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

Junjie Li, *Student Member, IEEE,* and Jinqiu Yang, *Member, IEEE*

Abstract—Static bug detection tools help developers detect problems in the code, including bad programming practices and potential defects. Recent efforts to integrate static bug detectors in modern software development workflows, such as in code review and continuous integration, are shown to better motivate 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. In addition, the utilization of tracking tools enables developers to concentrate on the most recent and actionable static warnings rather than being overwhelmed by the thousands of warnings from the entire project. This, in turn, enhances the utilization of static analysis tools. Hence, precisely tracking the warnings by static bug detectors is critical to improving the utilization of static bug detectors further.

In this paper, we study the effectiveness of the state-of-the-art (SOTA) solution in tracking static code warnings and propose a better solution based on our analysis of the insufficiency of the SOTA solution. In particular, we examined over 2,000 commits in four large-scale open-source systems (i.e., JClouds, Kafka, Spring-boot, and Guava) and crafted a dataset of 3,451 static code warnings by two static bug detectors (i.e., Spotbugs and PMD). We manually uncovered the ground-truth evolution status of the static warnings: persistent, $\text{removed}_{\text{fix}}$, $\text{removed}_{\text{non-fix}}$ and newly-introduced. Upon manual analysis, we identified the main reasons behind the insufficiency of the SOTA solution. Furthermore, we propose StaticTracker to track static warnings over software development history. Our evaluation shows that StaticTracker significantly improves the tracking precision, i.e., from 64.4% to 90.3% for the evolution statuses combined ($\text{removed}_{\text{fix}}$, $\text{removed}_{\text{non-fix}}$ and newly-introduced).

Index Terms—static analysis, code refactoring, software evolution, empirical study.

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 [1]. However, static bug detectors are known to be underutilized due to various reasons. First, static bug detectors report an overwhelming number of warnings, which may be far beyond what resources are allowed to resolve. For example, Spotbugs [2], i.e., the spiritual successor of *Findbugs*, reports 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 actionable warnings and reduce false static code warnings, such as recommending actionable warnings by learning from past development history [3, 4]. 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 [1, 5]. 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 [6], 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., shows the *diffs* of 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 (i.e., *persistent*) by developers, which warnings are fixed (i.e., $removed_{fix}$) by developers, which warnings are removed due to code deletions (i.e., $removed_{non\text{-}fix}$), and which warnings are *newly-introduced* in the commit. The analysis of removed and newly-introduced warnings helps developers discern trends in warning changes over time. Identifying fixed warnings can benefit downstream software engineering research, such as automatic program repair and mining fix patterns of static warnings. The presentation of the four warning types, i.e., $\text{removed}_{\text{fix}}$, $\text{removed}_{\text{non-fix}}$, persistent, and newly-

• *J. Li, and J. Yang are with the Department of Computer Science and Software Engineering, Concordia University, Montreal, Quebec, Canada. E-mail: l_unjie, jinqiuy@encs.concordia.ca*

introduced, provides a comprehensive understanding of the status of warnings across consecutive revisions. Developers may be more motivated to utilize static bug detectors if they are provided with a list of newly introduced static code warnings instead of thousands of persistent warnings that developers showed less interest in. In addition, developers could pay attention to the warnings that have been recently resolved and may be motivated to fix similar warnings.

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 [7, 8], 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 [9].

To this end, there has been little effort to systematically review existing solutions to track the evolution of static code warnings and accordingly propose better solutions. Prior studies rely on simple heuristics to track the static code warnings [4, 10], i.e., two warnings are identical if they are of the same warning type, in the same file, etc. Avgustinov et al. [11] 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 study, we refer to it as the state-of-the-art (SOTA) solution. For example, Liu et al. [7] adapt the SOTA solution to identify warning-fixing commits in software repositories for automated program repair. However, it remains unknown how accurate the SOTA solution is in tracking the static code warnings. An unacceptable performance of the SOTA 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 paper, we examine the performance of the SOTA solution in tracking static code warnings and propose a better approach StaticTracker after analyzing the insufficiency of the SOTA solution. In our study, we found that the SOTA approach has limitations in matching the warnings involved in code refactoring, code shift, and volatile metadata due to the use of an anonymous class or method. In light of these limitations, we propose StaticTracker with three key improvements (1) the detection of code refactoring (2) matching warning pairs using the Hungarian algorithm, and (3) the handling of volatile metadata of static warnings. In addition, StaticTracker is designed to differentiate between the warnings that are fixed by developers and the ones removed due to non-fix evolution. In comparison, the SOTA solution does not distill the two different categories, i.e., $\text{removed}_{\text{fix}}$ and $\text{removed}_{\text{non-fix}}$.

Figure 1 shows an overview of this study. In this work, We answer the following research questions:

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

RQ2 What are the limitations of the SOTA approach?

RQ3 What is the performance of our proposed approach StaticTracker?

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 four projects (i.e., *JClouds*, *Kafka*, *Guava*, and *Springboot*), and performed manual analysis to label whether each sampled static code warning is *persistent*, $removed_{fix}$, $removed_{non\text{-}fix}$, or *newly-introduced* between two consecutive revisions. Eventually, we crafted a dataset of **3,451** static code warnings and their evolution status for both manual analysis and future evaluation.

After analyzing the limitations of the SOTA solution (**RQ2**), we propose StaticTracker to address the uncovered limitations. StaticTracker leverages refactoring information and the Hungarian algorithm [12] to significantly improve the tracking precision. More importantly, we designed a heuristic-based algorithm in StaticTracker that can effectively decide whether a removed static warning is due to fix ($removed_{fix}$) or non-fix ($removed_{non\text{-}fix}$). The SOTA solution detects removed warnings and does not further categorize them as fix or non-fix.

To answer **RQ3**, we first (RQ3.1) performed a comparative evaluation between StaticTracker and the SOTA approach. Our evaluation of the collected 3,451 warnings shows that StaticTracker provides a significant improvement over the SOTA solution, i.e., the tracking precision improves from 64.4% to 90.3% for the tracking precision. In RQ3.2, we conducted an independent evaluation of the performance of StaticTracker on a set of 2,014 commits. The evaluation of RQ3.2 shows that StaticTracker achieves a precision of 90.2% in categorizing $removed_{fix}$, $removed_{non\text{-}fix}$, and *newly-introduced* warnings. Particularly, it achieves a precision of 69.9% in identifying removed warnings that are fixed by developers.

In summary, this paper makes the following contributions:

- We collected and manually labeled a dataset of 3,451 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 four real-world open-source software projects (*JClouds*, *Kafka*, *Guava*, and *Spring-boot*).
- We examined the SOTA solution in tracking the evolution of static code warnings in terms of tracking accuracy based on the collected dataset. Our investigation shows that the SOTA solution achieves inadequate results.
- We performed a manual analysis to uncover the inaccuracies and the reasons behind the low accuracy of the SOTA solution. Our findings offer empirical evidence to further improve the tracking of static code warnings in the development history.
- We proposed a better approach StaticTracker to tracking static code warnings in development history. In addition to a much higher tracking precision, StaticTracker includes a heuristic-based algorithm to further categorize fixed and non-fixed warnings among the removed warnings. The evaluation based on the crafted dataset shows that StaticTracker can significantly improve tracking precision.

Artifact. We provide a replication package [1]. Our replica-

1. https://doi.org/10.5281/zenodo.6549386

tion package includes the implementations of the SOTA approach and StaticTracker, as well as our manually labeled ground-truth data for future evaluations.

Paper organization. The rest of the paper is organized as follows. Section 2 describes the motivation and the background, i.e., the relevant knowledge on static code warnings and how the SOTA approach works to track the static code warnings in development history. Section 3 illustrates the process and results of our manual analysis to understand the problems of the SOTA solution, including how the dataset is crafted and what are the insufficiencies of the SOTA solution. Section 4 shows the details of **StaticTracker**, and its evaluation is shown in 5. Section 6 describes the threats to validity. Section 7 lists the related work, and Section 8 concludes this study and proposes future works.

