code warning across two revisions (i.e., before and after one commit) is modified due to the introduced code changes by the commit, tracking the evolution of this static code warning may become challenging.

Figure 2.1 provides an example of static code warning in *JClouds* that is detected by *Spotbugs*.

We show the example of metadata in XML format. The metadata of the static code warning includes the following detailed information: the type of the static code warning (i.e., *SE_BAD_FIELD*), and the problematic code region of this warning, which is represented by project name, class name, method name, field name, and the code range that is defined by a start line and an end line.

```
<WarningInstance>
  <WarningType>SE_BAD_FIELD</WarningType>
  <Project>jclouds</Project>
  <Class>ContextBuilderTest</Class>
  <Method></Method>
  <Field></Field>
  <FilePath>org/jclouds/ContextBuilder.java</FilePath>
  <StartLine>70</StartLine>
  <EndLine>75</EndLine>
</WarningInstance>
```

Figure 2.1: An example of metadata of one static code warning that is detected by SpotBug. Note that the metadata information has been simplified to only show the information used by the SOA tracking approach (Avgustinov et al., 2015).

2.2 Tracking Static Code Warnings

Static bug detectors, when being run in batch mode, report a list of static code warnings given one version of a software system (i.e., one snapshot). Tracking the evolution of static code warnings in development history is based on comparing the generated reports from every two consecutive revisions. Figure 2.2 shows a simple example of how the tracking works between two revisions. Given one commit, the left block represents all the warnings from the pre-commit revision and the right one is the post-commit revision. Between the two revisions, some static code warnings persist, i.e., `w1`, `w2`, `w3` and `w4`. Note that `w1` is in different shapes in the two revisions as the representation of `w1` may have different values due to code changes. One warning is resolved (i.e., `w5`) and one warning is newly-introduced (i.e., `w6`) by the commit.

A proper tracking mechanism needs to precisely label each static code warning as either *persistent*, *resolved* or *newly-introduced*. In particular, it is required that all the mappings are correctly established despite that code changes may modify the metadata information of the same warnings across versions. For example, the solid lines in the figure describe one possible matching outcome, which is not ideal as the `w2` warnings of the two revisions are not correctly mapped, and `w3` of the post-commit revision is incorrectly mapped with `w4` of the pre-commit revision. After the

mappings are established, a tracking solution will determine the label of each static code warning, i.e., both `w2` and `w4` are decided as *resolved* while both of them actually persist between the two revisions. Both are false positives of the tracking solution. Differently, `w5` and `w6` are true positives of the tracking solution as their labels are correct, i.e., `w5` is resolved and `w6` is newly-introduced. Interestingly, although `w4` is also incorrectly matched, the label of `w4` of the pre-commit revision, which is resolved, is indeed correct. However, the labels of `w3` of the pre-commit revision and `w4` of the post-commit revision are incorrect.

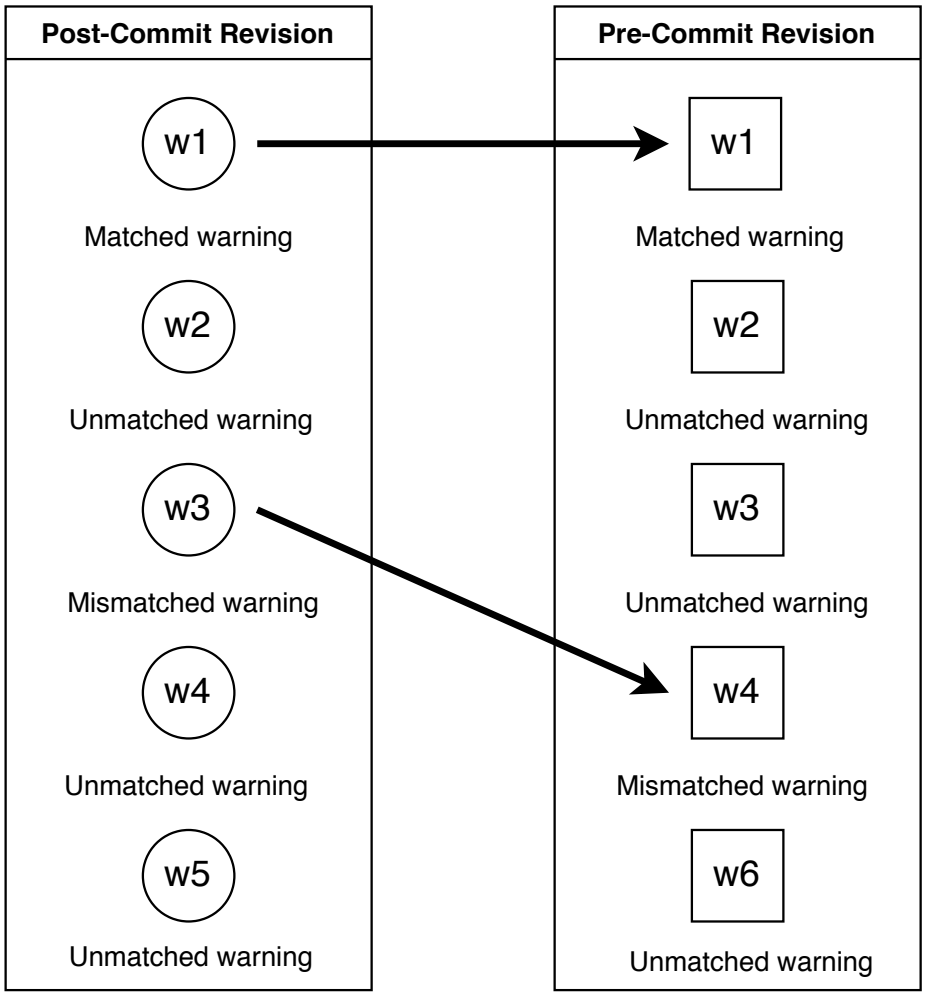


Figure 2.2: An example of matching the warnings between two consecutive revisions.

There exist various software maintenance efforts to make the tracking problem more complicated than one may imagine. For example, code changes that are irrelevant to efforts of resolving static code warnings, such as a drastic refactoring, may modify the metadata information of many

static code warnings so they cannot be mapped while they should. On the contrary, a warning that is resolved may share similar information (due to code changes) with one other irrelevant warning, which causes an incorrect mapping. In short, a fine line needs to be drawn to precisely distinguish the three evolution types.

Based on the status of the static code warnings (i.e., persistent, resolved, and newly-introduced), we define the true positive, false positive, true negative, and false negative of tracking static warnings in our study.

- **True positive (TP)**: A static warning is identified as a resolved/newly-introduced warning by a tracking approach correctly.
- **False positive (FP)**: A static warning is identified as a resolved/newly-introduced warning by a tracking approach incorrectly.
- **True negative (TN)**: A static warning is identified as a persistent warning (i.e., neither resolved nor newly-introduced) by a tracking approach correctly.
- **False negative (FN)**: A static warning is identified as a persistent warning (i.e., neither resolved nor newly-introduced) by a tracking approach incorrectly.

2.3 The State-of-the-Art (SOA) Solution

Avgustinov et al. (Avgustinov et al., 2015) proposed a multi-stage matching algorithm that can properly track the evolution of static code warnings under certain complicated software evolution, which we refer to as the state-of-the-art solution (SOA for short). The overall structure of the SOA solution is based on a pair-wise comparison between each warning in the pre-commit revision and each warning in the post-commit revision. Once a mapping is established, the two warnings from the two revisions are excluded for further comparisons. In particular, for each pair-wise matching process, i.e., between one warning from the pre-commit revision and one from the post-commit revision, four different matching strategies are placed in order, namely exact matching, location-based matching, snippet-based matching, and hash-based matching.

Algorithm 1 illustrates how the SOA solution works to establish mappings between the list of warnings of two consecutive revisions. Exact matching requires every piece of metadata information to be matched and therefore is the most strict matching strategy among the four. When exact matching fails, the SOA solution will then utilize the less strict matching strategy, i.e., location-based matching, which employes the diff algorithm to tolerate certain line shifts. If location-based matching fails, the SOA solution will continue to use snippet-based matching. When a class file was renamed or moved, the above matching strategies cannot handle that. Thus, the SOA solution will utilize hash-based matching.

At the end, when all the possible mappings are established, the unmatched warnings in the pre-commit revision are determined as resolved, and the ones in the post-commit revision are considered as newly-introduced.

2.3.1 Exact Matching

Exact matching establishes the mappings for the warnings that are totally unaffected by the commit. For the two warnings, it is required that they have exactly the same source location (i.e., defined by the start and end line of the warning), warning type, and code information (i.e., class name, method name, and variable name).

2.3.2 Location-based Matching

A commit may modify the information of certain static code warnings. Therefore when the exact matching fails, the following matching strategy, location-based matching, is used to tolerate the impacts to some extent. Location-based matching utilizes the *diff* algorithms (Hunt & Szymanski, 1977) (Myers, 1986) to derive source position mappings for each modified file. When a (potential) matching pair of warnings is in the diff output, location-based matching compares the offset of the corresponding warnings in the mappings. This matching requires the same warning metadata of code information (i.e., class name, method name, and variable name), but does not require the same source location(i.e., the start and end line of the warning). If the difference of offsets is equal to or lower than *3* (i.e., a fixed threshold), the location-based matching will decide the two warnings as a matching pair.

```
    @@ -84,3 +84,5 @@
84  84   public class MyClass{
    85 + // add code
    86 + //
85  87    private String str = null;
86  88   }
```

Figure 2.3: An example commit to show how location-based matching works.

