ViolationTracker: Building Precise Histories for Static Analysis Violations

Ping Yu*†, Yijian Wu*†‡, Xin Peng*†, Jiahan Peng*†, Jian Zhang*†, Peicheng Xie*†, and Wenyun Zhao*†
*School of Computer Science, Fudan University, Shanghai, China
†Shanghai Key Laboratory of Data Science, Shanghai, China
‡Corresponding author: wuyijian@fudan.edu.cn

***Abstract*—Automatic static analysis tools (ASATs) detect source code violations to static analysis rules and are usually used as a guard for source code quality. The adoption of ASATs, however, is often challenged because of several problems such as a large number of false alarms, invalid rule priorities, and inappropriate rule configurations. Research has shown that tracking the history of the violations is a promising way to solve the above problems because the facts of violation fixing may reflect the developers' subjective expectations on the violation detection results. Precisely identifying the revisions that induce or fix a violation is however challenging because of the imprecise matching of violations between code revisions and ignorance of merge commits in the maintenance history.**

In this paper, we propose ViolationTracker, an approach to precisely matching the violation instances between adjacent revisions and building the lifecycle of violations with the identification of inducing, fixing, deleting, and reopening of each violation case. The approach employs code entity anchoring heuristics for violation matching and considers merge commits that used to be ignored in existing research. We evaluate ViolationTracker with a manually-validated dataset that consists of 504 violation instances and 162 threads of 31 violation cases with detailed evolution history from open-source projects. ViolationTracker achieves over 93% precision and 98% recall on violation matching, outperforming the state-of-the-art approach, and 99.4% precision on rebuilding the histories of violation cases. We also show that ViolationTracker is useful to identify actionable violations. A preliminary empirical study reveals the possibility to prioritize static analysis rules according to further analysis on the actionable rates of the rules.

I. Introduction

The quality of source code is increasingly gaining attention in the software community and industry. As an efficient means of code quality check, automatic static analysis tools(ASATs) such as SonarQube[1], PMD[2] and FindBugs[3], are widely used. However, there are quite a few problems that pose a barrier to the adoption of ASATs [1], [2]. These problems include high rate of false alarms, large number of violations, cumbersome rule configuration, invalid rule severity, and non-guarantee of real bugs [1]–[5]. Therefore, a number of approaches have been proposed to solve these problems [5]–[15].

In particular, to prune false alarms and identify actionable violations which developers would fix, many researchers [16]–[23] consider different aspects of a violation to study the features which are important for identifying actionable violations. Wang et al. [13] conducted a systematic evaluation of all the available features and filtered 23 "Golden Features" which are the most important for identifying actionable violations. Using these features, some studies [8], [24], [25] found that any machine learning technique, e.g. linear SVMs, can achieve good performance. However, Kang et al. [4] found that the strong performance of using "Golden Features" [13] to predict actionable violations was caused by the data leakage and data duplication issues in the actionable violations oracle constructed by the closed-warning heuristic. They highlighted the need for building a large and reliable benchmark of violations, where the actionable violations could be easily collected if violation-inducing and violation-fix commits were precisely identified.

On the other hand, the history of violations can also be used for customizing ASATs [9], [26], analyzing technical debt [27]–[30], evaluating project quality [31]–[33], and optimizing rule priority [16], [20], [34], etc. To this end, many advances highlight the importance of tracking static code violations and proposed violation matching methods [16], [17], [35]–[37] to track violations over time. These methods are mostly developed for matching violations but ignore the construction of whole evolutionary histories. Even so, the matching is not precise enough to be directly applied to a large number of violations in a long history of revisions [12]. The state-of-the-art approach proposed by Avgustinov et al. [35] performs ineffectively in quite a few cases [36], such as when dealing with violations located in multiple locations.

Figure 1 shows an example violation with three locations, detected by SonarQube in the revisions `635b2c` (the parent revision[4], Figure 1(a)) and `1fd449` (the child revision, Figure 1(b)) in the Skywalking[5] repository. This is a violation of the type "*Null pointers should not be dereferenced*" [38], located in the method `stopSpan` in file `TracerContext.java`. In the parent revision, the variable `lastSpan` declared at Line 150 may be *null* at Line 151 and thus may cause a `NullPointerException`. In the child revision, the same variable is declared at Line 176 and may also be null at Line 178, causing the same violation. Developers are able to identify that they are the same violations even

[1]SonarQube: https://www.sonarqube.org
[2]PMD: https://pmd.github.io
[3]FindBugs: http://findbugs.sourceforge.net

[4]In Git, a commit to a revision produces a new revision. These two revisions linked by a commit are called the *parent* and the *child*, respectively. It is the official terminology suggested by Git.
[5]Apache Skywalking: https://github.com/apache/skywalking

```java
149 public void stopSpan(AbstractSpan span, Long endTime) {
150     Span lastSpan = peek();
151     if (lastSpan.isLeaf()) {
152         LeafSpan leafSpan = (LeafSpan)lastSpan;
153         leafSpan.pop();
154         if (!leafSpan.isFinished()) {
155             return;
156         }
157     }
158     if (lastSpan == span) {
159         pop().finish(segment, endTime);
160     } else {
161         throw new IllegalStateException("Stopping the unexpected span = " + span);
162     }
163 
164     if (activeSpanStack.isEmpty()) {
165         this.finish();
166     }
167 }
```

(a) The parent revision (`635b2c`)

```java
174 @Override
175 public void stopSpan(AbstractSpan span) {
176     AbstractTracingSpan lastSpan = peek();
177     if (lastSpan == span) {
178         if (lastSpan.finish(segment)) {
179             pop();
180         }
181     } else {
182         throw new IllegalStateException("Stopping the unexpected span = " + span);
183     }
184 
185     if (activeSpanStack.isEmpty()) {
186         this.finish();
187     }
188 }
```

(b) The child revision (`1fd449`)

Fig. 1. A *Null-pointers-should-not-be-dereferenced* violation detected in two revisions of the file `TracerContext.java` in Apache Skywalking

if the type of the variable `lastSpan` is changed, the `if` condition is moved, and the method called by `lastSpan` is changed. This is because the potential *Null Pointer Exception* is esstially caused by the same reason. In other words, it is very likely that the developer who writes the code is not aware of the violation and not intentionally fixing it. These heuristics for tracking violations are typically recognized as part of the Golden Features [13]. However, the state-of-the-art tools lose the track of the two violations, producing a pair of fake-fix and fake-introduction of the violations, which may confuse developers and bring unexpected statistics data.

In order to provide a general-purpose infrastructure for precisely constructing the complete evolutionary histories of violations, we propose and implement a match-and-track approach, named ViolationTracker. It contains a violation life-cycle model that captures the inducing and fixing of violations and tracks violation cases and violation threads. ViolationTracker is designed to reveal the violations' histories and to provide with a historical view of violations. It does not only match two violations between revisions but also establishes matching status for each revision according to the complete histories of revisions.

To evaluate the effectiveness, we built manually-validated benchmark datasets of 31 violation cases and 504 violation instances that were introduced or fixed in the history. Compared to the state-of-the-art violation matching technique [35], our approach achieved 93% precision, 55 percentage points higher, and 98% recall, over 7 percentage points higher, in violation matching. Our approach additionally identified nearly all violation cases with the whole evolution histories. We also evaluate the usefulness of our approach w.r.t. typical software engineering tasks with six popular open-source Java projects. Specifically, we evaluate the usefulness of actionable violation identification compared with the *closed-warning heuristic* used in many studies [4], [8], [13], [25]. The results show that the average F1 score of our approach achieves 0.89, which is better than the *closed-warning heuristic* method [4], [8], [13], [25]. We also conduct a preliminary empirical study to analyzes the fix rate of violation between different projects. The results point to the importance of rule configuration and the oppotunities in ASAT optimization.

In summary, we make the following contributions.