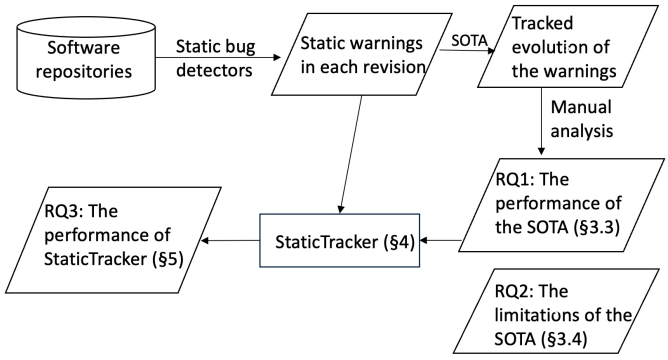


Fig. 1: An overview of our study.

2 Motivating Examples and Background

In this section, we formulate the problem we aim to solve, which is to provide a better approach to tracking the static code warnings in development history. Also, we provide background knowledge on the basics of static code warnings and how the state-of-the-art (SOTA) solution proposed by Avgustinov et al. [13] is used for tracking the evolution of static code warnings.

2.1 Challenges of tracking static code warnings in development history

Static bug detectors report a list of static code warnings given one version of a software system (i.e., one revision). A static code warning is subject to change as code evolves. Similar to tracking code changes, tracking the evolution of static code warnings in development history is based on comparing the generated reports from every two consecutive revisions. Applying a tracking solution, such as the SOTA approach, finds mappings of static code warnings between every two executive revisions and categorizes each static code warning to one of the four following statuses, i.e., **removed$_{\text{fix}}$**, **removed$_{\text{non-fix}}$**, **newly-introduced**, and **persistent**. Note that in the rest of this paper, we use **removed** to denote the combination of **removed$_{\text{fix}}$** and **removed$_{\text{non-fix}}$**.

- **Removed$_{\text{fix}}$**: A warning from the pre-commit revision is removed in the post-commit revision due to being fixed by developers.

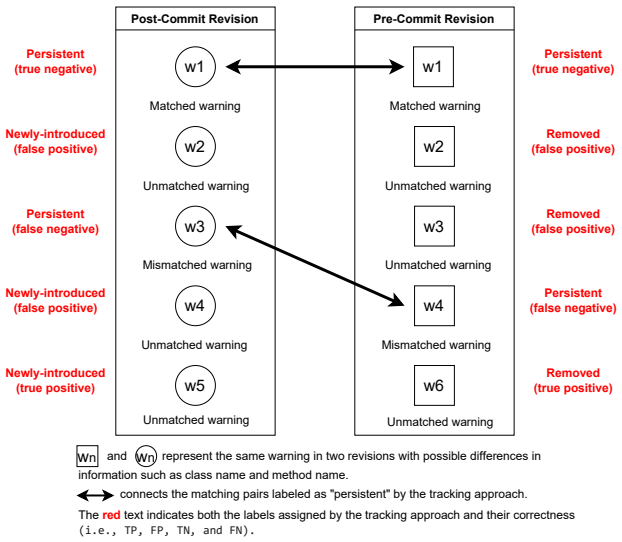


Fig. 2: An example to show how the SOTA approach may produce false positives and false negatives due to incorrect mappings. Note that the SOTA approach only reports the combined status *removed* rather than *removed$_{fix}$* and *removed$_{non\text{-}fix}$* separately.

- **Removed$_{\text{non-fix}}$**: A warning in the pre-commit revision is removed in the post-commit revision due to a non-fix reason, such as code deletion.
- **Newly-introduced**: A warning is not in the pre-commit revision, and later is introduced in the post-commit revision.
- **Persistent**: A warning does not change between the pre-commit and post-commit revision.

A good tracking mechanism should precisely label each static code warning as either persistent, removed$_{\text{non-fix}}$, removed$_{\text{fix}}$ or newly-introduced. In particular, it requires that all the mappings of static code warnings across two executive revisions are correctly established despite that the information of static code warnings used for mapping may be modified as code evolves. For example, the file name and code line number of the same warning may differ in consecutive revisions, which is challenging to find a correct mapping. Various types of software maintenance efforts contribute to making 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 (such as start line, end line, or class name) of static code warnings, which makes the tracking tool map warnings incorrectly due to the different metadata of a warning between two revision.

We use a simplified example (Figure 2) to explain the challenges of tracking the static warnings between two consecutive revisions. For easy understanding, we use three statuses only, i.e., persistent, removed (including removed$_{\text{fix}}$ and removed$_{\text{non-fix}}$), and newly-introduced. Given one commit, the left block represents all the warnings from the post-commit revision, and the right one is the pre-commit revision. Between the two revisions, the correct labels of the static warnings are as follows: persistent (`w1`, `w2`, `w3` and

w4), removed (w6), and newly-introduced (w5). Applying a tracking approach (i.e., the SOTA approach) establishes the mappings (shown as double arrow lines in Figure 2). However, the established mappings are erroneous due to the limitations of the tracking approach, and result incorrect labels.

For example, w2 warnings from the two revisions are not correctly mapped by the SOTA approach due to code changes. As a result, w2 is labeled as newly-introduced in the post-commit revision, while the correct label should be persistent. Additionally, w3 of the post-commit revision is incorrectly mapped with w4 of the pre-commit revision. After the mappings are established, these warnings are classified into three statuses. w2, w3 and w6 in the pre-commit revision are labeled as *removed*. However, only one (i.e., w6) is correctly labeled among them, so it is a true positive. The others (w2 and w3) are false positives of the tracking solution. Similarly, w2, w4 and w5 in the post-commit revision are labeled as *newly-introduced*, but only w5 is a true positive. w2 and w4 are false positives. The matched pairs are labeled as *persistent* by the tracking approach. Both w1 warnings in the post-commit and pre-commit revision are true negatives since they are mapped correctly. However, the labels of w3 of the post-commit revision and w4 of the pre-commit revision are matched incorrectly, so they are false negatives.

Evaluation Metrics. We define a metric, namely precision, to evaluate the effectiveness of a tracking approach. For each status, we identify true positives ($TP_{[status]}$), false positives ($FP_{[status]}$), true negatives ($FN_{[status]}$), and false negatives ($FN_{[status]}$). The precision is then calculated as $TP_{[status]}/(TP_{[status]} + FP_{[status]})$. Below we give examples of these definitions using the status removed$_{fix}$, fix for short.

- **TP$_{fix}$**: A removed warning is identified as a fix by the tracking approach correctly.
- **FP$_{fix}$**: A removed warning is identified as a fix by the tracking approach incorrectly.
- **TN$_{fix}$**: A removed warning is identified as a non-fix by the tracking approach correctly.
- **FN$_{fix}$**: A removed warning is identified as a non-fix by the tracking approach incorrectly.

2.2 The Metadata of Static Code Warnings

Static bug detectors often represent detected static code warnings using metadata to help developers locate static warnings. Previous studies [11] [7] utilize the metadata information that static bug detectors provide for each warning to track the evolution of the detected static code warnings.

Figure 3 provides an example of a 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.

2.3 The state-of-the-art (SOTA) solution

Avgustinov et al. [11] 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 SOTA solution. The overall structure of the SOTA 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.

```
<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>
```

Fig. 3: An example of the representation of one static code warning from Spotbugs. Note that the representation has been simplified to only show the information used by the SOTA matching approach.

Algorithm 1: The basic algorithm of the SOTA 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 removed warings, $W_{removed}$; The set of newly-introduced warings, $W_{newly-introduced}$; The set of matched pairs, $MatchedPairs$;

```
for each w_i in W_p do
    for each w_j in W_c do
        if source file of w_i is not a changed file then
            take ExactMatching(w_i, w_j);
        else
            take ExactMatching(w_i, w_j);
            take LocationMatching(w_i, w_j);
            take SnippetMatching(w_i, w_j);
            take HashMatching(w_i, w_j);
            if w_i is matched up by any one of the four
              matching strategies then
                make a MatchedPair(w_i, w_j);
                remove w_j from W_c ;
W_newly-introduced = W_c − MatchedPairs;
```

Algorithm 1 illustrates how the SOTA 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 SOTA 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

the location-based matching fails, the SOTA 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 SOTA 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 removed, and the ones in the post-commit revision are considered as newly-introduced.

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).

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 [14] [15] 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.