As an example, Figure 2.3 shows a diff mapping. The numbers on the left hand are the line numbers in the pre-commit revision. The numbers on the right hand are the line numbers in the post-commit revision. There is a warning reported in the pre-commit revision (line 85) and line 87 in the post-commit revision. Due to code adding, the source location (i.e., part of the warning metadata) has been changed. Location-based matching firstly computes the offsets between the source location and the diff mappings, respectively. The offset between the first line of the diff mapping (line 84) and the warning (line 85) is 1 for the pre-commit revision and 3 (line 87 and line 84) for the post-commit revision. Then, the difference between the two offsets is calculated. In this example, the difference is smaller than 3, so that location-based matching will match the two warnings.

2.3.3 Snippet-based Matching

When code location changes significantly, the location-based matching approach may fail to identify persistent warnings across revisions. Snippet-based matching is used to address this problem. Given the source location defined by a start line and an end line, code snippets in between are extracted from both revisions. By performing the string matching on the two code snippets, snippet-based matching will decide a mapping if they are identical. Same as location-based matching, snippet-based matching requires the same warning metadata of code information (i.e., class name, method name, and variable name).

2.3.4 Hash-based Matching

It is possible that a file is moved to a new location or renamed (i.e., class and file path are modified). Snippet-based matching cannot handle such cases well since the class name are required

to be identical to perform snippet-based matching. For such cases, a hash-based matching approach can be helpful. This matching approach tries to match warnings based on the similarity of their surrounding code. It first splits the text of warning location into several tokens. Two hash values are calculated $h(W_{topN})$ and $h(W_{latter})$. W_{topN} is n tokens from the first one. W_{latter} is tokens from the $n + 1^{\text{th}}$ token to the last token. n is a fixed threshold. If two warnings(i.e., one is from pre-commit revision and another one is from post-commit revision) whose $h(W^{post}_{topN}) = h(W^{pre}_{topN})$ or $h(W^{post}_{latter}) = h(W^{pre}_{latter})$, they will be considered as a matched pair.

2.3.5 How One Incorrect Mapping May Impact All

Tracking the warnings across two consecutive revisions should be treated as one integral issue because each incorrect mapping not only impacts the two warnings involved but also impacts other warnings. For example, in the imperfect mappings in Figure 2.2, `w3` on the left is incorrectly mapped with `w4` on the right. This incorrect mapping not only impacts these two warnings, but also `w4` on the left and `w3` on the right. Now `w4` on the left is decided as resolved (incorrect), and `w3` on the right is considered as newly-introduced (also incorrect). Interestingly, not all the incorrect mappings must introduce false positives. Figure 2.4 shows an example in this category. On the left figure, we can see two incorrect mappings: `w1` to `w2` and `w2` to `w1`. Although both mappings are incorrect, there is no false positive generated since these four warnings form a closure that therefore does not impact the mappings of other warnings. On the contrary, Figure 2.4(b) is a worse case: two false positives (`w1` on the left and `w2` on the right) are generated). Our goal is to reduce the second type of mappings through our analysis.

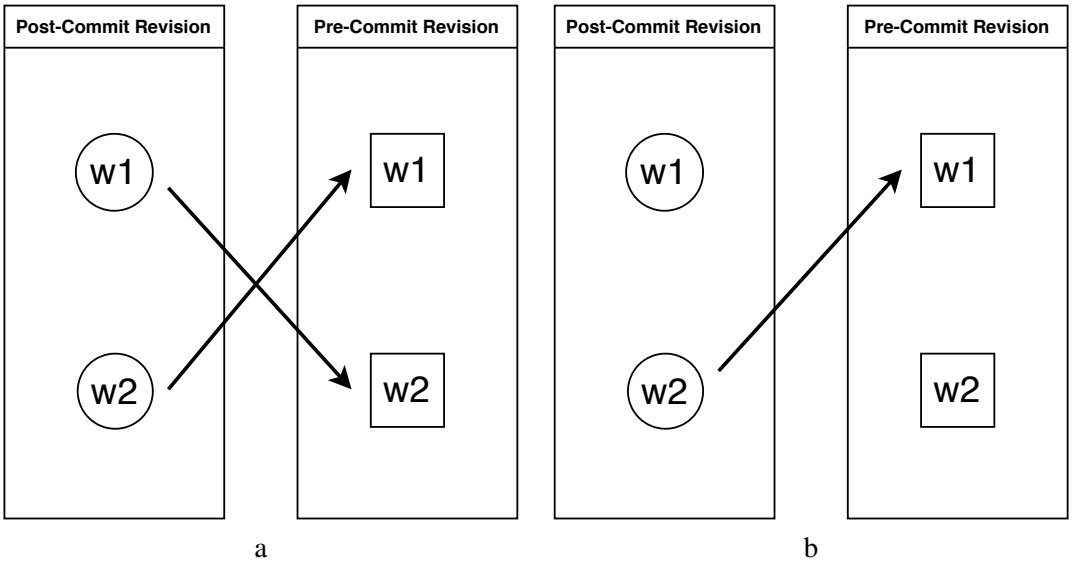


Figure 2.4: Examples to demonstrate how incorrect mappings may impact all.

Algorithm 1: The basic algorithm of the SOA approach.

Input: The set of warnings from the pre-commit revision, W_p; The set of warnings from the post-commit revision, W_c;

Output: The set of resolved warings, $W_{resolved}$; The set of newly-introduced warings, $W_{newly-introduced}$; The set of matched pairs, $MatchedPairs$;

```
1  for each w_i in W_p do
2      for each w_j in W_c do
3          if source file of w_i is not a changed file then
4              take ExactMatching(w_i, w_j);
5          else
6              take ExactMatching(w_i, w_j);
7              if w_i is not be matched up then
8                  take LocationMatching(w_i, w_j);
9              else
10                 make a MatchedPair(w_i, w_j);
11                 remove w_j from W_c ;
12                 break;
13             if w_i is not be matched up then
14                 take SnippetMatching(w_i, w_j);
15             else
16                 make a MatchedPair(w_i, w_j);
17                 remove w_j from W_c;
18                 break ;
19             if w_i is not be matched up then
20                 take HashMatching(w_i, w_j);
21             else
22                 make a MatchedPair(w_i, w_j);
23                 remove w_j from W_c;
24                 break;
25             if w_i is not be matched up then
26                 add w_i into W_resolved;
27 W_newly-introduced = W_c − MatchedPairs;
```

Chapter 3

Examining the performance of the State-of-the-Art Solution in Tracking the Evolution of Static Code Warnings

In this chapter, we describe how we investigated the performance of the SOA approach in terms of the tracking accuracy, and answer **RQ1** in Chapter 3.3 and **RQ2** in Chapter 3.4. In particular, we first crafted a dataset of static code warnings and their evolution status (i.e., persistent, resolved, or newly-introduced) between two consecutive revisions. To craft this dataset, we re-implemented the SOA approach, applied it to the development history of the studied open-source systems, and performed a manual analysis to determine the evolution status for each sampled static code warning. Then we manually analyzed whether the SOA approach correctly tracked each sampled static code warning and categorized the reasons behind any failures. Figure 3.1 shows the overview of our study.

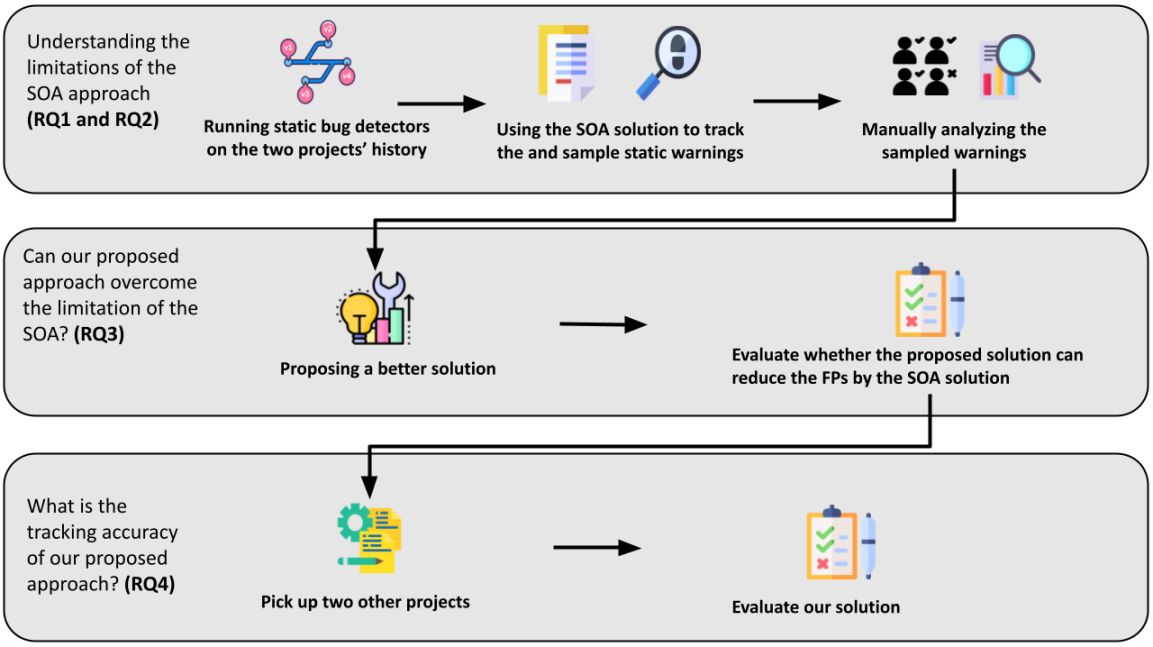


Figure 3.1: An overview of our study.