- We create a benchmark dataset containing 504 manually-validated static violations and 31 violation cases with fully-analyzed evolution history details[6]. To the best of our knowledge, it is the first dataset that addresses both violation *matching* and *tracing* to serve as an established baseline for violation tracking research, and also the first dataset that consists of manually-validated full histories of violations, as far as we know.
- We propose a violation tracking approach that automatically builds precise violation histories, which enables multiple applications for typical software engineering tasks.
- We evaluate our approach on real open-source software projects and demonstrate the application scenarios of using the ViolationTracker in actionable violation identification.

The rest of the paper is organized as follows. Section II defines the terminology. Section III presents the technical details of our approach. Section IV reports the evaluation of the effectiveness of our approach and potential application scenarios of the violation evolutionary histories constructed by our approach. Section V discusses the related work before a final conclusion in Section VI.

II. Definitions

In this section, we define violation instance, violation case, violation history and related terminology.

A. Violation Instance

In a version control system such as Git, software projects are stored as repositories. Violations are detected on specific revisions, or *snapshots*, of a project by certain ASATs. We call these violations on a specific revision *violation instance*s.

Recall the violation instance detected by SonarQube described in Figure 1(a). Although the potential exception occurs only at Line 151, SonarQube indicates three *location*s that

[6]Available at https://github.com/FudanSELab/violationTracker.

are related to the violation instance. In general, a violation instance may have one or multiple locations. We formalize the definitions of a *violation instance* and a *location*.

1) Violation Instance: We define a violation instance V as a four-tuple $(r, f, t, \mathcal{L})$, where r is the revision where the violation instance is detected, f is the file where it belongs to, t is the violation type, and $\mathcal{L}$ is a set of locations. This definition is slightly different from Avgustinov et al.'s [35] in that we opt to consider multiple *location*s. It is a technical choice that contributes to the accurate matching between violation instances in adjacent revisions, as we will see in the evaluation.

2) Location: Each *location* is regarded as a syntactic component in the source code file. Normally, a syntactically-correct source code file could be parsed into an abstract semantic tree (AST). A location can correspond to a node in the AST but contains more information than a node in AST. We consider a *location* with both textual and syntactical information based on the following aspects.

- Absolute position: the line numbers in the file where the violation resides, i.e., the start line number and the end line number.
- Anchor node: an upper-level AST node of the node that the location corresponds to. Typically, if the location is in a method, the anchor is the method node; otherwise, the anchor could be the class node to which the location belongs or the root node representing the whole file. Note that we choose the Anchor node because the anchor is a program entity that can be tracked by version control systems or is comparatively stable for tracking.
- Relative position: the offset lines between the start line of the location and the start line of the anchor.
- Text: the source code text between the start line number and the end line number, inclusive.

We do not consider the context (i.e., the source code surrounding the location) of the location simply because (1) the context is likely to be changed and not easy to be referred to and (2) the intrinsic characteristics of a location lies more on the content of the code than the surrounding code.

Formally, we define a *location* L as a five-tuple $(l_s, l_e, A, l_A, code)$, where l_s and l_e are the start and end line numbers, respectively, A is the anchor node along with the source text of (e.g., the method signature), l_A is the line offset from the start line of the location to the start line of the anchor, and $code$ is the source code text at the location. For example, the first location of the violation instance in Figure 1 can be denoted as $L1$:*(150, 150, Method("stopSpan(AbstractSpan, Long)"), 13, "Span lastSpan = peek();")*, where *Method(s)* stands for the Method node of the method with signature *s*.

We declare two locations L_1 and L_2 are the same if all elements in the five-tuple are the same, denoted as $L_1 = L_2$. Given two sets of locations $\mathcal{L}_1$ and $\mathcal{L}_2$, we say $\mathcal{L}_1$ is equivalent to $\mathcal{L}_2$ iff $|\mathcal{L}_1| = |\mathcal{L}_2| \land \forall L \in \mathcal{L}_1 : \exists L' \in \mathcal{L}_2 \Rightarrow (L = L')$, denoted as $\mathcal{L}_1 \equiv \mathcal{L}_2$.

B. Match between Violation Instances and Matching Status

We empirically define that there is a *match* between two violation instances if they represent the same underlying problem in different snapshots of the source code. Previous studies [35], [39] have noted that even if experienced developers may not agree on whether two code snippets or violation instances are the *same*. Therefore, the common sense of developers is required to identify "the same violation instances". For example, the type of the violations should be the same, the code context around the violations should be very similar, the methods and files where the violation instances were located should be tracked.

We denote $V_1 \sim V_2$ if two violation instances V_1 and V_2 match. The technical details will be explored in Subsection III-C.

Given a violation instance $V_0 = (R_0, F, T, \mathcal{L})$ detected in the file F in revision R_0, and a set of violation instances $\mathcal{V}_p$ in a parent revision R_p, and a set of violation instances $\mathcal{V}_c$ in a child revision R_c, we define the *matching status* for V_0 with regard to its parent or child revisions:

- V_0 is recognized as *NEW* w.r.t. R_p iff $\neg\exists V \in \mathcal{V}_p(V \sim V_0)$;
- V_0 is recognized as *NON-CHANGED* w.r.t. R_p iff $\exists V \in \mathcal{V}_p(V \sim V_0) \land V.\mathcal{L} \equiv V_0.\mathcal{L}$;
- V_0 is recognized as *CHANGED* w.r.t. R_p iff $\exists V \in \mathcal{V}_p(V \sim V_0) \land \neg(V.\mathcal{L} \equiv V_0.\mathcal{L})$.
- V_0 is recognized as *FIXED* w.r.t. R_c iff $\neg\exists V \in \mathcal{V}_c(V \sim V_0) \land \forall L \in V_0.\mathcal{L}(L.anchor \in f_c)$;
- V_0 is recognized as *DELETED* w.r.t. R_c iff $\neg\exists V \in \mathcal{V}_c(V \sim V_0) \land \exists L \in V_0.\mathcal{L}(L.anchor \notin f_c)$,

where f_c is the snapshot of file F at revision R_c and $L.anchor$ denotes the AST node that location L is anchoring to in the snapshot of R_c.

Note that we distinguish FIXED and DELETED based on whether the anchor node is present in the child revision for it can enable various applications, such as actionable violation collection and mining fix patterns.

C. Violation Case

A *violation case* is an abstraction of the collection of all violation instances that match, representing the history of a specific violation. Figure 2 shows an example of a violation case with a complicated evolution history. Each circle node represents a revision while the linkage represents the parent-child relation between the two adjacent revisions. A red node indicates the existence of the violation instance in the revision whereas a white node indicates that the violation does not exist.

A *violation thread*, which captures the life-cycle (i.e., creation, changes, and disappearance) of the source code quality problems, starts with a NEW instance ends with a FIXED or DELETED instance or to the last commit in the maintenance history. According to this definition, a violation case may have multiple violation threads, as shown by the green lines in Figure 2. In general, a violation case is either

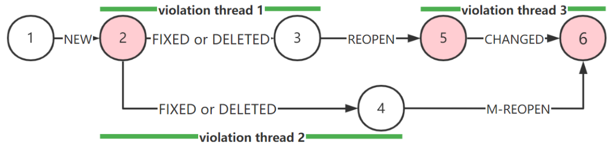

Fig. 2. A violation case with three violation threads

open or *closed* after we have traversed all violation threads over time. When a NEW violation instance is detected, we create a violation case and keep it *open* when tracking it along the violation thread. When the violation instance is FIXED or DELETED, the corresponding violation case is *closed*. For a closed violation case, we retain the details of all *last occurrence*s (i.e., the FIXED or DELETED violation instance) in the corresponding violation threads for a quick match in the future.

However, in order to track violation cases, we expect more details than merely *open* and *closed*. As shown in Figure 2, the violation instance was NEW in revision 5. However, the violation case was not new but was closed earlier in the history. Therefore the violation case should be tracked to an earlier fixed revision and marked as *re-open* in revision 5. Furthermore, when merge commits are considered, a developer may introduce or fix violations when the source code changes are merged, especially in the case of conflicts. Therefore, a violation case introduced or fixed by a merge commit should be tracked back by the two parent revisions so that developers could understand the whole life-cycle of the violation case. Previous work [35] typically disregard all merge commits but we opt to specify the details accordingly.