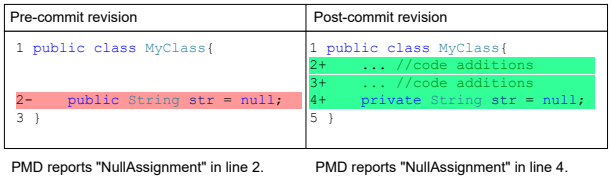


Fig. 4: An example to show how the location-based matching works to match the two "NullAssignment" despite the different line numbers.

As an example, Figure 4 shows a diff mapping. The numbers on the left hand are the line numbers in the pre-commit revision. The numbers on the right are the line numbers in the post-commit revision. There is a PMD warning ('NullAssignment') reported in the pre-commit revision (line 2) and line 4 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 1) and the warning (line 2) is 1 for the pre-commit revision and 3 (line 1 and line 4) for the post-commit revision. Then, the differences between the two offsets are calculated. In this example, the difference is smaller than 3, so location-based matching will match the two warnings.

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). In comparison to location-based matching, snippet-based matching relies purely on code snippets, resulting in more precise matches. However, it has the disadvantage of being unable to match the warning when its code snippet has changed between revisions. For example, in Figure 4, there is a change from 'public' to 'private' on line 2 of the pre-commit revision and line 4 of the post-commit revision. Snippet-based matching will be failed to work in this case. In contrast, location-based matching can handle this scenario by disregarding the code snippet and attempting to locate near warnings based on the diffs. As long as the offset between the two warnings does not exceed a specific threshold, location-based matching can successfully match them.

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 the 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 the hash values of the top N or latter tokens of two warnings are identical, i.e., $h(W^{post}_{topN}) = h(W^{pre}_{topN})$ or $h(W^{post}_{latter}) = h(W^{pre}_{latter})$, they will be considered as a matched pair by the hash-based matching.

Limitations of the SOTA approach. As we mentioned in Section 2.1, the metadata for the same warning may change in software evolution, while the matching strategies of the SOTA approach utilize the metadata in the matching process, which produces false positives of the tracked warnings from the SOTA approach. In this study, we improve the tracking approach to minimize the impact of the metadata changes of warnings in software evolution by employing three improvements.

3 Examining the Performance of the SOTA Approach

In this section, we describe how we investigated the performance of the SOTA approach in terms of the tracking accuracy, and answer **RQ1** and **RQ2**. In particular, we first crafted a dataset of static code warnings and their evolution status (i.e., persistent, removed$_{\text{fix}}$, removed$_{\text{non-fix}}$, or newly-introduced) between two consecutive revisions. To craft this dataset, we re-implemented the SOTA approach, applied it to the development history of the studied open-source systems, and performed a manual analysis to determine

TABLE 1: The studied systems and development periods. The release marks the end date of our studied development period, and we include 18 months of 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	2,695	400	v2.3.6	5,931	6,918
Guava	2,112	2,000	v20.0	6,474	3,819

the evolution status of each sampled static code warning. Then we manually analyzed whether the SOTA approach correctly tracked each sampled static code warning and categorized the reasons behind the incorrect tracking.

3.1 Studied Subjects

Static bug detectors. We include two static bug detectors, i.e., *PMD* and *Spotbugs*, both of which are widely used in prior studies and adopted in practice. A study [16] compared six static analysis tools. Among them, PMD and Findbugs achieved promising precision (52% and 57%, respectively). Thus we adopt PMD and Spotbugs in this paper. 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.

Analyzed open-source systems. Our study includes four Java open-source systems, *JClouds*, *Kafka*, *Spring-boot*, and *Guava*. They are four popular Java projects in different areas. *Spring-boot* is a framework for web applications, *Kafka* is a distributed system to handle streaming data, *Guava* is a core Java library for Google, and *JCloud* is a cloud toolkit for the Java platform. Projects with different areas may be collected different static warnings. The four projects are used to summarize the insufficiencies of the SOTA approach and provide reasons for the introduction of false positives. Then, they are used to evaluate StaticTracker and compare StaticTracker with the SOTA approach. 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., 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 years 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 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 Crafting the Dataset with Manual Labeling

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 [17]. 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 *removed* or *newly-introduced*.

Fourth, we choose to label all the warnings per sampled commit for constructing a ground-truth dataset, instead of the alternative, which is to sample warnings per commit and to include a larger set of commits, due to the inherent challenge in the mapping problem: one incorrect mapping may impact others, if only including one, we may only observe part of the impact, i.e., "the tip of the iceberg" as illustrated in Figure 2. Last, we have a decent confidence in the performance of the SOTA approach from its design. For example, we find that the majority of the established mappings by the SOTA approach are 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 decided to craft the dataset based on the tracking results of the SOTA approach and set our priorities on removed and newly-introduced static code warnings. We apply the static bug detectors on a total of 4,700 commits in the four projects. We re-implemented the SOTA approach based on the released artifact by Liu et al. [7]. Moreover, since Liu's work focuses exclusively on Spotbugs, we had to implement the SOTA approach for PMD. Then, we applied the SOTA 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 labeling following the steps below:

1) For *JClouds_Spotbugs*, *JClouds_PMD*, *Spring-boot_Spotbugs*, and *Spring-boot_PMD* we include all the static warnings labeled as removed by the SOTA approach.
2) Since there are many (i.e., 2,038 and 1,359) *removed* static warnings in *Kafka_PMD* and *Kafka_Spotbugs*, we took a statistically significant (95%±5%) sample, i.e., 326 and 301 *removed* static code warnings for both of them. Using *Kafka_PMD* as an example, we pursued the sampling process by firstly getting an estimation of the sample size, i.e., 323 warnings, then starting to randomly select one commit from the 436 *Kafka* commits with at least one removed warning, until we collected more than 323 warnings. In the end, we collected 326 removed warnings from 53 commits in *Kafka_PMD*.
3) In *Kafka_Spotbugs*, *Kafka_PMD*, *JClouds_Spotbugs* and *JClouds_PMD*, there exist a large number of *newly-introduced* code warnings. Hence, we took a statis-

TABLE 2: A summary of how we collect the static code warnings based on the results of the SOTA approach.

	SOTA: Removed		SOTA: Newly-Introduced	
	# Commits	# Warnings	# Commits	# Warnings
PMD				
JClouds	57	279	19	155
Kafka	53	326	14	255
Spring-boot	59	218	59	189
Guava	41	296	41	188
Spotbugs				
JClouds	23	104	5	78
Kafka	26	301	9	216
Spring-boot	17	193	17	160
Guava	44	289	44	204
Total	320	2,006	208	1,445

tically 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 a total of 704 warnings (i.e., in 47 commits) labeled as 'newly-introduced' by the SOTA approach.

4) We took the same sample strategy on *Guava_Spotbugs* and *Guava_PMD*, a statistically significant (95%±5%) sample of removed warnings with all newly-introduced warnings of their commits, i.e., 41 commits with 296 removed warnings and 188 newly-introduced warnings in *Guava_PMD*, and 44 commits with 289 removed warnings and 204 newly-introduced warnings in *Guava_Spotbugs*.

Table 2 summarizes the breakdown of the static code warnings we collected following the above-mentioned steps. Note that in Table 2, the evolution statuses such as *removed* and *newly-introduced* are labeled by the SOTA approach, which might be incorrect. We performed a manual analysis to reveal the true evolution status of each warning. The SOTA approach cannot detect whether a warning is removed for fix or non-fix reasons. Since one of our goals is to identify the warnings that are fixed by developers, we further manually categorize to $\text{removed}_{\text{fix}}$ or $\text{removed}_{\text{non-fix}}$. Table 3 summarizes the ground-truth evolution status of the dataset. In total, our dataset contains 3,451 static code warnings and their true evolution status in the development history of the four projects: 34.0% are persistent, 4% are $\text{removed}_{\text{fix}}$, 33.1% are $\text{removed}_{\text{non-fix}}$, and 28.9% 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 involve 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 2 and Table 3 regarding the distribution of the evolution statuses. That is because the SOTA approach produces a non-negligible number of incorrect results. We present more details on the inaccuracies in Section 3.4.

TABLE 3: A data set of 3,451 static code warnings with manually-labeled evolution statuses.