3.1 Studied Subjects

3.1.1 Static Bug Detectors

In this thesis, we include two static bug detectors, i.e., *PMD* and *Spotbugs*, both of which are widely used in prior studies and adopted in practice. In particular, Spotbugs, a spiritual successor of the well-known Findbugs, can detect more than 400 bug patterns in Java programs through bytecode analysis. Differently, PMD supports multiple languages and is known to be easily integrated into the build process. We use the two static bug detectors with their default configuration.

3.1.2 Analyzed Open-Source Systems

Our study includes four Java open-source systems, *JClouds*, *Kafka*, *Spring-boot* and *Guava*. Two of the systems (i.e., *JClouds* and *Kafka*) are used to summarize the insufficiencies of the SOA and provide reasons for the introduction of false positives. The other two systems (i.e., *Spring-boot* and *Guava*) are selected to evaluate both approaches systematically. We applied the two static bug detectors to all the revisions in a specific development period of the four studied software systems. We started with the official releases of the two software systems when we started this study, i.e.,

Table 3.1: The studied systems and development periods. The release marks the end date of our studied development period, and we include 18 months development history before the specified release.

	KLOC	# Commits	Release	# Average Warnings	
				PMD	Spotbugs
Kafka	434	2,000	2.3.1-rc2	12,972	27,911
JClouds	494	300	2.1.0	18,176	2,090
Spring-boot	2695	400	v2.3.6	5,931	6,918
Guava	2112	2,000	v20.0	6,474	3,819

2.3.1-rc2 of *JClouds*, 2.1.0 of *Kafka*, v2.3.6 of *Spring-boot* and v20.0 of *Guava*. We selected the last commit of the studied release as the end date and its previous one and a half year as the studied development history. We were not able to successfully compile some revisions of systems in the studied period and excluded them from further studies. Besides, we also excluded the revision that has multiple pre-commit revisions. Table 3.1 lists the statistics of the studied systems, including the lines of code (LOC), the number of analyzed commits, the official release that we used to decide the end date of the studied development history, and the number of aggregated warnings in all the analyzed commits.

3.2 Collecting the Dataset Through Manual Labelling

Before we describe how we collect the dataset, i.e., a list of static code warnings and their evolution status, we would like to motivate a few key points that drive our design choice in crafting the dataset. First, given a large number of accumulative warnings across the revisions, we have to set our priorities, i.e., the evolution status of which static code warnings can better showcase the performance of the tracking approach since we have limited manual resources to spare. Second, it is not surprising that in reality, the majority of the warnings persist in the codebases (Johnson, Song, Murphy-Hill, & Bowdidge, 2013). Therefore we do not consider it particularly interesting to include a corresponding percentage of persistent warnings in the dataset. Third, considering the downstream software engineering research this study can benefit from, we set our priorities to focus more on the static warnings that are *resolved* or *newly-introduced*. Fourth, when doing sampling, whenever possible, we include all the warnings **in one commit** due to the inherent challenge in

the mapping problem: one incorrect mapping may impact others, if only including one, we may observe part of the impact (i.e., "the tip of the iceberg"). Last, we have certain confidence in the performance of the SOA approach from its design. For example, we find that majority of the established mappings by the SOA approach is by the *exact mapping* (e.g., 3,137 out of the 3,163 by Spotbugs in *JClouds*-09936b5). The exact matching is the most strict matching process and rarely produces wrong results.

Guided by these key points, we decide to craft the dataset based on the tracking results of the SOA approach and set our priorities on resolved and newly-introduced static code warnings. As illustrated in Chapter 3.1, we apply the static bug detectors on a total of 2,300 commits in *Kafka* and *JClouds*. We re-implemented the SOA approach based on the original paper (Avgustinov et al., 2015) with references to a recent implementation by Liu et al. (Liu et al., 2018). Note that the implementation by Liu et al. is based on *Findbugs*. Thus, we decided to re-implement the SOA approach for *PMD* and *Spotbugs*. Then, we applied the SOA approach to track the evolution of the static code warnings across all the analyzed commits.

We select a subset of static code warnings for manual labelling following the steps:

(1) For *JClouds_Spotbugs* and *JClouds_PMD*, we include all the static warnings labelled as resolved by the SOA approach.

(2) Since there are many (i.e., 2,038 and 1,359) *resolved* static warnings in *Kafka_PMD* and *Kafka_Spotbugs*, we took a statistically significant (95%±5%) sample, i.e., 326 and 301 *resolved* static code warnings for both of them. Using *Kafka_PMD* as an example, we pursued the sampling process by firstly getting an estimation on the sample size, i.e., 323 warnings, then starting to randomly select one commit from the 436 *Kafka* commits with at least one resolved warning, until we collected more than 323 warnings. In the end, we collected 326 resolved warnings from 53 commits in *Kafka_PMD*.

(3) In each of the four settings, there exist a large number of *newly-introduced* code warnings. Hence, we took a statistically significant sample (95%±5%) of warnings in each setting and followed a similar sampling process as Step 2, i.e., including all newly-introduced warnings in one sampled commit. In the end, we collected totally 704 warnings (i.e., in 47 commits)

Table 3.2: A summary on how we collect the static code warnings based on the results of the SOA approach.

	SOA: "Resolved"		SOA: "Newly-Introduced"	
	# Commits	# Warnings	# Commits	# Warnings
PMD				
JClouds	57	280	19	155
Kafka	53	326	14	255
Spotbugs				
JClouds	23	104	5	78
Kafka	26	301	9	216
Total	159	1,011	47	704

labeled as 'newly-introduced' by the SOA approach.

Table 3.2 summarizes the breakdown of the static code warnings we collect following the above-mentioned steps. Note that in Table 3.2, the evolution statuses such as *resolved* and *newly-introduced* are labeled by the SOA approach, which might be incorrect. We performed a manual analysis to reveal the true evolution status of each warning. Table 3.3 summarizes the ground-truth evolution status on the dataset. In total, our dataset contains 1,715 static code warnings and their true evolution status in the development history of *JClouds* and *Kafka*: 37.6% are persistent, 32.2% are resolved, and 30.1% are newly-introduced. In particular, two of the researchers individually performed a manual analysis to uncover the *ground-truth* evolution status of each selected static code warning. The manual analysis includes understanding the nature of each static code warning and code changes that may evolve the code warnings. The two researchers discussed the labels to resolve any disagreements. In our experiments, most of the disagreements are caused by human errors and can be easily agreed on. We calculated Cohen's kappa to measure the inter-rater agreement, which is the almost perfect level (0.96) in our experiment.

It is noticeable that there exists a non-trivial discrepancy between Table 3.2 and Table 3.3 regarding the distribution of the evolution statuses. That is because the SOA approach produces a non-negligible number of incorrect results. We present more details on the inaccuracies in the next chapter.

Except for the resolved and newly-introduced warnings, we also inspect the persistent warnings of the SOA approach. Due to a large number of persistent warnings, we take a statistically significant (95%±5%) sample, i.e., 384 persistent warnings. After checking all of them, we do not find there

Table 3.3: A summary of the 1,715 static code warnings in the dataset based on manually-labeled evolution statuses. The dataset with ground-truth labels is used to evaluate the SOA approach and ours.

	Persistent	*Resolved*	*Newly-Introduced*
PMD			
JClouds	235	102	98
Kafka	170	178	233
Spotbugs			
JClouds	32	86	64
Kafka	208	187	122
Total	645	553	517

Table 3.4: The performance of the SOA approach.

	# Resolved by SOA			# New-Introduced by SOA		
	Total	True Positive	False Positive	**Total**	True Positive	False Positive
PMD						
JClouds	280	102 (36.4%)	178 (63.6%)	155	98 (63.2%)	57 (36.8%)
Kafka	326	178 (54.6%)	148 (45.4%)	255	233 (91.4%)	22 (8.6%)
Spotbugs						
JClouds	104	86 (82.7%)	18 (17.3%)	78	64 (82.0%)	14 (18.0%)
Kafka	301	187 (62.1%)	114 (37.8%)	216	122 (56.5%)	94 (43.5%)
Total	1,011	553 (54.7%)	458 (45.3%)	704	517 (73.4%)	187 (26.6%)

are any mismatching pairs. The sampled persistent are available online (*The shared dataset*, 2020).