We refine the matching status and update the violation case status with the following heuristics.

1) Violation Case Reopen: First, we introduce REOPEN. Consider a violation instance V_0 in a revision R_0 caused by a non-merge commit to the parent revision R_p. The matching status of V_0 is updated to REOPEN w.r.t. R_p if V_0 is NEW w.r.t. R_p and there is at least one closed violation case whose fix occurrence matches V_c. The closed violation case is updated to open.

Each violation case is represented by a status (either *open* or *closed*) and all matched violation instances with a matching status attached. In Subsection III-D, we describe the technical details for creating a violation case.

2) Violation Cases at Merged Revisions: Then we introduce M-NEW and M-REOPEN to precisely describe how a violation instance V_0 is introduced or fixed in a revision caused by a merge commit. Since there are two parent revisions, we first determine the matching status w.r.t. the parent revisions separately, then we decide the matching status based on the following rules:

- V_0 is updated to M-NEW w.r.t. R_{p1} if V_0 is NEW w.r.t. R_{p1} and CHANGED/NON-CHANGED w.r.t. R_{p2}. This is the case that a clean branch is merged to a dirty branch but the violation in the dirty branch is not fixed.
- V_0 is updated to M-NEW w.r.t. R_{p1} if V_0 is NEW w.r.t. R_{p1} and R_{p2}. This is the case that two clean branches are merged but a new violation is introduced, which is typically the case that the developer resolves a conflict.
- V_0 is updated to M-REOPEN w.r.t. R_{p1} if V_0 is REOPEN w.r.t. R_{p1} and NEW/REOPEN w.r.t. R_{p2}. This is usually the case that a dirty branch is merged to a clean branch and pollutes the clean branch with the violation in the dirty one. As shown in figure 2, violation instance in revision 6 is M-REOPEN relative to revision 4.
- Otherwise, the matching status remains unchanged.

Based on these definitions, we propose our approach to building a precise history of violations.

III. Building Violation Histories

In this section, we first sketch an overview of our approach and then elaborate on each step in detail, and finally brief some implementation decisions.

A. Approach Overview

Our approach enables incremental analysis to build the precise history of violations consisting of three main phases, including Pre-Processing, Matching Violation Instances, and Tracking Violation Cases, as shown in Figure 3.

Our approach takes a git repository as input and the output is a set of violation cases, which contains matching status, violation threads, and all violation instances ever detected in each revision of the project.

In the Pre-Processing step, we employ the capability provided by the version control system (currently Git) to get the commit history list that need to be checked by our tool. Then we invoke an ASAT (currently SonarQube) to detect violations for each revision in the commit history list in an incremental manner.

In the Matching Violation Instance step, we traverse the commit history and match violation instances as per pair of revisions with a parent-of relation. Thanks to the version control system, violation instances in the files that were not changed will be automatically matched and we only do matching for those in the changed files. After this step, each violation instance is assigned an original matching status with regard to the parent revision and to the child revision, respectively.

In the Tracking Violation Cases step, we mainly create the violation cases and update the matching status as per violation case. The original violation matching status is updated according to the whole maintenance history so that REOPEN, M-NEW, and M-REOPEN violations are identified.

We describe the technical details in the following subsections.

B. Pre-Processing

The main purpose of the pre-processing is to detect all violation instances for each revision. To do this, we employ the mechanisms provided by Git to check out the source code snapshots revision by revision and detect violation

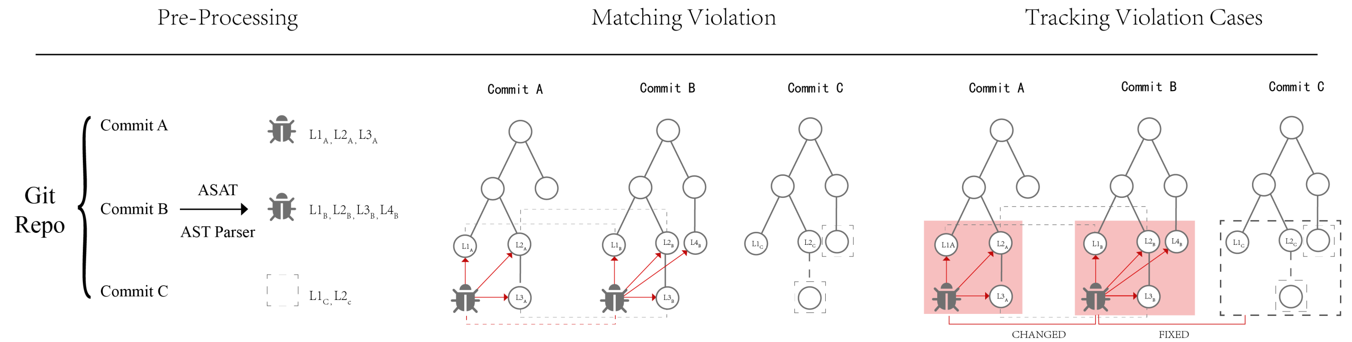

Fig. 3. An Overview of ViolationTracker

instances with an off-the-shelf SA tool (e.g., SonarQube) on each revision. Since we need to traverse all commits in the history, we conveniently reconstruct the commit graph [40] and topologically sort the commits in the graph using the Kahn algorithm [41]. This information is important for the matching step since we rely on the *parent-of* relation between the revisions.

The improvement in the implementation is that we apply incremental analysis on the changed files. Previous research has shown that analyzing every revision in a code base is resource-intensive [29], and that code changes usually affect only a small part of the source code file [42]. We also note that most of the off-the-shelf ASATs (e.g., SonarQube, PMD, and FindBugs) do not support cross-file analysis. In other words, the violation instances are found only within a single file. Therefore only detecting violation instances on the changed files is reasonable. By this means, the efficiency of violation detection is increased.

C. Matching Violation Instances

In this step, we try to create a best match in terms of *maximum matching likelihood* between the two sets of violation instances detected in two revisions that have a parent-of relation in between.

In general, all violation instances in one revision can be divided into subsets in which the violation instances are in the same file and with the same type. To match all violation instances between revisions, we first match the files in the two snapshots, then match the violation instances with the same type in the matched file pair.

1) Tracking Files: Modern version control systems provide mechanisms to track the changes of the files. Git, for example, tracks file renaming and movement as well as the files that have the same full-qualified names. Therefore, we leverage this mechanism to match files between revisions. If two files match, we regard them as the *same* file that can be tracked between the two revisions.

2) Matching Two Sets of Violation Instances: Apparently, the violation instances only match when they are of the same type. Therefore, we divide the set of violation instances into a number of subsets according to the violation types. In each subset, the violation instances are of the same type. We use $\mathcal{V}^{\mathtt{ft}}$ to represent a set of violation instances of the type $\mathtt{t}$ in the same file $\mathtt{f}$.

Now we are facing a problem how to get the *best* match between the two sets of violation instances $\mathcal{V}_p^{\mathtt{ft}}$ and $\mathcal{V}_c^{\mathtt{ft}}$. Intuitively, it could be difficult for even an experienced developer to determine whether two violation instances match especially when multiple candidate matches exist. However, human developers instinctively prescribe the *matching likelihood* based on the code and violation characteristics, such as position and source code text of each violation instance. Meanwhile, Wang et al. [13] have listed the characteristics of a violation from different aspects.

Inspired by this observation and previous studies, we define the *match-likelihood* for a pair of violation instances. Each violation instance in $\mathcal{V}_p^{\mathtt{ft}}$ is supposed to have a link, whose weight is the value of the match-likelihood, to another in $\mathcal{V}_c^{\mathtt{ft}}$. The calculation of the *match-likelihood* is to be detailed later.