	Persistent	$Removed_{fix}$	$Removed_{non\text{-}fix}$	*Newly-Introduced*
PMD				
JClouds	232	23	77	102
Kafka	164	26	158	233
Spring-boot	159	9	127	112
Guava	257	4	163	60
Spotbugs				
JClouds	32	29	56	65
Kafka	205	15	174	123
Spring-boot	12	1	186	154
Guava	113	31	199	150
Total	1,174	138	1,140	999

TABLE 4: The performance of the SOTA approach. Note that 'removed' stands for both $\text{removed}_{\text{fix}}$ and $\text{removed}_{\text{non-fix}}$ as the SOTA approach detects three statuses only: removed, persistent, and newly-introduced.

	SOTA: Removed			SOTA: Newly-Introduced		
	TP	FP	Precision	TP	FP	Precision
PMD						
JClouds	100	179	35.8%	102	53	65.8%
Kafka	184	142	56.4%	233	22	91.4%
Spring-boot	136	82	62.4%	112	77	59.3%
Guava	167	129	56.4%	60	128	31.9%
Spotbugs						
JClouds	85	19	81.7%	65	13	83.3%
Kafka	189	112	62.8%	123	93	56.9%
Spring-boot	187	6	96.9%	154	6	96.3%
Guava	230	59	79.6%	150	54	73.5%
Total	1,278	728	63.7%	999	446	69.1%

3.3 Evaluating the Performance of the SOTA Approach

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

We manually investigated the sampled 3,451 static code warnings. Table 4 summarizes the performance of the SOTA approach on the crafted dataset. Among the 2,006 warnings that are determined as *removed* by the SOTA approach, only 1,278 (63.7%) are *truly removed*. Among the 1,445 warnings that are determined as *newly-introduced*, only 999 (69.1%) are *actually* newly-introduced. The false positives (FPs) of "removed" and "newly-introduced" in Table 4 are the warnings with a ground-truth status of *persistent*. In short, the precision in tracking both removed and newly-introduced warnings of the SOTA approach on the collected dataset is only 66.0% (2,277/3,451). Our evaluation of the SOTA approach reveals that tracking the evolution of static code warnings over the development period is not that straightforward. The low precision of the SOTA approach will negatively impact many downstream software engineering

TABLE 5: Causes of false positives by the SOTA approach.

Cause	Number
1. Code refactoring	437
2. Code shifting	366
3. Volatile class/method/variable names	86
4. Drastic and non-refactoring code changes	285
Total	1,174

tasks, such as mining fix patterns from software repositories or performing empirical studies on software quality.

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

We present a dataset of 3,451 static code warnings and their evolution statuses. The dataset is crafted with support from the SOTA approach. The tracking precision of the SOTA approach on the dataset is only 66.0%, and this tracking approach will impact many downstream software engineering tasks negatively.

3.4 Investigating the Inaccuracies of the SOTA Approach

RQ2. What are the limitations of the SOTA approach?

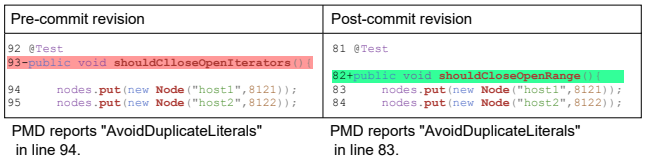


Fig. 5: An example of false positives due to method renaming.

Furthermore, we manually analyzed the insufficiencies of the SOTA approach, i.e., based on 1,174 false positives, and concluded into four categories as follows. Table 5 summarizes four causes of false positives in our dataset.

Code refactoring. We find that the SOTA approach cannot properly handle three common types of refactoring, namely class renaming, method renaming, and variable renaming. In particular, the first three matching strategies of the SOTA approach require the exact same class name, method name, and variable name, with a slight tolerance for line information. To tolerate minor differences in class/method/variable names (commonly caused by refactoring), the SOTA approach relies on the last matching strategy, namely the hash-based matching strategy. However, our experiment reveals that the hash-based matching strategy is sensitive to the regional code change, and fails to elegantly handle the refactoring changes since in reality, refactoring code changes are often combined with other code changes [18]. Note that we notice cases that are caused by more than one type of refactoring, e.g., one commit may contain both method renaming and variable renaming, which causes drastic changes in the metadata of warnings.

Figure 5 is a case of a false positive due to method renaming. In this example, PMD detects a warning reported as 'AvoidDuplicateLiterals' on line 94 of the pre-commit revision and line 83 of the post-commit revision. This warning indicates that the string 'host1' is used multiple times in this class file. Due to the metadata of the method name changes, the first three matching strategies cannot match the warnings in the method. Thus the SOTA approach relies on the 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. Except for method renaming, other common code refactorings, such as class renaming and variable renaming also affects the consistency of the metadata between revisions. In total, we find 437 false positives in this category.

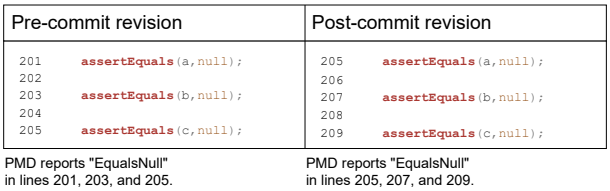


Fig. 6: An example of the false positives due to code shifting.

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 SOTA approach does not always handle the shifting well, and false positives will be produced. Totally, we find 366 false positives in this category.

Figure 6 shows an example of how code shifting may cause the SOTA approach to malfunction. This example contains three warnings of "EqualsNull" in lines 201, 203, and 205 in the pre-commit revision and lines 205, 207, and 209 in the post-commit revision. The task is a 3x3 mapping problem. When the SOTA approach tries to process this 3x3 problem, it first matches line 205 in the pre-commit revision with line 205 in the post-commit revision through the exact matching strategy (i.e., identical line number), because the highest priority of all the four matching strategies provided by the SOTA approach. This incorrect mapping (line 205 v.s., line 205) has a butterfly impact on the remaining mapping. For example, when the SOTA approach tries to find a mapping warning for line 201 in the pre-commit revision, it fails to match with line 205 in the post-commit revision since the latter has been matched with line 205 in the pre-commit revision. As a result, the SOTA approach produces false positives as lines 201 and 203 are labeled as removed.

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. 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 lines 201 and 203 from the pre-commit revision to be considered as resolved, and lines 207 and 209 from the post-commit revision to be considered as newly-introduced warnings, while they actually persist.

Volatile class/method/variable names. Even though there are 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 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

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

Fig. 7: An example of Scala code that has implicit code changes. The meta data of the relevant warning from the pre- and post-commit revisions are shown in Figure 8 and Figure 9.

```
<WarningInstance>
  <WarningType>SE_BAD_FIELD</WarningType>
  <Project>kafka</Project>
  <Class>AclCommand</Class>
  <Method></Method>
  <Field>opts$4</Field>
  <FilePath>kafka/admin/AclCommand.scala</FilePath>
  <StartLine>206</StartLine>
  <EndLine>206</EndLine>
</WarningInstance>
```

Fig. 8: The warning information from pre-commit revision

Scala code when anonymous classes and methods are used heavily. Then some persistent warnings are not matched correctly. Totally, we find 86 false positives in this category.

Figure 7 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 8 and Figure 9) and found that not only the line numbers are different, the variable names are also different (the differences are highlighted using blue lines in Figure 8 and Figure 9).

Drastic and non-refactoring code changes. In cases where the code change is significant, all the matching strategies applied by the SOTA approach may fail to function adequately. Such a scenario may arise, for instance, if the offset of the diffs surpasses the threshold, causing location-based matching to be failed. Similarly, the modified code snippet of the warnings makes snippet-based matching fail. The introduction of significant changes also poses a challenge to hash-based matching, as differing hash values are calculated between the two revisions.

We perform further manual analysis on the FPs of the crafted dataset, and identify four main causes behind the inaccuracies of the SOTA approach in tracking the evolution of static code warnings.