3.3 Investigating the Inaccuracies of the SOA approach

```
--- org/jclouds/SGOrCreate.java
+++ org/jclouds/SGInRegOrCreate.java
    @@ -73,4 +71,4 @@
73 71   RegionName input = new RegionName();
74 72
75    - Func<SGReg> group = new Func<SGReg>() {
   73 + Func<SG> group = new Func<SG>() {
76 74
```

Figure 3.2: An example of the false positives due to class renaming.

For the crafted dataset of 1,715 static code warnings, we manually uncover their ground-truth evolution status (i.e., one of *persistent*, *resolved*, or *newly-introduced*), and compare their status decided by the SOA approach. Table 3.4 summarizes the performance of the SOA approach on

Table 3.5: Six causes of False Positives.

Cause	Number
1. Class relocating or renaming	106
2. Method renaming	33
3. Attribute and variable renaming	46
4. Code shifting	135
5. Volatile class/method/variable names	93
6. Drastic and non-refactoring code changes	232

the crafted dataset. Among 1,011 warnings that are determined as *resolved* by the SOA approach , only 553 (54.7%) are *truly* resolved, i.e., true positives. Among 704 warnings that are determined as *newly-introduced*, only 517 (73.4%) are *actually* newly-introduced, i.e., true positives. The false positives in these two categories are the warnings that are *persistent* as the ground-truth. In short, the overall precision of the SOA approach on the collected dataset is only 62.4%, i.e., (553+517)/1,715. Our evaluation of the SOA approach reveals that tracking the evolution of static code warnings over the development period is not that straightforward. The low precision of the SOA approach will negatively impact many downstream software engineering tasks, such as mining fix patterns from software repositories or performing empirical studies on software quality.

Till this end, we answer **RQ1** after examining the performance of the SOA approach by analyzing a number of tracked static code warnings.

> ***RQ1***. *Is the SOA approach good at tracking the evolution of static code warnings?*
> *We present a dataset of 1,715 static code warnings and their evolution statuses. The dataset is crafted with support from the SOA approach. The precision of the SOA approach on the dataset is only 62.4%, and this tracking approach will impact many downstream software engineering tasks negatively.*

Furthermore, we manually analyzed the insufficiencies of the SOA approach, i.e., based on 645 cases, and concluded into six categories as follows. Table 3.5 summarizes six causes of false positives in our dataset.

3.3.1 Class Relocating or Renaming

In the SOA approach, specifically in the first three types of matching algorithms (i.e., exact_matching, location_matching, and snippet_matching, in Chapter 2), having the same class name is a condition that must be satisfied. Hence, if a Java class is renamed or relocated, these three types would fail to find a mapping. The last matching strategy, namely hash-based matching, may handle some of such cases. However, hash-based matching is highly sensitive to code changes, i.e., if the hashed region has code changes, the hash values would alter,and then the hash-based matching strategy would also fail to match the warnings. In total, we find 106 false positives in this category.

Figure 3.2 is a case of a false positive due to Class renaming. The top two lines show the class file has been renamed, i.e., a different file name and also a different file path, which will cause the first three matching strategies to fail. Hash-based matching has the potential to handle this case; however, this file does have code changes, e.g., lines 73-75, and then hash-based matching may fail. In the end, there exist no mappings established by the SOA algorithm at all due to the file renaming. As a result, warnings in `SGOrCreateTest.java` are considered resolved even though the code changes do not resolve the static warnings.

```
    @@ -92,5 +81,5 @@
92 81   @Test
93    - public void shouldCllloseOpenIterators() {
   82 + public void shouldCloseOpenRange(){
94 83    nodes.put(new Node("host1",8121));
95 84    nodes.put(new Node("host2",8122));
96 85
```

Figure 3.3: An example of false positives due to method renaming.

3.3.2 Method Renaming

Similar to class renaming, method renaming affects the tracking in a similar fashion, i.e., the same method name is required in the first three matching strategies. Different method names force the SOA approach to rely on the hash-based matching strategy. However, the hash-based matching strategy is sensitive to regional code changes, which causes it unable to map the persistent warnings between two revisions. In total, we find 33 false positives in this category. Figure 3.3 is a case of a false positive due to method renaming. There is one warning reported in the pre-commit

revision (line 95), which persists (line 83) in the post-commit revision. However, as the method name changes (i.e., highlighted in red and green), the first three matching strategies cannot match the warnings in the method. Thus the SOA approach relies on hash-based matching strategy. Due to the high sensitivity of the hash-based matching strategy, some persistent warnings in this method will not be mapped at all, which leads to inaccurate evolution statuses.

```
    @@ -64,6 +64,6 @@
64 64    this.name = name;
65 65    this.des = options.description();
67    -  this.IPAddress = option.ipAddress();
   67 +  this.ipAddress = option.ipAddress();
68 68    this.portRange = option.portRange();
69 69    this.target = option.target();
70 70
```

Figure 3.4: An example of the false positives due to attribute renaming.

3.3.3 Attribute and Variable Renaming

Attribute and variable renaming, as a common refactoring, also causes the SOA approach to malfunction. However, the impact process is different from the two above-mentioned cause categories. The SOA approach uses attribute and variable information to filter out some matching candidates. Obviously, the two warnings with the attribute or variable information cannot be matched after such changes. Totally, we find 46 false positives in this category.

Figure 3.4 is a case of a false positive due to Attribute and Variable renaming. The same warning is located in line 67 of the pre-commit revision and the post-commit revision. An attribute IPAddress is renamed to ipAddress, which causes false positives in the original algorithm.

```
200 204
201 205    assertEquals(a,null);
202 206
203 207    assertEquals(b,null);
204 208
205 209    assertEquals(c,null);
```

Figure 3.5: An example of the false positives due to code shifting.

3.3.4 Code Shifting

Commits may modify the line numbers of some code statements, although these code statements are not directly modified by the commits. We call this code shifting. Because there exist similar code statements with similar static code warnings (e.g., same warning type, same variable, etc.), when code shifting happens, the SOA approach does not always handle the shifting well, and false positives will be produced. Totally, we find 135 false positives in this category.

Figure 3.5 shows an example of how code shifting may cause the SOA approach to malfunction. Even though the three statements remain unchanged, their line numbers become different. In the pre-commit revision, the line numbers are 201, 203, and 205, while in the post-commit revision, the line numbers are 205, 207, and 209. The SOA approach uses line numbers as part of the code range to build a mapping. As a result, the warning in line 205 from the pre-commit revision is mapped with the warning in line 205 from the post-commit revision by exact matching. This incorrect mapping causes the warnings on line 201 and 203 from the pre-commit revision to be considered as *resolved* while they actually persist.

```
groups.map(_ -> getAcl(opts,
        Set(Read))).toMap[ResourcePatternFilter, Set[Acl]]
```

Figure 3.6: An example of Scala code that has implicit code changes. The metadata of the relevant warning from the pre- and post-commit revisions are shown in Figure 3.7 and Figure 3.8.

```
1  <WarningInstance>
2    <WarningType>SE_BAD_FIELD</WarningType>
3    <Project>kafka</Project>
4    <Class>AclCommand</Class>
5    <Method></Method>
6    <Field>opts$4</Field>
7    <FilePath>kafka/admin/AclCommand.scala</FilePath>
8    <StartLine>206</StartLine>
9    <EndLine>206</EndLine>
10 </WarningInstance>
```

Figure 3.7: The warning information from pre-commit revision.

3.3.5 Volatile Class/Method/Variable Names

Even though there is no explicit code changes in one commit, on certain files, the warning reports by *Spotbugs*, which uses bytecode analysis, are sensitive to compilation. Although everything

```
<WarningInstance>
  <WarningType>SE_BAD_FIELD</WarningType>
  <Project>kafka</Project>
  <Class>AclCommand</Class>
  <Method></Method>
  <Field>opts$1</Field>
  <FilePath>kafka/admin/AclCommand.scala</FilePath>
  <StartLine>330</StartLine>
  <EndLine>330</EndLine>
</WarningInstance>
```

Figure 3.8: The warning information from post-commit revision.

else remains unchanged, *persistent* warnings across revisions may have different line numbers or different class/method/variables names. Such differences will cause all the matching strategies to malfunction. This happens frequently in Scala code when anonymous classes and methods are used heavily. Then some persistent warnings are not matched correctly. Totally, we find 93 false positives in this category.

Figure 3.6 is an example of false positives even though there are no explicit code changes. The line number of the code line with a warning changes from 206 to 330. We examined the metadata of this warning across two revisions (Figure 3.7 and Figure 3.8) and found that not only the line numbers are different, the variable names are also different (line 6 in Figure 3.7 and line 6 in Figure 3.8).

3.3.6 Drastic and Non-refactoring Code Changes

If there exist drastic code changes in close proximity with the persistent warnings, it is possible that location-based matching will not decide a mapping pair. Furthermore, if there exist code changes surrounding the code locations of the warnings, snippet-based matching strategy may also malfunction to establish correct mappings. Figure 3.9 shows an example of such a case. A warning is located from lines 382 to 387 of pre-commit revision, and the potential matched warning is from lines 463 to 467 of post-commit revision. Location-based matching cannot match them because the difference of offsets is higher than the location matching threshold. Snippet-based matching also fails due to the modified snippet.