We now have a bipartite weighted graph, whose nodes are the violation instances and the edges are the links with a match-likelihood weight. Our goal is to find a maximum match of the bipartite weighted graph, which can be solved by the widely-used Kuhn-Munkres algorithm [43]. Based on this observation, we describe our violation instance matching algorithm in Algorithm 1.

In this algorithm, we simply calculate the match-likelihood between every pair of violation instances. If the likelihood value is greater than zero, we take the pair as a candidate match. Then we find the best match with the Kuhn-Munkres algorithm and output the set of matched pairs in the best match.

3) Match-Likelihood for a Violation Instance Pair: Now we detail the calculation of match-likelihood in Algorithm 2. We first calculate a *location match degree* for each pair of locations of the two violation instances. The location match degree is based on the absolute position, anchor AST node, relative position, and the text of source code at the location. The four dimensions are assigned a different weight to represent different importance for the matching. The four weight factors (W_k) satisfy $\sum_{k=1}^{4} W_k = 1$. The $\mathtt{LMD}$ function

Algorithm 1: Matching the Violation Instances of the Same Type Detected in Two Files

Input: Two set of violation instances with type `t` in file `f` $\mathcal{V}_p^{\texttt{ft}}$, $\mathcal{V}_c^{\texttt{ft}}$
Output: Set of Matched Violation Instance Pairs
1 **Function** `matchViolations` $(\mathcal{V}_p^{\texttt{ft}}, \mathcal{V}_c^{\texttt{ft}})$
2 $C = \varnothing$; `// candidate pairs of matched violations`
3 $B = \varnothing$; `// best matched violation intance pairs`
4 **foreach** $V_p \in \mathcal{V}_p^{\texttt{ft}}$ **do**
5 **foreach** $V_c \in \mathcal{V}_c^{\texttt{ft}}$ **do**
6 ml = `MatchLikelihood` (V_p, V_c);
7 **if** $ml > 0$ **then**
8 $C.add(V_p, V_c, ml)$; `// a candidate match`
9 **end**
10 **end**
11 **end**
12 B = `KM` (C); `// Use Kuhn-Munkres algorithm to find best match`
13 **return** B;

Algorithm 2: Matching-Likelihood of a Pair of Violation Instances

Input: Two violation instances V_p, V_c of the same type
Output: Matching-Likelihood of (V_p, V_c)
1 **Function** `MatchLikelihood` (V_p, V_c)
2 C=$\varnothing$; `// location match candidates`
3 B=$\varnothing$; `// best matched pairs of locations`
4 **foreach** $L_p \in V_p.\mathcal{L}$ **do**
5 **foreach** $L_c \in V_c.\mathcal{L}$ **do**
6 lmd = `LMD` (L_p, L_c);`// location match degree`
7 **if** $lmd > \xi$ **then**
8 $C.add(L_p, L_c, lmd)$;
9 **end**
10 **end**
11 **end**
12 B = `KM` (C); `// to find the best location match`
13 mlc= 0; `// matched locations coverage`
14 **if** $|B|/(|V_p.\mathcal{L}| + |V_c.\mathcal{L}| - |B|) >= \theta$ **then**
15 $mlc = (\sum_{b \in B} b.locSimilarity)/\max(|V_p.\mathcal{L}|, |V_c.\mathcal{L}|)$;
16 **end**
17 **return** mlc;
18 **Function** `LMD` (L_p, L_c)
19 $minAbsLoc_p = \min(L_p.l_s + L_p.l_e,\ L_c.l_s + L_c.l_e)$;
20 $maxAbsLoc_c = \max(L_c.l_s + L_c.l_e,\ L_p.l_s + L_p.l_e)$;
21 $nc_1 = minAbsLoc_p/maxAbsLoc_p$;
22 $nc_2 = \min(L_p.l_A, L_c.l_A)/\max(L_p.l_A, L_c.l_A)$;
23 $nc_3 = equivalent(L_p.A, L_c.A)?1:0$;
24 $nc_4 = sim(L_p.code, L_c.code)$; `// code similarity`
25 $lmd = \sum_{k=1}^{4} nc_k * W_k$;`//` W_k `are weight constants`
26 **return** lmd;

calculates the location match degree.

If the location match degree is lower than a preset threshold ξ, we deny the location match; otherwise, it is a candidate location match (Line 8). Then we again use the Kuhn-Munkres algorithm to find the best match of the multiple locations out of all candidate location matches so that one-to-one location matches are established. After that, we check whether the matched location's coverage is no less than a preset threshold θ. If so, the match likelihood value is the matched locations coverage, otherwise, the match likelihood is zero.

Currently we use the following parameter values as defaults: the location match threshold $\xi = 0.7$; matched locations coverage threshold $\theta = 0.5$; the weights of the dimensions W_1 through W_4 are 0.05, 0.05, 0.2, and 0.7, respectively.

4) Traversing the Commit History: Now that we have the matching algorithm for any two sets of violation instances of the same type in the same file, we can assign each violation instance a *matching status* as defined in Subsection II-B.

Since we already have the commit topological sequence during the Pre-Processing step, we simply traverse all the revisions based on the parent-of relation and match the violation instances between them. The traversal is expensive. However, we have spent several engineering efforts (e.g., incremental analysis and multi-thread computation) to improve the performance. Moreover, this is a batch task only needed as a "cold start". When all history revisions are traversed, our tool will be automatically triggered by new commits to do the matching.

Finally, we get the matching status of each violation instance w.r.t. the parent or the child revision. Note that if a file is not changed in a commit, all violation instances in the file are marked automatically as NON-CHANGED w.r.t. the parent revision.

D. Tracking Violation Cases

In this step, we create violation cases and track them by updating the matching status of each violation case.

According to the definitions in Subsection II-C, a violation case is created only at NEW violation cases. Therefore, we revisit all NEW violation cases in topological order. For a NEW violation case, if there is no linked violation case, a new violation case is created; otherwise, an existing violation case is re-opened and the matching status of the NEW violation instance is updated to REOPEN (Subsection II-C1). In either case, all violation instances in the violation thread are linked to the violation case.

The matching status may also be updated when merge commits are considered. We then revisit all revisions that were caused by a merge commit and update the matching status accordingly (Subsection II-C2).

IV. Evaluation

A. Research Questions

To evaluate our approach, we try to answer the following research questions.

RQ1: What is the effectiveness of the matching of violation instances?

We explore the precision, recall, and F1-score of our violation instances matching algorithm. For violation instances matching, we evaluate whether the reported NEW violation instances and FIXED/DELETED violation instances are the same as a manually-validated ground-truth dataset (details in Section IV-B) of NEW and FIXED/DELETED violation instances. The evaluation of CHANGED and NON-CHANGED violation instances is trivial because, in these two cases, the violations are matched. If we thoroughly evaluated the precision and recall of the matching algorithm and find which are NEW/FIXED/DELETED, the CHANGED/NON-CHANGED instances are also identified.

For each matching status, we define precision (P), recall (R), and F1-score (F1) w.r.t. specific matching status (S) as follows.

- P_S = the number of correctly reported violation instances with status S / the number of all reported violation instances with status S;
- R_S= the number of correctly reported violation instances with status S / the number of all ground-truth violation instances with status S;
- $F1_S$= $2P_SR_S/(P_S+R_S)$.

To the best of our knowledge, the state-of-the-art competitor is Agustinov et al.'s work which is applied in the tool TeamInsight [35]. Since the tool is not open-source, we re-implemented the three violation matching strategies [35] with around 1.6K lines of Java code. In order to ensure the correctness of our re-implementation, we constructed test cases to verify the matching cases stated in the paper and successfully replicated the similar results of their work. So we use it as a baseline.

RQ2: What is the correctness of the tracking of the violation cases?

We employ the violation threads coverage for evaluating the correctness of violation case tracking. The violation threads coverage is defined as the percentage of the number of identified violation threads over the number of all ground-truth violation threads contained in a manually-validated dataset (details in Section IV-B). The reason is that, if the tracking is correct, the threads should be the same as the ground-truth. If there are any threads not covered, a loss of accuracy is observed. We also check whether the length (in days) of the violation threads is correct.