4 StaticTracker: A Better Approach to Track Static Warnings

Guided by our manual analysis results, we propose to improve the SOTA approach by better handling refactoring changes and revising a few key steps to improve the accuracy of irrelevant code changes. In particular, StaticTracker (as illustrated in Algorithm 2) reuses the three matching strategies of the SOTA approach (i.e., Exact matching, Location-based matching, and Snippet-based matching) and revises a few key steps to improve the inaccurate tracking. In addition, we develop an algorithm to further distinguish fixes and non-fixes from the removed warnings, i.e., line 17 in Algorithm 2.

```
<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>
```

Fig. 9: The warning information from post-commit revision

Algorithm 2: The algorithm of StaticTracker.

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

Output: $W_{removed\_fix}$ is the set of removed$_{\text{fix}}$ warnings; $W_{removed\_non\_fix}$ is the set of removed$_{\text{non-fix}}$ warnings; $W_{newly-introduced}$ is the set of newly-introduced warnings; $MatchedPairs$ is the set of matched pairs.

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 take $LocationMatching(w_i^{'}, w_j^{'})$;
13 **if** *there is any candidate from both approaches* **then**
14 $HMatrix[i][j] += 1$;
15 $MatchedPairs = Hungarian(HMatrix)$;
16 $W_{removed} = W_p - MatchedPairs$;
17 $W_{removed\_fix}, W_{removed\_non\_fix}$= identifyFixNonfixRemoval($W_{removed}$);
18 //This function is detailed in Algorithm 3
$W_{newly-introduced} = W_c - MatchedPairs$;

Improvement 1 - Including refactoring. We included the refactoring information to improve the tracking of static warnings using RefactoringMiner 2.0 [19]. RefactoringMiner 2.0 is the state-of-the-art tool to detect refactoring for Java language. RefactoringMiner 2.0 is shown to outperform RefDiff [20] and GumTreeDiff [21] with a precision of 99.6% and a recall of 94%. We first created a replica of w_i (namely $w_i^{'}$), which is from the pre-commit revision, and then modified the metadata of $w_i^{'}$ with the information from RefactoringMiner. For instance, if RefactoringMiner reveals that the class in w_j is a result of refactoring of "move and rename class", we modify the class name in $w_i^{'}$ with the one after the refactoring activity to keep the consistent metadata of the warning from two revisions. 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 hash-based matching. As of now, we include **22** types of refactoring that cause the modified metadata of warnings.

Improvement 2 - Deciding matched pairs using the Hungarian algorithm. Commonly, a warning of pre-commit revision may have more than one matched warning from post-commit. Thus it is a problem of which one should be matched up. In the SOTA 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 location-matching first and snippet-matching first. The order in the SOTA 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 6). 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 10. $w1_p$, $w2_p$ and $w3_p$ are the warnings from pre-commit 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 two (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 SOTA 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 10. The size of $HMatrix$ is (the number of W_p) $*$ (the number of W_c) and the values are zero 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 SOTA 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 StaticTracker. However, we keep it for unchanged files.

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.

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

Fig. 10: A simple example of Hungarian matrix.

Improvement 4 - An approach to identify the removed warnings that are fixed by developers. We further proposed a heuristic-based algorithm (described in Algorithm 3) that identifies fixes from removed warnings. StaticTracker applies GumTreeDiff [21] to extract the *Diff* based on Abstract Syntax Tree (AST) representation. Algorithm 3 takes a conservative way of identifying fixed warnings (line 8 and line 22) while identifying non-fix warnings proactively. For a given warning (w_i), the corresponding class, method, and field information are extracted (function `locate_context` in line 2). If any of the class, method, and field declarations are completely deleted, then the warning is deemed `non-fix` (line 4). When w_i is about the declaration of a source code entity (line 6), we expect that a fix would be about modifying the declaration, such as removing `synchronized` from the modifiers. Alternatively, line 11–28 analyzes the detailed changes in the commit when w_i is reported in a method (mth). If mth does not contain any code changes (line 13) or mth only contains code deletions (line 15), w_i is classified as non-fix. When w_i is about issues with a variable (line 19, a field name is reported in the metadata of w_i), our strategy is to identify whether there exists any modification on the reported field of w_i (line 21). When w_i is not about a particular field, our strategy is about trying to estimate a repair scope. If an estimated repair scope is not changed by the commit, i.e., no overlap between $Diffs$ and mth, our algorithm classifies this commit as non-fix proactively. By default, the repair scope is the method of the warning. For a few exceptional cases, our algorithm refines the repair scope further. In particular, such repair scope is estimated to include the range from the warning line (w_i.end) till the end of the method when w_i is a one-line warning. Then, we check whether there are any modifications in the repair scope. If there are no modifications, w_i is considered non-fix (line 28). Finally, the algorithm considers a fix for each of the remaining unlabeled warnings (line 30).

5 Evaluation of StaticTracker

RQ3. What is the performance of StaticTracker? We evaluated the performance of StaticTracker and set two sub-RQs for RQ3. The first one is a comparison evaluation between StaticTracker and the SOTA approach. We evaluated StaticTracker on the crafted dataset to show how much improvement StaticTracker has compared to the SOTA approach and answer **RQ3.1**. Furthermore, in **RQ3.2**, We re-sampled new

Algorithm 3: StaticTracker's algorithm to identify fix and non-fix warnings among removed warnings.

```
Input: The set of removed warnings, W_removed; The
       commit diffs computed by GumTree, Diff
Output: 1. The set of removed_non-fix warnings,
          W_non_fix;
        2. The set of removed_fix warnings, W_fix;
foreach w_i in W_removed do
    (cls, mth, field) = locate_context(w_i);
    if is_deleted(cls) || is_deleted(mth) || is_deleted(field)
     then
        W_non_fix.add(w_i);
        Next;
    if same_range(w_i, cls) || same_range(w_i, mth) ||
     same_range(w_i, field) then
        if is_declaration_modified(cls) ||
          is_declaration_modified(mth) ||
          is_declaration_modified(field) then
            W_fix.add(w_i);
        else
            W_non_fix.add(w_i);
    else if range(w_i) ∈ range(mth) then
        repair_scope = {range(mth)};
        diffs_repair_scope = Diff ∩ repair_scope;
        if diffs_repair_scope == ∅ then
            W_non_fix.add(w_i);
        else if all_deletions(diffs_mth) then
            W_non_fix.add(w_i);
        else
            if field != NULL then
                repair_scope.append(range(mth));
                if (exists field is modified by Diffs ∩
                  repair_scope) then
                    W_fix.add(w_i);
                else
                    W_non_fix.add(w_i);
            else if w_i.start == w_i.end then
                repair_scope.append({(w_i.end,
                  mth.end)});
                if Diff ∩ repair_scope == NULL then
                    W_non_fix.add(w_i);
    if w_i ∉ W_fix && w_i ∉ W_non_fix then
        W_fix.add(w_i);
```

commits and conducted an analysis of StaticTracker about how accurate it is on these commits.

RQ3.1. Can StaticTracker perform better than the SOTA approach?

We compared the SOTA approach with StaticTracker by running both approaches on the same commits we labeled before. 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 320 commits**, which is a superset of the 3,451 warnings in the manually-labeled dataset. The remaining warnings in the 320 commits, while not in our crafted dataset, have a pre-assumed label, "*persistent*". If StaticTracker changes the pre-assumed label of some warnings, then we manually examine the ground-truth labels of these warnings.

Table 6 shows the comparison results between the SOTA approach and our improved approach on the collected dataset of 3,451 static code warnings. Note that there are 3,451 static warnings from the SOTA approach. However, when we applied StaticTracker to the same dataset, we obtained only 2,463 removed and newly-introduced warnings, which means that the rest are identified by StaticTracker as persistent warnings. We categorized the 3,451 warnings into three categories (i.e., removed$_{\text{fix}}$, removed$_{\text{non-fix}}$, and newly-introduced) according to the labels by the SOTA approach for easy comparison. The SOTA approach does not detect if a removed warning is due to a fix. In order to ensure a fair and comprehensive comparison, we have integrated our fix-detecting algorithm (Algorithm 3) into the SOTA approach. The SOTA approach labels 226 fixes. Among them, 123 fixes are true positives. Therefore, the precision of the SOTA approach is 54.4% for detecting fixes. The StaticTracker achieved a precision of 72.5% (i.e., among 171 labeled fixes by StaticTracker, 124 are true positives). The evaluation shows that StaticTracker can significantly improve the tracking performance. Overall, for the 3,451 warnings, the SOTA approach has 1,227 warnings with wrong evolution statuses, i.e., the tracking precision is 64.4%. Compared to that, the precision of StaticTracker achieved 90.3%. **StaticTracker reduces the false positives by correctly labeling the *persistent* warnings, which are mistakenly labeled as *removed* or *newly-introduced* by the SOTA approach.**

StaticTracker is shown to effectively reduce false positives for the four causes listed from the SOTA approach in Table 5. Table 7 shows the breakdown of the left false positives by each cause after using StaticTracker on the *removed* warning dataset. One more category is named 'Fix-detection labels warnings incorrectly', which is caused by our proposed fix-detection approach.