3.3.7 Discussions on Composite False Positives

We notice that many false positives have more than one cause. For example, a file combines class renaming with attribute renaming or drastic non-refactoring code changes.

```
     @@ -381,5 +429,37 @@
381 429   @Test
    430 +   ...
    431 +   ...
    ... +   ...
382 463   final Runtime = new RuntimException()
383 464
384     -  client.Response(new RequestMatcher()){
385     -     @Override
    465 +  cient.Response(body -> {
386 466   ... ...
387 467  }
```

Figure 3.9: An example of false positives due to a change of method name and drastic code changes.

3.4 A Summary of Our Investigation

Overall, to answer **RQ2**, we performed further manual analysis on the 645 FPs of tracked static warnings. We summarized six main causes which leads to FPs in the SOA approach. There are Class relocating or renaming, Method renaming, Attribute and variable renaming, Code shifting, Volatile class/method/variable names, and Drastic and non-refactoring code changes.

> ***RQ2***. *What are the limitations of the SOA approach?*
>
> *We perform further manual analysis on the FPs of the crafted dataset, and identify six main causes behind the inaccuracies of the SOA approach in tracking the evolution of static code warnings.*

Chapter 4

A Better Approach and Its Comparison with the SOA Approach

In this chapter, guided by our manual analysis results, we propose to improve the SOA approach by better handling refactoring changes and revises a few key steps to improve the accuracy of irrelevant code changes. In particular, our proposed approach (as illustrated in Algorithm 2) reuses the three matching strategies of the SOA approach (i.e., Exact matching, Location-based matching, and Snippet-based matching) and revise a few key steps to improve the inaccurate tracking.

4.1 Improvement 1 - Including Refactoring

We include the refactoring information to improve the tracking using RefactoringMiner (Tsantalis, Mansouri, Eshkevari, Mazinanian, & Dig, 2018). We firstly create a replica of w_i (namely $w_i^{'}$), which is from the pre-commit revision, and then modify the metadata of $w_i^{'}$ with the information from RefactoringMiner. For instance, if RefactoringMiner reveals that the class in w_j is a result from a refactoring of "move and rename class", we modify the class name in $w_i^{'}$ with the one after the refactoring activity. Two of the matching strategies (i.e., snippet matching in line 10 and location matching in line 13) are re-applied to decide two warnings (i.e., w_i, and w_j) whether they are candidates of a matched pair. In particular, Hash-based matching is designed to handle the case of the class files renamed or moved that are included in refactoring information. Thus we remove

Algorithm 2: The algorithm of our improved approach.

Input: The set of warnings from the pre-commit revision, W_p; The set of warnings from the post-commit revision, W_c;

Output: The set of resolved warings, $W_{resolved}$; The set of newly-introduced warings, $W_{newly-introduced}$; The set of matched pairs, $MatchedPairs$;

```
1  Construct W_c^hash, a hash index of W_c
2  Initialize a Two-dimensional array HMatrix.
3  Remove all Identifiers in W_p and W_c
4  for each w_i in W_p do
5      if source file of w_i is not a changed file then
6          take ExactMatching(w_i, W_c^hash[h(W_i)]);
7      else
8          w_i' = refactoring(w_i);        ▷ if there is no refactoring in the location of w_i, w_i' = w_i.
9          for each w_j in W_c do
10             else
11                 take SnippetMatching(w_i', w_j);
12                 if there is a candidate from snippet matching then
13                     HMatrix[i][j] += 1;
14                 take LocationMatching(w_i', w_j');
15                 if there is a candidate from location matching then
16                     HMatrix[i][j] += 1;
17 MatchedPairs = Hungarian(HMatrix);
18 W_resolved = W_p − MatchedPairs;
19 W_newly−introduced = W_c − MatchedPairs;
```

hash-based matching.

As of now, we include **22** types of refactoring that cause the modified metadata of warnings. Table 4.1 shows the refactoring types we include.

4.2 Improvement 2 - Decide Matched Pairs Using Hungarian Algorithm

Commonly, a warning of pre-commit revision may have more than one of matched warnings from post-commit. Thus it is a problem which one should be matched up. In the SOA approach, it takes the first-come-first-matched, which may cause mismatching. Besides, the order of the matching strategies will affect the result. For example, we may get different results when we adopt

Table 4.1: Refactoring types included in our proposed approach.

Refactoring type	Modified metadata of static wanrings
1. Extract method	Method name
2. Rename method	Method name
3. Move class	Class name
4. Rename class	Class name
5. Rename variable	Field name
6. Rename parameter	Field name
7. Rename attribute	Field name
8. Move and rename class	Class name
9. Move Method	Method name
10. Move attribute	Field name
11. Pull up method	Method name
12. Pull up attribute	Field name
13. Push down method	Method name
14. Push down attribute	Field name
15. Extract Superclass	Class name
16. Extract and move Method	Method name
17. Extract Class	Class name
18. Extract Subclass	Class name
19. Move and Rename Attribute	Field name
20. Replace Variable with Attribute	Field name
21. Move and Rename Method	Method name
22. Move and Inline Method	Method name

location-matching first and snippet-matching first. The order in the SOA approach is doing Exact matching first, then Location-based matching, and last one, Snippet-based matching. In our investigation, this order has introduced many false positives like code shifting (Figure 3.5). Besides, the first-matched warning may not be the best or correct one,i.e., there exist better-matched warnings. Thus we adopt **Hungarian algorithm**, a classic approach to solve the assignment problem in bipartite graphs. When a warning of post-commit revision is found that can be matched with a warning of pre-commit revision from the two matching strategies (i.e., Location-based matching and Snippet-based matching), instead of deciding it as a matched pair (i.e., a persistent warning), we construct a Hungarian matrix to save it as a potential matched pair. An example is like Figure 4.1. $w1_p$,$w2_p$ and $w3_p$ are the warnings from parent revision. $w1_c$,$w2_c$ and $w3_c$ are the warnings from post-commit revision. When two warnings are considered as a (potential) matched pair, the Hungarian matrix adds one (e.g., $w1_p$ and $w1_c$). A value of 2 (e.g., $w2_p$ and $w2_c$) means they are a (potential) matched pair

from both matching strategies. It also means that this pair is more likely to be an actual pair of persistent warnings. If the SOA is applied on the six static warnings , it is possible that $w1_p$ is matched with $w1_c$, and $w2_p$ is matched with $w3_c$, so $w3_p$ and $w2_c$ become false positives. In our algorithm, we construct a matrix $HMatrix$ (line 2) like Figure 4.1. The size of $HMatrix$ is (the number of W_p) $*$ (the number of W_c) and the values are 0 initially. Two matching strategies, Snippet-based matching and Location-based matching are used to find out the potential matched warnings. Then we leverage maximum matching to decide the matched pairs. Besides, there is an Exact matching for changed files in the SOA matching, but if we adopt **Hungarian algorithm**, the matched warnings by Exact matching can also be identified by Location-based matching or Snippet-based matching. Thus, we simply remove Exact matching for changed files in our approach. However, we keep it for unchanged files.

	$w1_p$	$w2_p$	$w3_p$
$w1_c$	1	1	0
$w2_c$	1	2	0
$w3_c$	0	1	1

Figure 4.1: A simple example of Hungarian matrix.

4.3 Improvement 3 - Working with Volatile Identifiers

Anonymous classes and methods are given an identifier after compilation. However, the assigned identifiers are sensitive to change when there are code changes, even irrelevant. We try to minimize such sensitivity by removing the variable part in such identifiers. In particular, for identifiers such as *opt$1*, we use a regular expression to remove the numeric suffix after $ and only keep *opt* as the variable identifier in the metadata of a warning for the subsequent matching process.

4.4 Evaluation

We evaluate our improved approach on the crafted dataset to show how much improvement our approach has compared to the SOA approach and answer **RQ3** by conducting an evaluation. In addition to *J*Clouds and *K*afka, we also select two other open-source software systems (i.e., *S*pring-boot and *G*uava), which provides a systematic evaluation between our approach and the SOA approach. Two static bug detectors are applied on a total of 400 commits for *Spring-boot* and 2,000 commits for *Guava*. Then we take the SOA approach on them. Table 4.2 shows the results of the SOA approach.

Table 4.2: A summary on *G*uava and *S*pring-boot based on the results of the SOA approach.

	SOA: "Resolved"		SOA: "Newly-Introduced"	
	# Commits	# Warnings	# Commits	# Warnings
PMD				
Spring-boot	59	218	82	277
Guava	220	1241	281	1417
Spotbugs				
Spring-boot	17	193	22	182
Guava	344	1164	430	1441
Total	640	2816	815	3317

For *S*pring-boot_Spotbugs and *S*pring-boot_PMD, all static warnings are included. We take the same sample strategy on *G*uava_Spotbugs and *G*uava_PMD, a statistically significant (95%±5%) sample of resolved warnings with newly-introduced warnings of their commits, i.e., 41 commits with 296 resolved warnings and 188 newly-introduced warnings in *G*uava_PMD, and 44 commits with 289 resolved warnings and 204 newly-introduced warnings in *G*uava_Spotbugs. Our approach is also applied to these commits. Sampled warnings and the warnings from our approach are labeled by two authors with Cohen's kappa coefficient of 0.62, which has a substantial agreement. Table 4.3 illustrates the labeled results of the two projects.