RQ3: What is the usefulness of violation histories in identifying actionable violations?

ViolationTracker can serve as an infrastructure for storing, querying and manipulating violation cases, which facilitates a wide range of applications. We demonstrate the usefulness of ViolationTracker in the application of identifying actionable violations. To evaluate the accuracy, we explore the precision, recall, and F1-score of the actionable violations compared with the *closed-warning heuristic* (CWH) method, which is widely used as the warning oracle to label actionable violations in many studies [4], [8], [13], [25]. Given two revisions, an earlier *test* revision and a latter *reference* revision, the *closed-warning heuristic* labels a violation in the test revision *actionable* if the violation is not detected in the reference revision and the file containing the violation in the test revision still exists in the reference revision. The time interval between the test revision and the reference revision is typically one year or two years. We also built a ground-truth dataset (details in Section IV-B) containing the violation cases that are manually validated as *actionable*, according to the same criteria with CWH. We define precision (P), recall (R), and F1-score (F1) as follows.

- P = the number of actionable violations correctly identified to be actionable / the number of all reported actionable violations;
- R= the number of actionable violations correctly identified to be actionable / the number of all ground-truth actionable violations;
- $F1$= $2PR/(P+R)$.

B. Datasets for Evaluation

As the subject software projects for our evaluation, we choose six highly-starred open-source Java repositories from GitHub. The basic information of the projects are listed in Table I. Each of the projects has a rich evolution history and are of comparatively higher quality in software development. Totally 13,244 commits were analyzed in our study, ranging from 303 to 4,356 in each project. The number of merge commits is up to 2,003, which is about one-third of the commits in the corresponding project (i.e., Spring-cloud-alibaba). The projects vary in size and the number of contributors, but they are all popular and actively maintained during the period we analyze, which is essential for us to capture subtle evolutionary characteristics of the violation cases. We employ SonarQube, one of the most frequently used ASATs in the community, for code analysis and totally enabled 452 rules.

For RQ1 and RQ2, we constructed a violation-matching dataset containing 501 violation instances and a violation-tracking dataset containing 162 threads of 31 violation cases, respectively. We opt to construct our own benchmark datasets as we are not aware of any existing ground truth dataset of the full histories of violation cases that take multi-branches and merge commits into consideration.

In the violation-matching dataset, the 501 violation instances are randomly selected from the subject projects. Two of the authors collaboratively validated 315 NEW violation instances and 196 FIXED/DELETED violation instances. In the violation-tracking dataset, each of the 31 violation cases underwent at least eight commit changes so that the history is long enough for tracking evaluation. All 162 violation threads cover 397 commits. We manually validated each

TABLE I
SUBJECT OPEN-SOURCE PROJECTS

Projects	Stars	Start	End	Size(LOC)	#Revisions(#Merge)	LOC(+)	LOC(-)	#Developers
Jedis	10.4k	6/11/2010	3/7/2022	222-54,816	1,966 (357)	94,574	37,901	216
Cim	8.3k	5/20/2018	10/12/2021	16-9,081	346 (27)	8,989	1,619	7
Jmeter	6.4k	1/2/2016	2/10/2019	183,730-203,958	4,356(0)	36,490	19,385	90
Skywalking	20k	2/9/2017	2/19/2020	16,136-171,855	4,153(847)	315,737	74,321	458
Curator	2.7k	2/3/2018	3/30/2022	52,241-55,737	303 (65)	4,912	1,934	131
Spring-cloud-alibaba	22.1k	8/9/2018	4/7/2022	737-24,993	2,120 (707)	100,776	16,854	162

matching status of violation instances and each violation case in the datasets. At least two authors collaboratively check the violation instances detected by SonarQube and check the adjacent commits to determine the matching status. If there is disagreement on the matching status, a third author makes the final decision before including the instance in the dataset. For each violation case, we recruit three graduate students majored in software engineering and with at least 2 years of Java development experience and manually traverse the revisions to track the violation cases. Any disagreement will be judged by an experienced author.

For RQ3, we also created a dataset for actionable violations. We first detect violation instances in the *test* revision selected in the early period in history. Then we randomly choose 50 violation instances that do not exist in the *latest* revision of the project. According to the CWH, some of these violation instances might not be actionable because we do not contract the existence of the containing file. In order to validate the actionability, we further investigate each violation instance revision by revision and manually construct the violation thread. After that, we mark a violation instance as *actionable* if it is found in a revision but not found in the next revision due to the code changes that are related to fixing the violation. Whether the code changes are related to the fixing of the violation is determined independently by two authors. Each author is familiar with SonarQube's rules and has extensive Java development experience. If a disagreement occurs, a third author with senior experience makes the final decision on the actionability before including the instance in the dataset. Finally, we get a dataset containing 277 CLOSED violation cases, among which 108 are validated as actionable. Note that, different from the CWH approach where the determination of actionability is based on a given period of time (e.g., 2 years), we develop the dataset with a long enough history (i.e., to the latest revision) and collect each revision for manual validation. This allows for flexible evaluation in the comparison with various lengths of time periods between the *test* revision and the *reference* revision in CWH.

C. *Effectiveness of Matching Violation Instances (RQ1)*

We report the precision, recall, and F1-score for violation matching results in Table II. In general, ViolationTracker outperforms TeamInsight with significantly higher precision and F1-score. This is because TeamInsight matches much fewer violation instances so that the mismatched violation instances increase the numbers of NEW and FIXED/RESOLVED instances. Therefore, TeamInsight exhibit high recall but very low precision, which is not desirable in real development circumstances because developers may lost track of the same violation cases.

In the case of inaccurate matches, ViolationTracker is mainly because the violation has moved to another anchor node and the code has changed. A more accurate match can be achieved by adjusting the matching threshold or increasing the context comparison around the violation. We also investigated a number of cases that TeamInsight failed to match and find that most of them include violations with multiple locations and code snippet changes, which can be handled better by ViolationTracker's best match mechanism.

Answer to RQ1. ViolationTracker achieves 0.959 precision and 0.981 recall on NEW violation instances and 0.901 precision and 0.98 recall on FIXED/DELETED violation instances in the benchmark dataset. The average F1-score is 0.96, outperforming TeamInsight whose average F1-score is 0.58. The reason for the improvement mainly lies in our consideration of best match mechanisms with multiple violation instances and multiple locations for each violation instance.

D. *Effectiveness of Tracking Violation Cases (RQ2)*

We use the benchmark dataset for this evaluation. Table III reports the number of violation cases and the corresponding violation threads in each project both in the benchmark dataset and in the results detected by ViolationTracker.

It shows that ViolationTracker successfully detected all 31 violation cases and 162 out of 163 violation threads. We double checked the detected threads revision by revision and find high accuracy (over 99%) of the threads in terms of the sum of the lifespans (in days) of all violation threads.

Answer to RQ2. ViolationTracker is effective in constructing the histories of violation cases. Over 99% of violation threads are precisely detected, with an accurate length of lifespan.

E. *Usefulness of Violation Histories in Identifying Actionable Violations (RQ3)*

We investigate the effectiveness of ViolationTracker in identifying actionable violations.