> *StaticTracker outperforms the SOTA approach by reducing false positives significantly (i.e., from 1,227 to 239) and yields a precision of 90.3% in detecting evolution statuses.*

RQ3.2. How accurate is StaticTracker for tracking the evolution of static code warnings?

Apart from the SOTA approach, we also take a generalization evaluation of StaticTracker by a statistically significant (95%±5%) sample on commits for each project to answer this RQ. StaticTracker is applied to collect removed and newly-introduced warnings on sampled commits. 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 8 shows the results of StaticTracker for RQ3.2. Note that there are many commits that have no removed or newly-introduced warnings in this evaluation. In other words, the code changes of many commits are too small to change the status of all static warnings. We sampled 2,014 commits. Our approach correctly identified 51 removed$_{\text{fix}}$, 339 removed$_{\text{non-fix}}$ and 794 newly-introduced warnings on these commits. Overall, StaticTracker has a great performance in detecting evolution statuses with a

TABLE 6: Performance comparison between the SOTA approach and StaticTracker. Prec. is short for precision. A higher precision means fewer false positives and a better tracking performance.

	Removed$_{fix}$		Removed$_{non\text{-}fix}$		Newly-Introduced		**Total** removed$_{fix}$, removed$_{non\text{-}fix}$, and newly-introduced	
	Prec. (SOTA)	Prec. (StaticTracker)	Prec. (SOTA)	Prec. (StaticTracker)	Prec. (SOTA)	Prec. (StaticTracker)	Prec. (SOTA)	Prec. (StaticTracker)
PMD								
JClouds	48.8% (21/43)	80.8% (21/26)	30.9% (73/236)	83.9% (73/87)	65.8% (102/155)	97.2% (104/107)	45.2% (196/434)	90.0% (198/220)
Kafka	50.0% (25/50)	58.1% (25/43)	51.4% (142/276)	90.0% (144/160)	91.4% (233/255)	98.3% (233/237)	68.8% (400/581)	91.4% (402/440)
Spring-boot	56.2% (9/16)	69.2% (9/13)	60.9% (123/202)	85.4% (123/144)	59.3% (112/189)	86.7% (111/128)	60.0% (244/407)	85.3% (243/285)
Guava	17.6% (3/17)	50.0% (3/6)	57.7% (161/279)	89.0% (161/181)	31.9% (60/188)	75.9% (60/79)	46.3% (224/484)	84.2% (224/266)
Spotbugs								
JClouds	70.3% (26/37)	89.7% (26/29)	80.6% (54/67)	94.7% (54/57)	83.3% (65/78)	100.0% (65/65)	79.7% (145/182)	96.0% (145/151)
Kafka	62.5% (10/16)	66.7% (10/15)	59.3% (169/285)	85.4% (169/198)	56.9% (123/216)	83.8% (119/142)	58.4% (302/517)	83.9% (298/355)
Spring-boot	50.0% (1/2)	100.0% (1/1)	97.4% (186/191)	100.0% (186/186)	96.2% (154/160)	100.0% (154/154)	96.6% (341/353)	100.0% (341/341)
Guava	62.2% (28/45)	76.3% (29/38)	79.5% (194/244)	93.7% (194/207)	73.5% (150/204)	93.8% (150/160)	75.5% (372/493)	92.1% (373/405)
Total	54.4% (123/226)	**72.5% (124/171)**	61.9% (1102/1,780)	**90.5% (1,104/1,220)**	69.1% (999/1,445)	**92.9% (996/1,072)**	64.4% (2,224/3,451)	**90.3% (2,224/2,463)**

TABLE 7: A breakdown of StaticTracker's false positives.

Cause	Number
1. Code refactoring	78
2. Code shifting	32
3. Volatile class/method/variable names	8
4. Drastic and non-refactoring code changes	68
5. Fail to differentiate removed$_{fix}$ and removed$_{non\text{-}fix}$ warnings	53
Total	239

TABLE 8: The performance of StaticTracker (RQ3.2).

	# Commits	Prec. (Removed$_{fix}$)	Prec. (Removed$_{non\text{-}fix}$)	Prec. (Newly-Intr.)
PMD				
JClouds	169	82.4% (14/17)	82.4% (61/74)	96.0% (144/150)
Kafka	322	26.7% (4/15)	91.0% (101/111)	94.8% (145/153)
Spring-boot	194	100.0% (2/2)	100.0% (4/4)	100.0% (17/17)
Guava	322	50.0% (1/2)	68.4% (26/38)	91.7% (100/109)
Spotbugs				
JClouds	169	90.9% (20/22)	83.3% (15/18)	100.0% (106/106)
Kafka	322	50.0% (4/8)	75.8% (69/91)	90.3% (168/186)
Spring-boot	194	NA. (0/0)	100.0% (20/20)	100.0% (10/10))
Guava	322	85.7% (6/7)	93.5% (43/46)	98.1% (104/106)
Total/Avg.	2,014	69.9% (51/73)	84.3% (339/402)	94.9% (794/837)

tracking precision of 90.2% (1,184/1,312). For detecting removed$_{fix}$ warnings, StaticTracker yields a precision of 69.9%.

By conducting the generalization analysis of StaticTracker, results show that StaticTracker achieves a tracking precision of 90.2% in identifying evolution status of warnings.

Discussions

We provide further discussions on 1) the false positives of our proposed approach StaticTracker and the reasons behind such false positives; 2) an ablation analysis on the three improvements we designed for StaticTracker; and 3) correlations between the warning types and the number of warnings tracked correctly or not.

Analysis on the false positives of StaticTracker. We conducted a detailed investigation into the false positives in StaticTracker and discussed them. To achieve this, we sampled 75 warnings from the total of 239 warnings with a statistically significant (95%±10%) sample and manually analyze each one to uncover its root causes. We summarize the uncovered causes below.

- *Undetected refactoring (32/75).* Even with the state-of-the-art refactoring detection tool (i.e., RefactoringMiner), there exist refactoring changes that are not detected. This contributes to almost half of the false positives of StaticTracker. Figure 11 shows an example of unmatched static warnings due to undetected refactoring, i.e., the same warning of 'NullAssignment' (line 187 in the pre-commit revision and line 51 in the post-commit revision). The commit includes one class renaming and one method renaming and the latter (from `'ThrowingFuture'` to `'UncheckedThrowingFuture'`) is not detected by RefactoringMiner. As a result, StaticTracker fails to match the same warning from the two consecutive revisions and labels the warning incorrectly as removed and newly-introduced, respectively.

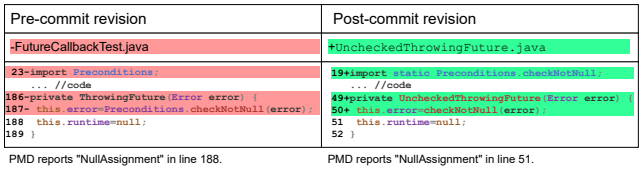


Fig. 11: An example of the false positive in StaticTracker due to undetected method renaming by RefactoringMiner. The commit is ba2024d from Guava.

- *Superseded by a new warning (15/75).* We find for 15 cases, the warnings in the pre- and post-commit revisions are highly similar, i.e., one is superseded by the other as code evolves. Figure 12 shows an example of a warning 'AvoidDuplicateLiterals' detected by PMD. The string "key cannot be null" is used multiple times and PMD reports one warning for the multiple uses of the duplicate literal. This warning is then superseded by a highly similar warning when the new code contains another use of the duplicate literal. The warning in the post-commit revision contains a different location, i.e., from line 191 in the method *get* to line 173 in the method *delete*. As a result, StaticTracker fails to match the two correctly. Another type of example is about one warning is superseded by another warning of a different but related warning type. Detecting the two warning types share some similarities. Some code changes irrelevant to the scope of the reported warning may easily alter the detection from one type to another type. For example, after Guava-5562218, one warning of 'SE_BAD_FIELD_INNER_CLASS' is changed to 'SE_INNER_CLASS' due to a code change to the outer class, which is not on the reported code scope (i.e., the inner class).