Since tracking the static code warnings is not a standalone task for each warning, it is, in fact, a mapping problem between two sets. Hence, we applied our improved approach to **all the warnings in the 200 commits**, which is a superset of the 3,452 warnings in the manually-labeled dataset. The remaining warnings in the 200 commits, while not in our crafted dataset, have a pre-assumed label, "*persistent*". If our approach changes the pre-assumed label of some warnings, then we manually

Table 4.3: The labeled results of the two projects.

	SOA: Resolved		SOA: Newly-Introduced	
	TP/FP (SOA)	TP/FP (our approach)	TP/FP (SOA)	TP/FP (our approach)
PMD				
Spring-boot	138/80	138/22	109/80	109/22
Guava	172/124	172/15	64/124	64/15
Spotbugs				
Sprin-boot	186/7	186/1	153/7	153/1
Guava	231/58	231/15	151/53	151/10
Total	727/269	727/53	477/264	477/48

examine the ground-truth labels of these warnings. If our approach does not change the pre-assumed labels, it means that our approach is **at least not worse** than the SOA approach on the warnings that are not in our evaluation dataset.

Table 4.4: The performance comparison between the SOA approach and our approach. Note that FP is short for false positive. A lower FP ratio is desired.

	Resolved		Newly-Introduced	
	FP (SOA)	FP (our approach)	FP (SOA)	FP (our approach)
PMD				
JClouds	63.6% (178/280)	3.8% (4/106)	36.8% (57/155)	8.4% (9/107)
Kafka	45.4% (148/326)	27.0% (66/244)	8.6% (22/255)	4.1% (10/243)
Spring-boot	36.7% (80/218)	13.8% (22/160)	42.3% (80/189)	16.8% (22/131)
Guava	41.9% (124/296)	8.0% (15/187)	66.0% (124/188)	19.0% (15/79)
Spotbugs				
Jclouds	17.3% (18/104)	1.1% (1/87)	17.9% (14/78)	3.0% (2/66)
Kafka	37.8% (114/301)	15.8% (35/222)	43.5% (94/216)	17.0% (25/147)
Spring-boot	3.6% (7/193)	0.5% (1/187)	4.4% (7/160)	0.6% (1/154)
Guava	20.1% (58/289)	6.1% (15/246)	26.0% (53/204)	6.2% (10/161)
Total	36.2% (727/2007)	11.1% (159/1437)	31.2% 451/1445)	8.6% (94/1087)

Table 4.4 lists the comparison results between the SOA approach and our improved approach on the collected dataset of 3,452 static code warnings. Note that there are 3,452 static warnings from the SOA approach. However, when we applied our approach on the same dataset, we obtained only 2,524 resolved and newly-introduced warnings, which means that the rest (i.e., 928 warnings) are identified by our approach as persistent warnings. We categorized the 3,452 warnings into two categories according to the labels by the SOA approach for ease of comparison. The evaluation shows that our proposed approach can significantly reduce the false positive rate. Overall, for the

Table 4.5: The evaluation of execution time for both approaches.

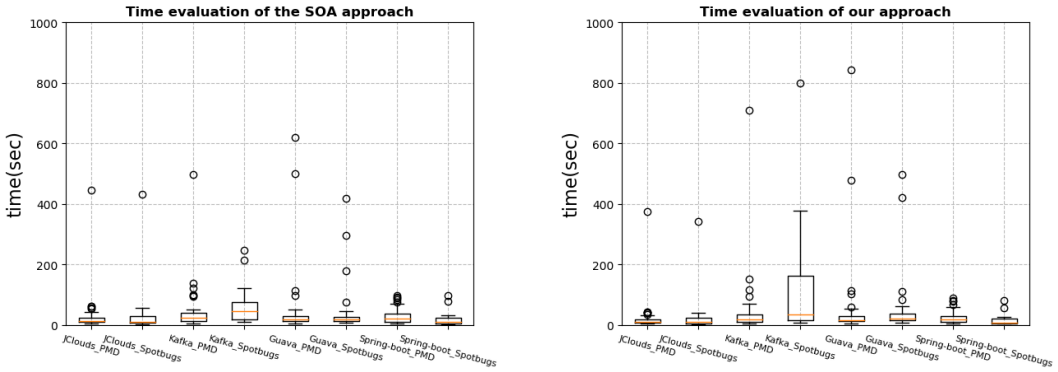
	The SOA approach		Our approach	
	Median	Average	Median	Average
PMD				
JClouds	12.9	25.8	9.0	20.5
Kafka	23.0	41.0	18.1	41.2
Guava	18.0	50.5	16.2	55.2
Spring-boot	21.2	29.5	17.7	24.3
Spotbugs				
JClouds	10.9	30.4	9.3	24.0
Kafka	45.6	60.4	33.9	119.2
Guava	17.7	39.4	19.2	45.9
Spring-boot	9.3	20.4	7.8	16.8

3,452 warnings, the SOA approach has 1,178 warnings with a wrong evolution status, i.e., the false positive rate is 34.1%. Compared to that, our proposed approach reduces the false positives significantly, from 1,178 to 253, i.e., the false positive rate drops to 10.0%, yielding a precision of 90.0%. To give an example in Table 4.4, for JCloud with PMD, the SOA labels 168 persistent warnings wrongly as *resolved* while our approach correctly labels 160/168 as persistent, leaving eight false positives with a wrong label. **Our approach reduces the false positives by *correctly labeling the persistent* warnings, which are mistakenly labeled as *resolved* or *newly-introduce* by the SOA approach.**

Among the six causes of the false positives of the SOA approach, our approach is shown to effectively reduce false positives for all causes. Table 4.6 shows the breakdown of the left false positives by each cause after using our approach on the resolved warning dataset.

Additionally, the execution time between the two approaches is evaluated on four projects. Table 4.5 shows that the SOA approach is faster than our approach on *Kakfa_Spotbugs*. It has no significant difference in others. Figure 4.2 shows box plots of time evaluation for both the SOA approach and our approach. The time performance of our approach is not bad compared with the SOA approach.

Till this end, to answer **RQ3**, we proposed a tracking approach combining three improvement



a b

Figure 4.2: The box plot of time execution for both approaches.

Table 4.6: Six causes of False Positives after using our approach in Kafka and JClouds.

Cause	Number
1. Class relocating or renaming	23
2. Method renaming	2
3. Attribute and variable renaming	3
4. Code shifting	4
5. Volatile class/method/variable names	1
6. Drastic and non-refactoring code changes	119

steps, which outperforms the SOA approach.

> ***RQ3***. *Can our proposed approach perform better than the SOA approach?*
>
> *We proposed a better tracking approach. Compared with the SOA approach, our approach can reduce FPs significantly (i.e., from 1,178 to 253) and yield a precision of 90.0%.*

Apart from the SOA approach, we also take an independent evaluation of our approach by a statistically significant (95%±5%) sample on commits for each project to answer **RQ4**. Our approach is applied on sampled commits to collect resolved and newly-introduced warnings. Then two authors manually check them to determine whether a warning is a false positive or a true positive with Cohen's kappa coefficient of 0.82. Table 4.7 shows the number of commits we sampled. Note that there are a lot of commits that have no resolved or newly-introduced warning in this evaluation. In other words, the code changes of many commits are too small to change the status of all static warnings. Totally, we sampled 2,014 commits with 495 resolved and 837 newly-introduced

Table 4.7: The sampled commits for the independent evaluation.

	# Commits	# Resolved	# Newly-Introduced
PMD			
JClouds	169	83	150
Kafka	322	154	152
Guava	322	40	109
Spring-boot	194	6	17
Spotbugs			
JClouds	169	41	107
Kakfa	322	98	186
Guava	322	53	106
Spring-boot	194	20	10
Total	2,014	495	837

Table 4.8: The performance of our approach in the independent evaluation.

	TP (Resolved)	# TP (Newly-Introduced)
PMD		
JClouds	97.6% (81/83)	96.0% (144/150)
Kafka	78.6% (121/154)	95.4% (145/152)
Guava	75.0% (30/40)	91.7% (100/109)
Spring-boot	100% (6/6)	100% (17/17)
Spotbugs		
JClouds	97.6% (40/41)	98.1% (105/107)
Kakfa	82.7% (81/98)	90.3% (168/186)
Guava	96.2% (51/53)	98.1% (104/106)
Spring-boot	100% (20/20)	100% (10/10)
Total	86.9% (430/495)	94.7% (793/837)

warnings by our improved approach. Table 4.8 shows the performance of our approach. Overall, Our approach has a great performance with a precision of 91.8% (i.e., (430+793)/(495+837)), which means our approach can handle the task of tracking static warnings very well.

RQ4 could be answered by conducting an independent evaluation on our approach.