TABLE II
PRECISION, RECALL, AND F1-SCORE FOR VIOLATION MATCHING

Projects	Detecting NEW Violation Instances						Detecting FIXED/DELETED Violation Instances					
	P(TI)	R(TI)	F1(TI)	**P(VT)**	**R(VT)**	**F1(VT)**	P(TI)	R(TI)	F1(TI)	**P(VT)**	**R(VT)**	**F1(VT)**
Jedis	75.6%	85.3%	0.80	99.1%	96.3%	0.98	70.9%	100.0%	0.83	100.0%	97.4%	0.99
Cim	28.8%	90.5%	0.44	96.9%	100.0%	0.98	16.4%	84.8%	0.27	91.7%	100.0%	0.96
Jmeter	21.2%	78.6%	0.33	100.0%	100.0%	1.00	31.7%	83.3%	0.46	96.0%	100.0%	0.98
Skywalking	49.5%	94.0%	0.65	89.3%	100.0%	0.94	37.3%	96.9%	0.54	72.7%	100.0%	0.84
Curator	50.0%	93.8%	0.65	90.9%	93.8%	0.92	49.2%	88.6%	0.63	88.9%	91.4%	0.90
SpringCloud	78.4%	100.0%	0.88	97.6%	100.0%	0.99	70.5%	93.9%	0.81	97.1%	100.0%	0.99
Avg	48.0%	90.3%	0.63	**95.9%**	**98.1%**	**0.97**	37.6%	91.8%	0.53	**90.1%**	**98.0%**	**0.94**

TABLE III
VIOLATION CASE TRACKING ON THE BENCHMARK DATASET

Projects	#Benchmark Violation Threads (Cases)	# Reported Violation Threads	Precision	Detected / Benchmark Lifespan (days)
Jedis	56 (8)	56	100%	609.5 / 609.5
Cim	8 (5)	7	87.5%	261.7 / 262.4
Jmeter	2 (2)	2	100%	1.5 / 1.5
Skywalking	39 (6)	39	100%	65.4 / 65.4
Curator	4 (3)	4	100%	804 / 804
SpringCloud	54 (7)	54	100%	305.2 / 305.2
Total	163 (31)	162	99.4%	2,047.3 / 2,048

We compared actionable violations identified by ViolationTracker with those identified by *closed-warning heuristic*.

Unlike previous studies [4], [8], [13], [25], for the reference revision, we use three reference revisions set one, two, and three years after the test revision to check the F1-score of actionable violations. By switching the reference revision, we aim to make sure of a stable F1-score for better comparison.

To verify whether the closed violation is actionable, we adopt the similar process proposed by Kang et al. [4]. Previous studies have generally used two-year intervals between test and reference revision.

Table IV shows the precision, recall, F1-score for actionable violations. Column *# CLOSED* is the total number of CLOSED violation cases in the test revision, the number of validated CLOSED violations is in parentheses. Column *# Actionable* is the number of actionable violation instances that are manually validated.

The results show that the average F1-score of our approach achieved 0.89, which is far better than the *closed-warning heuristic* method within three years intervals. The F1-score of CWH plateaus after two years intervals, which means most of the violations are closed in two years. However, the recall is not high for some files that are deleted after violations are fixed. The improvement in F1-score is heavily dependent on ViolationTracker's ability to precisely construct violation histories. In terms of precision, our *FIXED matching status* heuristic is proven to be effective in identifying actionable violations. For the cases which ViolationTracker inaccurately labeled, most of them are caused by wrong matching, and the others are *anchor nodes* exist but the code around the violation is removed. For the missed actionable violations, they are mostly related to specific rules whose main method to fix is deleting the related code. Although ViolationTracker has missed some actionable violations, it can still accurately identify violations closed by deletion.

Answer to RQ3 The actionable violation collection automatically identified by ViolationTracker achieved 82% precision, the 96.5% recall , and the 0.89 F1-score outperform the *closed-warning heuristic*. ViolationTracker is able to build the history of violations accurately, which is useful for identifying actionable violations.

F. A Preliminary Empirical Study on Actionable Violations

Moreover, we conducted a preliminary empirical study on the subject projects to find the distribution of actionable violations across the static analysis rules. We use 221 rules from SonarQube and aggregate the number of actionable violations as per rule to find whether any rules exhibit a high rate of actionable violations.

In the subject projects, we totally detected 23,295 violation cases, among which 15,261 are CLOSED, involving 199 rules, and 8,034 are still open. 3,232 of the CLOSED violations, involving 137 rules, are detected as actionable based on the maintenance history. Table V shows the detail statistics. To explore whether some rules are more likely to produce actionable violations, we define a rule to be of *high actionable rate* if the percentage of actionable violations to all violations detected by the rule is higher than 75%. We also define a rule to be of *low actionable rate* if the percentage of actionable violations to all violations detected by the rule is less than 25%.

As shown in Table V, the numbers of rules with high and low actionable rates vary across projects. But we also find some rules that have similar actionable rates. We find three rules with a high actionable rate and eight rules with a low fix rate that are common to at least two projects in our subject

TABLE IV
PRECISION, RECALL, AND F1-SCORE FOR ACTIONABLE VIOLATIONS

Projects	CLOSED (Benchmarked) Violations	Benchmarked Actionable Violations	ViolationTracker			Closed-Warning Heuristic								
						1-Year Ref. Revision			2-Year Ref. Revision			3-Year Ref. Revision		
			P	R	F1	P	R	F1	P	R	F1	P	R	F1
Jedis	153 (50)	32	89%	100%	0.94	17%	6%	0.09	66%	66%	0.66	62%	66%	0.64
Cim	27 (27)	11	78%	100%	0.88	38%	27%	0.32	55%	100%	0.71	55%	100%	0.71
Jmeter	1,952 (50)	21	84%	100%	0.91	50%	52%	0.51	50%	86%	0.63	49%	95%	0.65
Skywalking	255 (50)	9	64%	100%	0.78	32%	100%	0.49	32%	78%	0.45	32%	78%	0.45
Curator	216 (50)	23	96%	100%	0.98	100%	39%	0.56	60%	78%	0.68	60%	78%	0.68
SpringCloud	326 (50)	12	71%	71%	0.75	34%	63%	0.44	39%	94%	0.50	39%	94%	0.50
Total	2,909 (277)	108	**83%**	**97%**	**0.89**	40%	40%	0.40	50%	81%	0.62	49%	83%	0.62

projects. The reasons for the observed high and low actionable rates differ. For example, the rules *String literals should not be duplicated* and *Track uses of "TODO" tags* have a high actionable rate in the projects largely because they are easy to fix or they need to be fixed in the development process. The rules *Non-primitive fields should not be "volatile"* and *Fields in a "Serializable" class should either be transient or serializable* have a low fix rate in the projects largely because of the high false positive rates and the difficulties for developers to understand [44], [45].

Discussions on the High/Low Actionable Rate Rules. High actionable rate rules are usually considered useful in software development while low actionable rate rules are likely to be ignored by the developers and may indicate a possible high rate of false alarms. We suggest that configuration experts review the low-fix-rate rules before deciding whether they should be enabled. We recommend that rules should be configured with considerations on the actionable rates so that developers may not be bothered by the overly-numerous unactionable violations but focus on the actionable violations that are produced by rules with high actionable rates. The precise histories of violations produced by ViolationTracker are a promising way for measuring the actionable rate accurately.

G. Threats to Validity.

The main threats to the validity of our experiments and case study are twofold. First, as for internal threats, the ground truth dataset we used to evaluate the accuracy of violation matching, violation tracking, and actionable violation identification was not very large-scale. This is because such a manual validation is very time-consuming, involving the understanding of violation mapping, code changes, and data statistics. Hence, as for violation matching, we validated 60 commits involving 2.2k matching, which is four times as much as manual validation in the similar work [35]. Also, the evolution history is only based on Git mechanisms and our work partly relies on the file-level tracking provided by Git. Since file-level tracking could be erroneous, our results are also threatened. However, Git is already the most widely used infrastructure and we find it quite reliable in most cases, which alleviates the threat. Second, the external threat is that our implementation is based on Java projects and SonarQube and might not be applicable to other languages and ASATs. Further studies on other popular programming languages (e.g., C/C++ or Python) and other ASATs are still needed to generalize our results.

V. RELATED WORK

Our work is mostly related to three research areas, including violation tracking, bug/violation dataset construction, and software history manipulation and evolution analysis.