- *Limitations of using Hungarian algorithm (4/75).* Usually, the Hungarian algorithm is effective in establishing matching pairs. However, certain situations may arise wherein a warning from the pre-commit revision has two possible candidates from the post-commit revision, and the two

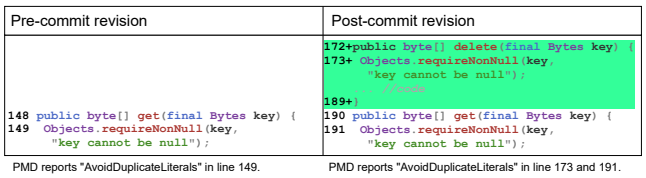


Fig. 12: An example of a warning in the pre-commit revision superseded by a new warning in the post-commit revision. The commit is 3c46b56 from Kafka.

possible candidates have equivalent weights. For such cases, the Hungarian algorithm may fail to accurately determine the matched pair, resulting in a mismatch.

- *Drastic code changes (16/75)*. Metadata of static warnings used for matching is changed significantly by drastic code changes. The two matching strategies (i.e., location and snippet matching) are designed to effectively match most persistent warnings by tolerating non-drastic code changes. When there is a significant change in the class file, both strategies may fail to match certain warnings, resulting in false positives.

- *Limitations of the fix-detection approach (8/75)*. Our proposed fix-detection approach cannot detect every fix and non-fix case correctly. Among the 75 false positives, eight cases are identified with a wrong status between $\text{removed}_{\text{fix}}$ and $\text{removed}_{\text{non-fix}}$.

Ablation analysis. We proposed three improvements in StaticTracker to tackle the limitations of the SOTA approach: 1) Handling volatile identifiers (VI), 2) Detecting refactoring using RefactoringMiner (RM), and 3) Matching using the Hungarian algorithm (HA).

To evaluate the impact of the improvements (individual and combined), we performed an ablation analysis. Since VI "correct" the metadata of static code warnings and RM and HA improve the matching process, we consider VI is fundamental and select combinations on top of VI: 1) VI, 2) VI+RM, and 3) VI+HA. Table 9 presents the false positive rates of these different combinations of the three improvements. Our findings demonstrate that handling volatile identifiers results in a slight increase to the tracking precision, i.e., from 64.4% to 67.9%. Additionally, both VI+RM and VI+HA approaches have similar performance, with precisions of 75.6% and 76.6%, respectively. Notably, the combination of all three improvements in StaticTracker resulted in a significant improvement of the tracking process, at 90.3% precision. In short, the proposed three improvements complement each other, and the combination of all three significantly outperforms the other combinations.

TABLE 9: Ablation analysis on the three improvements StaticTracker has over the SOTA approach.

Approach	Precision
Baseline (SOTA)	64.4% (2,224/3,451)
Baseline + **VI**	67.9% (2,211/3,256)
Baseline + **VI**+**RM**	75.6% (2,210/2,923)
Baseline + **VI**+**HA**	76.6% (2,220/2,900)
Baseline + **VI**+**RM**+**HA** (StaticTracker)	90.3% (2,224/2,463)

Correlation between the performance of StaticTracker and the types of static warnings. We analyzed whether there exist significant different performance improvements of StaticTracker over the SOTA approach on each warning type involved. Particularly, this evaluation involves 32 PMD warning types and 111 Spotbug warning types. We performed Fisher's exact test on the pair of true positives and false positives per warning type between the two approaches (StaticTracker and SOTA). We find that for seven PMD warning types and six Spotbugs warning types, the differences between StaticTracker and Spotbugs are statistically significant, i.e., the improvement of StaticTracker on these warning types is significant. Table 10 describes the detailed results.

False Negatives. The false negatives in our context are the warnings deemed as 'persistent' by StaticTracker have a ground-truth label of either $\text{removed}_{\text{fix}}$, $\text{removed}_{\text{non-fix}}$, or newly-introduced. It is extremely time-consuming and very challenging to identify the false positives in our context due to the tremendous number of *persistent* warnings (e.g., up to thousands of warnings between two consecutive versions). We believe both the SOTA and StaticTracker have low numbers of false negatives since both have highly strict rules to decide persistent warnings, i.e., the metadata of two warnings have to be highly similar to be considered for matched warnings between two consecutive versions. To provide evidence, we took a statistically significant sampling (95%±5%) of 384 persistent warnings from a total of over six million persistent warnings. Subsequently, we analyzed the sampled warnings and confirmed that all of them were true negatives, i.e., no false negatives are identified from this statistically significant sample. This demonstrates that StaticTracker likely has very high recall in identifying evolving warnings.

TABLE 10: Correlation between the performance of StaticTracker in comparison with the SOTA approach and warning types.

	SOTA		StaticTracker		
	TP	FP	TP	FP	p-value
PMD					
BeanMembersShouldSerialize	444	276	443	35	4.2e-37
DetachedTestCase	5	17	7	0	5.0e-4
AvoidDuplicateLiterals	121	220	121	50	5.5e-14
DataflowAnomalyAnalysis	301	164	300	41	2.1e-14
AvoidFieldNameMatchingMethodName	64	62	66	2	1.7e-12
NullAssignment	16	17	17	4	0.02
AvoidCatchingNPE	3	26	3	0	0.004
Spotbugs					
DLS_DEAD_LOCAL_STORE	54	21	53	4	0.003
NP_PARAMETER_MUST_BE_NONNULL_BUT_MARKED_AS_NULLABLE	49	10	49	2	0.03
SE_BAD_FIELD	102	86	102	18	1.6e-8
SIC_INNER_SHOULD_BE_STATIC_ANON	121	77	122	10	3.1e-11
NP_ALWAYS_NULL	38	60	37	8	1.6e-6
NP_LOAD_OF_KNOWN_NULL_VALUE	26	32	25	8	0.004

6 Threats to Validity

In this section, we describe threats to external and internal validity.

6.1 External Validity

In this paper, 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., the use of class/method/variable names and code ranges for matching purposes. The

improvement of StaticTracker may not be generalizable to a static bug detector with a 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 SOTA approach is based on four open-source projects. To increase the diversity, we analyzed a reasonable number of commits in the four projects. In general, we find that the evolution details that make the SOTA approach malfunction are consistent in our collected dataset. In the generalization analysis, we sampled commits to evaluate StaticTracker, but many commits have no disappeared and newly-introduced warnings, which means that all warnings from these commits are labeled as persistent warnings by StaticTracker.

6.2 Internal Validity

When it comes to manually labeling the dataset, human errors are inevitable. We tried to reduce human errors by having two people annotating the dataset and resolving 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 SOTA approach (i.e., most of the established mappings are correct) and also our priorities, which is to focus on the disappeared and newly-introduced warnings.

7 Related Work

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 [22], 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 inspired many researchers to work on improving *SZZ*. For example, a study [23] 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 [24] 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 [25] 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 [26] combined the tool with AST for the detection of code patterns across multiple versions. There is a study [27] 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 about a static warning. Dong-Jae et al. [28] conducted an empirical study on the evolution of annotation changes and created a taxonomy to uncover what annotation changes have and the motivation of annotation changes. In addition, Felix et al. [29] 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% of 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. [13] 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 disappeared. In particular, Sunghun et al. [30] proposed an algorithm to automatically identify bug-introducing changes with high accuracy by combining the annotation graphs and ignoring non-semantic source code changes. Results show that their algorithm outperforms the *SZZ*. Cathal and Leon [10] conducted an empirical study to investigate the relation between static warnings and actual faults. More recently, Avgustinov et al. [11] 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 SOTA approach is still lacking in the field. In this study, we manually crafted a dataset of 3,451 static code warnings and their evolution status from four real-world open-source systems and used it to identify potential improvements in the SOTA approach.