> ***RQ4***. *How accurate is our proposed approach for tracking the evolution of static code warnings?*
>
> *By conducting an independent evaluation on our approach, results show that our approach achieves a precision of 91.8%, and it can handle the task of tracking static warnings very well.*

Chapter 5

Threats to Validity

5.1 External Threats

In this thesis, we focus on tracking the static code warnings in Java projects. Our study results may not be generalizable to projects in other languages. It is expected that programs with similar evolution details to Java systems may benefit from our study. We include two static bug detectors in our study, whose representation of static code warnings are similar to some extent, i.e., use of class/method/variable names and code ranges for matching purposes. The improvement of our proposed approach may not be generalizable to a static bug detector with a totally different set of metadata of the reported warnings. However, most of the popular static bug detectors provide similar information. Last, our crafted dataset for evaluating and improving the SOA approach is based on two open-source projects. To increase the diversity, we analyzed a reasonable number of commits in the two projects. In general, we find that the evolution details that make the SOA approach malfunction are consistent in our collected dataset.

5.2 Internal Threats

When it comes to manually label the dataset, human errors are inevitable. We tried to reduce human errors by having two persons annotating the dataset and resolve conflicts through discussions. Although our dataset covers warnings with all three evolution statuses, we do not claim that

our dataset is representative in terms of following the distribution of the three evolution statuses. In particular, we set our criteria in crafting the dataset based on our observations on the SOA approach (i.e., most of the established mappings are correct) and also our priorities, which is to focus on the resolved and newly-introduced warnings.

Chapter 6

Related Work

6.1 Tracking the Evolution of Code Issues

Tracking the evolution of code issues, whether bugs, code smells, or static code warnings, is a central question in many software quality studies. For example, the *SZZ* algorithm, which identifies the origin of bug-introducing commits, is widely used in defect prediction studies. Recent evaluations have uncovered many previously unknown deficiencies in *SZZ* and inspire many researchers to work on improving *SZZ*. For example, a study (Neto, da Costa, & Kulesza, 2018) empirically investigated how bug-fix changes and bug-introducing changes of the *SZZ* are impacted by code refactoring. Then they proposed refactoring-aware *SZZ*. Another study (da Costa et al., 2017) proposed a framework to provide a systematic evaluation of the data collected by *SZZ*. Palix et al. conducted two studies on mining the code patterns. The first study (Palix, Lawall, & Muller, 2010) presented a language-independent tool for mining and tracking code patterns across the evolution of software by building graphs and computing statistics. Their other study (Palix, Falleri, & Lawall, 2015) combined the tool with AST for the detection of code patterns across multiple versions. There is a study (Querel & Rigby, 2018) that presented a tool that combines static analysis with statistical bug models to detect which commits are likely to contain risky codes, which provides more precise information of a static warning. Dong-Jae et al. (D. J. Kim, Tsantalis, Chen, & Yang, n.d.) conducted an empirical study on the evolution of annotation changes and create a taxonomy to uncover what annotation changes have and the motivation of annotation changes. In addition, Felix et

al. (Grund, Chowdhury, Bradley, Hall, & Holmes, n.d.) proposed a tool to uncover method histories with no pre-processing or whole-program analysis, which quickly produces complete and accurate change histories for 90% methods.

Compared to tracking the defects, tracking the static code warnings has been increasingly needed in recent research, yet rarely studied for its challenges and insufficiencies. Spacoo et al. (Spacco, Hovemeyer, & Pugh, 2006) propose to match warnings across revisions using a combination of some basic information of each warning (e.g., warning type, class/method names) and allow inexact matching to some extent. Their approach is not able to match warnings if they are moved to a different class/method. Other diff-based approaches are used to identify which static code warnings are resolved. In particular, Sunghun et al. (Kim, Zimmermann, Pan, & Jr. Whitehead, 2006) proposed an algorithm to automatically identify bug-introducing changes with high accuracy by combining the annotation graphs, ignoring non-semantic source code changes. Results show that their algorithm outperforms the *SZZ*. Cathal and Leon (Boogerd & Moonen, 2009) conducted an empirical study to investigate the relation between static warnings and actual faults. More recently, Avgustinov et al. (Avgustinov et al., 2015) proposed to combine several diff-based matching strategies to tackle this problem, which we refer to as the state-of-the-art approach in our study for evaluation and comparison.

However, a proper examination of the performance of the SOA approach is still lacking in the field. In this thesis, we manually crafted a dataset of 1,416 static code warnings and their evolution status from two real-world open-source systems and used it to identify potentials for improvement in the SOA approach.

6.2 Empirical Studies on Static Bug Detectors

Researchers have been working on understanding and improving the utilization challenge of static bug detectors. Johnson et al. (Johnson et al., 2013) study the reasons that developers do not fully utilize static bug detectors via conducting interviews with developers. Results show developers cannot be satisfied with the current static analysis tools due to the high rate of false positives.

This study also provides some suggestions to improve future static tools, e.g., improving the integration of the tool and automatic fixes. Beller et al. (Beller, Bholanath, McIntosh, & Zaidman, 2016) performed a large-scale study to understand the current status of using static bug detectors in open-source systems, e.g., whether or not use, and what running configurations are used. Wang et al. (Wang, Wang, & Wang, 2018) aimed to find whether there is a golden feature to indicate actionable static warnings. Additionally, a survey was conducted by Muske et al. (Muske & Serebrenik, 2016) who reviewed static warnings handling studies as well as collected and classified handling approaches.

Studies are also conducted to understand the nature of the issues found by static bug detectors. Ayewah et al. (Ayewah, Pugh, Morgenthaler, Penix, & Zhou, 2007) discuss the defects found by static bug detectors at Google with regards to false positives, types of warnings generated and their severity. Wedyan et al. (Wedyan, Alrmuny, & Bieman, 2009) found that the issues by static bug detectors are much more related to refactoring than defects. Habib et al. (Habib & Pradel, 2018) study the effectiveness of static bug detectors in terms of their ability to find real defects and find that static bug detectors do find a non-trivial portion of defects. An empirical study (Yan et al., 2017) evaluated the degree of correlation between defects and warnings on the evolution of projects. Tomassi et al. (Tomassi, 2018) examined static bug detectors by considering 320 real java bugs. Their evaluation shows that static analyzers are not as effective in bug detection, with only one bug detected by *S*potbugs. Trautsch et al. (Trautsch, Herbold, & Grabowski, 2019) conducted a longitudinal study on static analysis warning trends. They found that the quality of code with regards to static warnings is improving, and the long-term effects of static bug detectors are positive.

Our study focuses on a different aspect, which is to provide better ways to track how static code warnings involve. Also, our study includes manual analysis on a non-trivial dataset of static code warnings for the purpose of improving the tracking precision, which is not covered by prior work.

6.3 Utilizing the Tracking of Static Code Warnings

Better tracking static code warnings across development history provides many benefits. For example, there has been an increasing interest to conclude fix patterns. Kui et al. (Liu et al., 2018)

mines the fix patterns on static code warnings from the software repository, and the SOA approach was applied in their research. However, they did not conduct an evaluation on the approach about how accurate the SOA approach performs. A study (Bavishi et al., 2019) proposed a novel solution to automatically generate code fixing patches for static code warnings via learning from fixing examples. Another recent work (Yang, Tan, Peyton, & A Duer, 2019) proposed a tool to help developers better utilize static bug detectors on security issues by clustering based on common preferred fix locations. This line of work can definitely benefit from an improved tracking approach. In addition, there have been many works to prioritize and recommend certain types of warnings based on development history. Among them, a study (S. Kim & Ernst, 2007) observed the static warnings in different static bug detection tools and proposed a history-based warnings prioritization to mining the fix cases recorded in the code change history. Results show that over 90% of warnings remain in the projects or removed during code non-fix changes. Ted et al. (Kremenek & Engler, 2003) explored the ranking of warnings from static bug detectors, and presented a technique with a statistical model to rank the static warnings that are most likely to be true positives. In addition, another work, Quinn et al. (Hanam et al., 2014), aimed at actionable static warnings, and presented an actionable alert prediction model by creating feature vectors based on code characteristics. In comparison, our work focuses on the status changes of the static warnings in the evolution of the software projects. The other work (Burhandenny, Aman, & Kawahara, 2016) statistically investigated the trend of static warnings over the releases of OSS products, and introduced a novel metric (e.g., the index of programmers' attention) to analyze the automatically pointed static warnings and the actual attentions which programmers paid to those static warning. Higo et al. (Higo, Hayashi, Hata, & Nagappan, 2020) proposed an approach based on static analysis across the development history to identify project-specific bug patterns. A better tracking mechanism will provide more accurate results for such work.

Chapter 7

Conclusions and Future Work

In this chapter, we summarize the studies and contributions discussed in the thesis, and propose potential future work that might be complementary to this thesis for better understanding and utilizing of tracking the evolution status of static code warnings.

7.1 A Summary of the Thesis

Tracking the evolution of static code warnings across software development history becomes a vital question due to the increasing interest to further utilize static bug detectors by integrating them in developers' workflow, e.g., CI. Also, such tracking is widely used in many downstream software engineering tasks that include performing empirical studies for software quality, learning which static warnings are of more interest, as well as mining fix patterns of static code warnings. This study presents a careful investigation of the performance of the state-of-the-art approach in tracking static code warnings on two open-source projects. In particular, a dataset of 1,715 static code warnings and their evolution status is crafted through manual labeling. We performed a further manual analysis to summarize six main causes of false positives, and proposed an improved tracking approach based on the main causes. Last, this thesis independently evaluates our approach and the SOA approach on two other projects with a dataset of 1,737 static code warnings. Results show that our improved approach outperforms the SOA approach significantly in terms of tracking precision.