Tracking Violations. Violation tracking is the identification of a similar violation instance between revisions of the project. The idea of using violation information provided by ASATs to track violations is not new. The current work [16], [17], [34]–[37], [46] of violation tracking mainly studies the matching between violations, which uses code matching techniques and software change histories to match the code where the violation is located and its context code. Avgustinov et al. [35] proposed a tool named Team Insight, which includes three different ways to match violations when changing files containing violations. The core of this tool is based on a combination of hash-based context matching and diff-based location matching. Based on Team Insight, Li et al. [36] introduce refactoring information and adopt the Hungarian algorithm to improve the accuracy of violation matching. Boogerd et al. [46] used diff-based matching of code snippets to track violations forward while Kim et al. [34] used it to track solved violations backward. Based on software change histories, Heckman [17] and Hanam [16] provide violation matching heuristics but are not exact enough in violation matching. These violation tracking technologies are mainly designed for violation matching, while the ViolationTracker not only provides high matching accuracy but also pays attention to the evolution process of violation to accurately reveal the evolutionary history of violation.

Bug/Violation Dataset Construction. Bug datasets lay the empirical and experimental foundation for various tasks as the fundamental infrastructure. The existing work on bug datasets can be broadly categorized into the real bug [47]–[52], which may make projects fail to pass some test case(s), and static analysis violations [11], [12], [17], [29], [53] which may not fail to pass. The data set most relevant to our research is static analysis violations. SonarCloud [53] is a cloud-based code analysis service of SonarQube, it can integrate with code hosting platforms. Some open-source projects on GitHub have been analyzed in SonarCloud, and the data is available for

TABLE V
STATISTICS OF ACTIONABLE VIOLATIONS

Projects	CLOSED Violations		Actionable Violations		Open Violations		#Rules	#Rules
	#	#Rules	#	#Rules	#	#Rules	High Actionable Rate	Low Actionable Rate
Jedis	1,464	106	689	67	565	70	15	6
Cim	200	42	73	19	256	62	2	3
Jmeter	2,449	129	1247	93	2,023	116	16	23
Skywalking	9,329	163	1,010	91	3,177	126	6	0
Curator	132	34	31	19	949	79	0	27
SpringCloud	1,687	100	182	49	1,064	82	1	1
Total	15,261	199	3,232	137	8,034	189	37	51

users. Kim et al. [29] have spent over a month analyzing 19,800 revisions of fifty-seven systems using Sonarqube and opened up the data. But the strategy they choose for the revisions is picking the one and the last commit of each week from the projects. Marcilio et al. [11] implement a tool that can extract several data from SonarQube instances for dealing with distinct versions of SonarQube is a challenge. Liu et al. [12] constructed a dataset of static analysis violations using FindBugs. This dataset contains all revisions of selected projects but does not check the relationship between versions of violations, and FindBus stopped updating. Our violation benchmark dataset is based on Java projects scanned by Sonarqube. Unlike other data sets, ours focus on static violations with detailed evolution histories.

Software History Manipulation There is a huge assemblage of work on software history manipulation. The basic research goal is mining and using valuable information about code [12], [34], [54]–[56] and developers [27], [29], [31], [35] from software history. Steinbeck et al. [54] developed a Java programming library for repository mining, and Wu et al. [57] proposed a representation to store, query, and manipulate software history facts. Li et al. [56] proposed a semantic slicing approach to help understand software features. Zeller et al. [55] used software change history to localize bugs, and Wang et al. [58] use it to explain bugs. As for automatic repair of software, Tan et al. [59] proposed an approach to repair software regressions and Liu et al. [12] leverage CNNs and X-means to mine fix patterns for Findbugs violations. Kim et al. [34] propose a history-based warning prioritization by mining warning fix histories. Our research focuses on accurately rebuilding the evolution history of violations, which is useful in multiple software engineering applications.

VI. CONCLUSION AND FUTURE WORK

In this work, we propose ViolationTracker which can track the histories of violations by precisely matching the violation instances between adjacent revisions. ViolationTracker's goal is to help mine information concealed in the history of violations by precisely revealing their evolutionary history. Our experiments show that compared with the state-of-the-art approach, we have higher accuracy in violation matching, and at the same time, we can thoroughly uncover the evolution of the violation cases. Our case study shows the usefulness of ViolationTracker in actionable violation identification and violation fix analysis.

In the future, we will further explore interesting applications utilizing the evolution history of violations, such as collaboration relationships between developers, optimizing configurations of static violation rules, and prioritizing the detected violations according to the potential of false alarms. In addition, we will continue to expand our data set of fixed violations to explore different violation fixing patterns and provide accurate and generally-applicable automatic methods for fixing violations.

ACKNOWLEDGMENT

This work was supported in part by National Natural Science Foundation of China (62172099).

REFERENCES

[1] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 672–681.

[2] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, H. C. Gall, and A. Zaidman, "How developers engage with static analysis tools in different contexts," *Empir. Softw. Eng.*, vol. 25, no. 2, pp. 1419–1457, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09750-5

[3] 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, pp. 58–66, 2018. [Online]. Available: https://doi.org/10.1145/3188720

[4] H. J. Kang, K. L. Aw, and D. Lo, "Detecting false alarms from automatic static analysis tools: How far are we?" in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. IEEE, 2022, pp. 698–709. [Online]. Available: https://ieeexplore.ieee.org/document/9793908

[5] 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, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 317–328. [Online]. Available: https://doi.org/10.1145/3238147.3238213

[6] F. Zampetti, S. Mudbhari, V. Arnaoudova, M. D. Penta, S. Panichella, and G. Antoniol, "Using code reviews to automatically configure static analysis tools," *Empir. Softw. Eng.*, vol. 27, no. 1, p. 28, 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10076-4

[7] D. A. Tomassi and C. Rubio-González, "On the real-world effectiveness of static bug detectors at finding null pointer exceptions," in *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 2021, pp. 292–303. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678535

[8] X. Yang, J. Chen, R. Yedida, Z. Yu, and T. Menzies, "Learning to recognize actionable static code warnings (is intrinsically easy)," *Empir. Softw. Eng.*, vol. 26, no. 3, p. 56, 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-09948-6

[9] Y. Ueda, T. Ishio, and K. Matsumoto, "Automatically customizing static analysis tools to coding rules really followed by developers," in *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021.* IEEE, 2021, pp. 541–545. [Online]. Available: https://doi.org/10.1109/SANER50967.2021.00062

[10] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarqube rules inducing bugs?" in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs, and M. Zhou, Eds. IEEE, 2020, pp. 501–511. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054821

[11] D. Marcilio, R. Bonifácio, E. Monteiro, E. D. Canedo, W. P. Luz, and G. Pinto, "Are static analysis violations really fixed?: a closer look at realistic usage of sonarqube," in *Proceedings of the 27th International Conference on Program Comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019*, Y. Guéhéneuc, F. Khomh, and F. Sarro, Eds. IEEE / ACM, 2019, pp. 209–219. [Online]. Available: https://doi.org/10.1109/ICPC.2019.00040

[12] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, vol. 47, no. 1, pp. 165–188, 2018.

[13] 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, ESEM 2018, Oulu, Finland, October 11-12, 2018*, M. Oivo, D. M. Fernández, and A. Mockus, Eds. ACM, 2018, pp. 17:1–17:10. [Online]. Available: https://doi.org/10.1145/3239235.3239523

[14] L. Wei, Y. Liu, and S. Cheung, "OASIS: prioritizing static analysis warnings for android apps based on app user reviews," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 672–682. [Online]. Available: https://doi.org/10.1145/3106237.3106294

[15] W. Lee, W. Lee, D. Kang, K. Heo, H. Oh, and K. Yi, "Sound non-statistical clustering of static analysis alarms," *ACM Trans. Program. Lang. Syst.*, vol. 39, no. 4, pp. 16:1–16:35, 2017. [Online]. Available: https://doi.org/10.1145/3095021

[16] 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*, 2014, pp. 152–161.

[17] S. Heckman and L. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2008, pp. 41–50.