Empirical studies on static bug detectors. Researchers have been working on understanding and improving the utilization challenge of static bug detectors. Johnson et al. [17] 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. [31] performed a large-scale study to understand the current status of using static bug detectors in open-source systems, e.g., whether or not used, and what running configurations are used. Wang et al. [32] aimed to find whether there is a golden feature to indicate actionable static warnings. Additionally, a survey was conducted by Muske et al. [33] 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. [34] 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. [35] found that the issues by static bug detectors are much more related to refactoring than defects. Habib et al. [36] 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 [37] evaluated the degree of correlation between defects and warnings on the evolution of projects. Tomassi et al. [38] 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 Spotbugs. Trautsch et al. [39] conducted a longitudinal study on static analysis warning trends. They found that the quality of code with regard 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 evolve. Also, our study includes a manual analysis of a non-trivial dataset of static code warnings for the purpose of improving the tracking precision, which is not covered by prior work.

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 in concluding fix patterns. Kui et al. [7] mine the fix patterns on static code warnings from the software repository, and the SOTA approach was applied in their research. However, they did not conduct an evaluation on the approach about how accurate the SOTA approach performs. A study [8] proposed a novel solution to automatically generate code fixing patches for static code warnings via learning from fixing examples. Another recent work [40] 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 [4] 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 are removed during code non-fix changes. Ted et al. [41] 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. [3], 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 static warnings in the evolution of software projects. The other work [42] 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 attention that programmers paid to those static warning. Higo et al. [43] 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.

8 Conclusions

Tracking the evolution of static code warnings across software development history becomes a vital question due to the increasing interest in further utilizing static bug detectors by integrating them into developers' workflow. Also, such tracking is widely used in many downstream software engineering tasks.

This study presents a careful investigation of the performance of the state-of-the-art approach in tracking static code warnings. In particular, a dataset of 3,451 static code warnings for four open-source projects and their evolution status is crafted through manual labeling. Further, we summarize the six causes that introduce false positives for the SOTA approach. To address the false positives, this paper presents an improved approach **StaticTracker**.

Last, we perform comparative and generalization evaluations. Results show that our improved approach outperforms the SOTA approach significantly (i.e., the tracking precision from 64.4% to 90.3%).

References

[1] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at google," *Commun. ACM*, vol. 61, no. 4, p. 58–66, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3188720

[2] (2019) Spotbugs latest version. [Online]. Available: http://spotbugs.readthedocs.io

[3] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: Improving actionable alert ranking," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 152–161. [Online]. Available: https://doi.org/10.1145/2597073.2597100

[4] S. Kim and M. D. Ernst, "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*, ser. ESEC-FSE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 45–54. [Online]. Available: https://doi.org/10.1145/1287624.1287633

[5] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *International Conference on Software Engineering (ICSE)*, 2015.

[6] L. N. Q. Do, K. Ali, B. Livshits, E. Bodden, J. Smith, and E. Murphy-Hill, "Just-in-time static analysis," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 307–317.

[7] K. Liu, D. Kim, T. F. Bissyande, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[8] R. Bavishi, H. Yoshida, and M. R. Prasad, "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*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 613–624. [Online]. Available: https://doi.org/10.1145/3338906.3338952

[9] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *26th IEEE*

International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019, X. Wang, D. Lo, and E. Shihab, Eds. IEEE, 2019, pp. 456–467. [Online]. Available: https://doi.org/10.1109/SANER.2019.8667970

[10] C. Boogerd and L. Moonen, "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*, 2009, pp. 41–50.

[11] P. Avgustinov, A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. de Moor, M. Schäfer, and J. Tibble, "Tracking static analysis violations over time to capture developer characteristics," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, 2015, p. 437–447.

[12] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.

[13] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," 01 2006, pp. 133–136.

[14] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communications of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[15] E. W. Myers, "Ano (nd) difference algorithm and its variations," *Algorithmica*, vol. 1, no. 1-4, pp. 251–266, 1986.

[16] V. Lenarduzzi, S. Lujan, N. Saarimaki, and F. Palomba, "A critical comparison on six static analysis tools: detection, agreement, and precision," *arXiv preprint arXiv:2101.08832*, 2021.

[17] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. IEEE Press, 2013, p. 672–681.

[18] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: ACM, 2018, pp. 483–494. [Online]. Available: http://doi.acm.org/10.1145/3180155.3180206

[19] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, vol. 48, no. 3, pp. 930–950, 2022.

[20] D. Silva, J. P. da Silva, G. Santos, R. Terra, and M. T. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, vol. 47, no. 12, pp. 2786–2802, 2020.

[21] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

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

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

[24] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.

[25] N. Palix, J. Lawall, and G. Muller, "Tracking code patterns over multiple software versions with herodotos," in *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, 2010, pp. 169–180.

[26] N. Palix, J.-R. Falleri, and J. Lawall, "Improving pattern tracking with a language-aware tree differencing algorithm," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 43–52.

[27] L.-P. Querel and P. C. Rigby, "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*, 2018, pp. 892–895.

[28] D. J. Kim, B. Yang, J. Yang, and T.-H. P. Chen, "How disabled tests manifest in test maintainability challenges?" in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1045–1055. [Online]. Available: https://doi.org/10.1145/3468264.3468609

[29] F. Grund, "Codeshovel: constructing robust source code history," Ph.D. dissertation, University of British Columbia, 2019.

[30] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 81–90.

[31] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "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, 2016, pp. 470–481.

[32] J. Wang, S. Wang, and Q. Wang, "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*, 2018, pp. 1–10.

[33] T. Muske and A. Serebrenik, "Survey of approaches for handling static analysis alarms," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2016, pp. 157–166.

[34] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "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*, ser. PASTE '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 1–8. [Online]. Available: https://doi.org/10.1145/1251535.1251536

[35] F. Wedyan, D. Alrmuny, and J. M. Bieman, "The effec-

tiveness of automated static analysis tools for fault detection and refactoring prediction," in *2009 International Conference on Software Testing Verification and Validation*, 2009, pp. 141–150.

[36] A. Habib and M. Pradel, "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*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 317–328. [Online]. Available: https://doi.org/10.1145/3238147.3238213

[37] M. Yan, X. Zhang, L. Xu, H. Hu, S. Sun, and X. Xia, "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. IEEE, 2017, pp. 103–108.

[38] D. A. Tomassi, "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*, 2018, pp. 980–982.

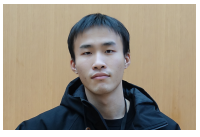
[39] A. Trautsch, S. Herbold, and J. Grabowski, "A longitudinal study of static analysis warning evolution and the effects of pmd on software quality in apache open source projects," *arXiv preprint arXiv:1912.02179*, 2019.

[40] J. Yang, L. Tan, J. Peyton, and K. A Duer, "Towards better utilizing static application security testing," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 51–60.

[41] T. Kremenek and D. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03. Berlin, Heidelberg: Springer-Verlag, 2003, p. 295–315.

[42] A. E. Burhandenny, H. Aman, and M. Kawahara, "Examination of coding violations focusing on their change patterns over releases," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2016, pp. 121–128.

[43] Y. Higo, S. Hayashi, H. Hata, and M. Nagappan, "Ammonia: an approach for deriving project-specific bug patterns," *Empirical Software Engineering*, pp. 1–29, 2020.



Junjie Li is a PhD student in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. He obtained MSc. in Computer Science at Concordia University, and received his BSc. in Computer Science of Sichuan University. Contact him at l_unjie@encs.concordia.ca.

Jinqiu Yang is an Assistant Professor in the Department of Computer Science and Software Engineering at Concordia University, Montreal, Canada. Her research interests include automated program repair, software testing, quality assurance of machine learning software, and mining software repositories. Her work has been published in flagship conferences and journals such as ICSE, FSE, EMSE. She serves regularly as a program committee member of international conferences in Software Engineering, such as ASE, ICSE, ICSME and SANER. She is a regular reviewer for Software Engineering journals such as EMSE, TSE, TOSEM and JSS. Dr. Yang obtained her BEng from Nanjing University, and MSc and PhD from University of Waterloo. More information at: https://jinqiuyang.github.io/.