7.2 Future Work

This thesis makes a major contribution towards the improvement of the utilization of static bug detectors in the software evolution. However, there are still many open problems that are related to this thesis. We highlight some aspects for future work that may complement this thesis.

7.2.1 Mining the Anti-patterns From Static Code Warnings

There are many newly-introduced warnings detected by our improved approach in our study. We can focus on the newly-introduced warnings to investigate and explore the reasons that new static warnings are introduced by conducting a systemic study and categorize the causes. Through analyzing them, the future study can uncover the frequent newly-introduced static warnings and might provide some suggestions to help developers having better practice in software development and avoid bad programming practice.

7.2.2 Exploring the Relationship Between Code Refactoring and Static Warnings

During the process of our manual analysis in the thesis, we noticed that some resolved warnings are fixed due to code refactoring. Previous studies (Lacerda, Petrillo, Pimenta, & Guéhéneuc, 2020) show that code smells and code refactoring have a strong relationship with quality attributes. Code refactoring is considered an effective process to remove code smells (*Refactoring: Improving the Design of Existing Code*, 1999). In addition, Part of code smells can be detected by static bug detection tools. Thus it is an open problem the relationship between code refactoring and static warnings.

References

Avgustinov, P., Baars, A. I., Henriksen, A. S., Lavender, G., Menzel, G., de Moor, O., ... Tibble, J. (2015). Tracking static analysis violations over time to capture developer characteristics. In *Proceedings of the 37th international conference on software engineering - volume 1* (p. 437–447). IEEE Press.

Ayewah, N., Pugh, W., Morgenthaler, J. D., Penix, J., & Zhou, Y. (2007). Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th acm sigplan-sigsoft workshop on program analysis for software tools and engineering* (p. 1–8). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1251535.1251536 doi: 10.1145/1251535.1251536

Bavishi, R., Yoshida, H., & Prasad, M. R. (2019). Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (p. 613–624). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3338906.3338952 doi: 10.1145/3338906.3338952

Beller, M., Bholanath, R., McIntosh, S., & Zaidman, A. (2016). Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 ieee 23rd international conference on software analysis, evolution, and reengineering (saner)* (Vol. 1, p. 470-481).

Boogerd, C., & Moonen, L. (2009). Evaluating the relation between coding standard violations and faults within and across software versions. In *2009 6th ieee international working conference on mining software repositories* (p. 41-50).

Burhandenny, A. E., Aman, H., & Kawahara, M. (2016). Examination of coding violations focusing on their change patterns over releases. In *2016 23rd asia-pacific software engineering conference (apsec)* (pp. 121–128).

da Costa, D. A., McIntosh, S., Shang, W., Kulesza, U., Coelho, R., & Hassan, A. E. (2017). A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, *43*(7), 641-657.

Grund, F., Chowdhury, S., Bradley, N. C., Hall, B., & Holmes, R. (n.d.). Codeshovel: Constructing method-level source code histories.

Habib, A., & Pradel, M. (2018). How many of all bugs do we find? a study of static bug detectors. In *Proceedings of the 33rd acm/ieee international conference on automated software engineering* (p. 317–328). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/3238147.3238213 doi: 10.1145/3238147.3238213

Hanam, Q., Tan, L., Holmes, R., & Lam, P. (2014). Finding patterns in static analysis alerts: Improving actionable alert ranking. In *Proceedings of the 11th working conference on mining software repositories* (p. 152–161). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/2597073.2597100 doi: 10.1145/2597073.2597100

Higo, Y., Hayashi, S., Hata, H., & Nagappan, M. (2020). Ammonia: an approach for deriving project-specific bug patterns. *Empirical Software Engineering*, 1–29.

Hunt, J. W., & Szymanski, T. G. (1977). A fast algorithm for computing longest common subsequences. *Communications of the ACM*, *20*(5), 350–353.

Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 international conference on software engineering* (p. 672–681). IEEE Press.

Kim, D. J., Tsantalis, N., Chen, T.-H. P., & Yang, J. (n.d.). Studying test annotation maintenance in the wild.

Kim, S., & Ernst, M. D. (2007). Which warnings should i fix first? In *Proceedings of the the 6th joint meeting of the european software engineering conference and the acm sigsoft*

symposium on the foundations of software engineering (p. 45–54). New York, NY, USA: Association for Computing Machinery. Retrieved from https://doi.org/10.1145/1287624.1287633 doi: 10.1145/1287624.1287633

Kim, S., Zimmermann, T., Pan, K., & Jr. Whitehead, E. J. (2006). Automatic identification of bug-introducing changes. In *21st ieee/acm international conference on automated software engineering (ase'06)* (p. 81-90).

Kremenek, T., & Engler, D. (2003). Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th international conference on static analysis* (p. 295–315). Berlin, Heidelberg: Springer-Verlag.

Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval research logistics quarterly*, *2*(1-2), 83–97.

Lacerda, G., Petrillo, F., Pimenta, M., & Guéhéneuc, Y. G. (2020). Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, *167*, 110610.

Liu, K., Kim, D., Bissyande, T. F., Yoo, S., & Le Traon, Y. (2018). Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 1-1.

Liu, K., Koyuncu, A., Kim, D., & Bissyandé, T. F. (2019). AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In X. Wang, D. Lo, & E. Shihab (Eds.), *26th IEEE international conference on software analysis, evolution and reengineering, SANER 2019, hangzhou, china, february 24-27, 2019* (pp. 456–467). IEEE. Retrieved from https://doi.org/10.1109/SANER.2019.8667970 doi: 10.1109/SANER.2019.8667970

Muske, T., & Serebrenik, A. (2016). Survey of approaches for handling static analysis alarms. In *2016 ieee 16th international working conference on source code analysis and manipulation (scam)* (pp. 157–166).

Myers, E. W. (1986). Ano (nd) difference algorithm and its variations. *Algorithmica*, *1*(1-4), 251–266.

Neto, E. C., da Costa, D. A., & Kulesza, U. (2018). The impact of refactoring changes on the szz algorithm: An empirical study. In *2018 ieee 25th international conference on software analysis, evolution and reengineering (saner)* (p. 380-390).

Palix, N., Falleri, J.-R., & Lawall, J. (2015). Improving pattern tracking with a language-aware tree differencing algorithm. In *2015 ieee 22nd international conference on software analysis, evolution, and reengineering (saner)* (pp. 43–52).

Palix, N., Lawall, J., & Muller, G. (2010). Tracking code patterns over multiple software versions with herodotos. In *Proceedings of the 9th international conference on aspect-oriented software development* (pp. 169–180).

Querel, L.-P., & Rigby, P. C. (2018). Warningsguru: Integrating statistical bug models with static analysis to provide timely and specific bug warnings. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 892–895).

Refactoring: Improving the design of existing code. (1999). USA: Addison-Wesley Longman Publishing Co., Inc.

Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., & Jaspan, C. (2018, March). Lessons from building static analysis tools at google. *Commun. ACM*, *61*(4), 58–66. Retrieved from https://doi.org/10.1145/3188720 doi: 10.1145/3188720

Sadowski, C., van Gogh, J., Jaspan, C., Soederberg, E., & Winter, C. (2015). Tricorder: Building a program analysis ecosystem. In *International conference on software engineering (icse)*.

The shared dataset. (2020). Retrieved from https://drive.google.com/drive/folders/1OSYkm6QIfHO7z_zgdZdDRMbIqQOZ5IGW?usp=sharing

Spacco, J., Hovemeyer, D., & Pugh, W. (2006, 01). Tracking defect warnings across versions. In (p. 133-136). doi: 10.1145/1137983.1138014

Spotbugs latest version. (2019). Retrieved from http://spotbugs.readthedocs.io

Tomassi, D. A. (2018). Bugs in the wild: examining the effectiveness of static analyzers at finding real-world bugs. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering* (pp. 980–982).

Trautsch, A., Herbold, S., & Grabowski, J. (2019). A longitudinal study of static analysis warning evolution and the effects of pmd on software quality in apache open source projects. *arXiv preprint arXiv:1912.02179*.

Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., & Dig, D. (2018). Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th international conference on software engineering* (pp. 483–494). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/3180155.3180206 doi: 10.1145/3180155.3180206

Wang, J., Wang, S., & Wang, Q. (2018). Is there a" golden" feature set for static warning identification? an experimental evaluation. In *Proceedings of the 12th acm/ieee international symposium on empirical software engineering and measurement* (pp. 1–10).

Wedyan, F., Alrmuny, D., & Bieman, J. M. (2009). The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *2009 international conference on software testing verification and validation* (p. 141-150).

Yan, M., Zhang, X., Xu, L., Hu, H., Sun, S., & Xia, X. (2017). Revisiting the correlation between alerts and software defects: A case study on myfaces, camel, and cxf. In *2017 ieee 41st annual computer software and applications conference (compsac)* (Vol. 1, pp. 103–108).

Yang, J., Tan, L., Peyton, J., & A Duer, K. (2019). Towards better utilizing static application security testing. In *2019 ieee/acm 41st international conference on software engineering: Software engineering in practice (icse-seip)* (p. 51-60).