[18] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *Fourth International Workshop on Mining Software Repositories, MSR 2007 (ICSE Workshop), Minneapolis, MN, USA, May 19-20, 2007, Proceedings.* IEEE Computer Society, 2007, p. 27. [Online]. Available: https://doi.org/10.1109/MSR.2007.26

[19] U. Koc, S. Wei, J. S. Foster, M. Carpuat, and A. A. Porter, "An empirical assessment of machine learning approaches for triaging reports of a java static analysis tool," in *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019.* IEEE, 2019, pp. 288–299. [Online]. Available: https://doi.org/10.1109/ICST.2019.00036

[20] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. G. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 2008, pp. 341–350. [Online]. Available: https://doi.org/10.1145/1368088.1368135

[21] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011.* IEEE Computer Society, 2011, pp. 299–308. [Online]. Available: https://doi.org/10.1109/ICST.2011.51

[22] U. Yuksel and H. Sözer, "Automated classification of static code analysis alerts: A case study," in *2013 IEEE International Conference on Software Maintenance, Eindhoven, The Netherlands, September 22-28, 2013.* IEEE Computer Society, 2013, pp. 532–535. [Online]. Available: https://doi.org/10.1109/ICSM.2013.89

[23] C. C. Williams and J. K. Hollingsworth, "Automatic mining of source code repositories to improve bug finding techniques," *IEEE Trans. Software Eng.*, vol. 31, no. 6, pp. 466–480, 2005. [Online]. Available: https://doi.org/10.1109/TSE.2005.63

[24] X. Yang and T. Menzies, "Documenting evidence of a reproduction of 'is there a "golden" feature set for static warning identification? - an experimental evaluation'," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds. ACM, 2021, p. 1603. [Online]. Available: https://doi.org/10.1145/3468264.3477220

[25] X. Yang, Z. Yu, J. Wang, and T. Menzies, "Understanding static code warnings: An incremental AI approach," *Expert Syst. Appl.*, vol. 167, p. 114134, 2021. [Online]. Available: https://doi.org/10.1016/j.eswa.2020.114134

[26] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, "Analyzing the state of static analysis: A large-scale evaluation in open source software," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1.* IEEE Computer Society, 2016, pp. 470–481. [Online]. Available: https://doi.org/10.1109/SANER.2016.105

[27] L. Xavier, F. Ferreira, R. Brito, and M. T. Valente, "Beyond the code: Mining self-admitted technical debt in issue tracker systems," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 137–146.

[28] D. Tsoukalas, D. D. Kehagias, M. G. Siavvas, and A. Chatzigeorgiou, "Technical debt forecasting: An empirical study on open-source repositories," *J. Syst. Softw.*, vol. 170, p. 110777, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110777

[29] G. Digkas, M. Lungu, P. Avgeriou, A. Chatzigeorgiou, and A. Ampatzoglou, "How do developers fix issues and pay back technical debt in the apache ecosystem?" in *2018 IEEE 25th International Conference on software analysis, evolution and reengineering (SANER).* IEEE, 2018, pp. 153–163.

[30] N. Saarimäki, V. Lenarduzzi, and D. Taibi, "On the diffuseness of code technical debt in java projects of the apache ecosystem," in *Proceedings of the Second International Conference on Technical Debt, TechDebt@ICSE 2019, Montreal, QC, Canada, May 26-27, 2019*, P. Avgeriou and K. Schmid, Eds. IEEE / ACM, 2019, pp. 98–107. [Online]. Available: https://dl.acm.org/citation.cfm?id=3355350

[31] V. Lenarduzzi, N. Saarimäki, and D. Taibi, "Some sonarqube issues have a significant but small effect on faults and changes. A large-scale empirical study," *J. Syst. Softw.*, vol. 170, p. 110750, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110750

[32] T. Das, M. D. Penta, and I. Malavolta, "Characterizing the evolution of statically-detectable performance issues of android apps," *Empir. Softw. Eng.*, vol. 25, no. 4, pp. 2748–2808, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09798-3

[33] 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," *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 5137–5192, 2020. [Online]. Available: https://doi.org/10.1007/s10664-020-09880-1

[34] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, I. Crnkovic and A. Bertolino, Eds. ACM, 2007, pp. 45–54.

[35] P. Avgustinov, A. I. Baars, A. S. Henriksen, G. Lavender, G. Menzel, O. De Moor, M. Schafer, and J. Tibble, "Tracking static analysis violations over time to capture developer characteristics," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 437–447.

[36] J. Li, "A better approach to track the evolution of static code warnings," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion).* IEEE, 2021, pp. 135–137.

[37] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking defect warnings across versions," in *Proceedings of the 2006 International Workshop on Mining Software Repositories*, 2006, pp. 133–136.

[38] SonarSource, "Null pointers should not be dereferenced," [Online] Available: https://rules.sonarsource.com/java/RSPEC-2259, 2022, Last Accessed on: April. 2022.

[39] S. Kim, K. Pan, and E. J. Whitehead, "When functions change their names: Automatic detection of origin relationships," in *12th Working Conference on Reverse Engineering (WCRE'05)*. IEEE, 2005, pp. 10–pp.

[40] Y. Kim, J. Kim, H. Jeon, Y.-H. Kim, H. Song, B. Kim, and J. Seo, "Githru: Visual analytics for understanding software development history through git metadata analysis," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 2, pp. 656–666, 2020.

[41] K. E. Iverson, "A programming language," in *Proceedings of the 1962 spring joint computer conference, AFIPS 1962 (Spring), San Francisco, California, USA, May 1-3, 1962*, G. A. B. III, Ed. ACM, 1962, pp. 345–351. [Online]. Available: https://doi.org/10.1145/1460833.1460872

[42] K. Huang, B. Chen, X. Peng, D. Zhou, Y. Wang, Y. Liu, and W. Zhao, "Cldiff: generating concise linked code differences," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, M. Huchard, C. Kästner, and G. Fraser, Eds. ACM, 2018, pp. 679–690. [Online]. Available: https://doi.org/10.1145/3238147.3238219

[43] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the society for industrial and applied mathematics*, vol. 5, no. 1, pp. 32–38, 1957.

[44] StackOverflow, "Is marking string type reference as volatile safe," [Online] Available: https://stackoverflow.com/questions/61628641/is-marking-string-type-reference-as-volatile-safe, 2022, Last Accessed on: April. 2022.

[45] ——, "Fields in a serializable class should either be transient or serialzable," [Online] Available: https://stackoverflow.com/questions/49632332/fields-in-a-serializable-class-should-either-be-transient-or-serializable, 2022, Last Accessed on: April. 2022.

[46] C. Boogerd and L. Moonen, "Evaluating the relation between coding standard violations and faultswithin and across software versions," in *2009 6th IEEE International Working Conference on Mining Software Repositories*. IEEE, 2009, pp. 41–50.

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

[48] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[49] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "Bugbench: Benchmarks for evaluating bug detection tools," in *Workshop on the evaluation of software defect detection tools*, vol. 5. Chicago, Illinois, 2005.

[50] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz, "Begbunch: benchmarking for c bug detection tools," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, 2009, pp. 16–20.

[51] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, 2007, pp. 433–436.

[52] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[53] SonarSource, "SonarCloud," [Online] Available: https://sonarcloud.io/, 2022, Last Accessed on: April. 2022.

[54] M. Steinbeck, "Mining version control systems and issue trackers with LibVCS4j," in *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, K. Kontogiannis, F. Khomh, A. Chatzigeorgiou, M. Fokaefs, and M. Zhou, Eds. IEEE, 2020, pp. 647–651.

[55] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[56] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Semantic slicing of software version histories," *IEEE Transactions on Software Engineering*, vol. 44, no. 2, pp. 182–201, 2017.

[57] X. Wu, C. Zhu, and Y. Li, "Diffbase: A differential factbase for effective software evolution management," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 503–515.

[58] H. Wang, Y. Lin, Z. Yang, J. Sun, Y. Liu, J. S. Dong, Q. Zheng, and T. Liu, "Explaining regressions via alignment slicing and mending," *IEEE Transactions on Software Engineering*, 2019.

[59] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 471–482.