Towards Understanding Fixes of SonarQube Static Analysis Violations: A Large-Scale Empirical Study

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

***Abstract*—Automated static analysis tools (ASATs) have become an integrated part of the software development workflow in many projects. While developers benefit from these tools to deliver quality code conforming to the pre-defined static analysis rules, it has been reported that many ASATs are underused. A number of detected violations are overlooked by developers due to false alarms or unactionable alerts. Despite of existing studies on the fixes of static analysis violations, there is still a gap in collecting and understanding the fact that some types of violations are fixed more often and/or more quickly than other types. To fill this gap, we conduct a large-scale empirical study on 56,506,892 violations from 30 active, popular, and high-quality open-source Java projects with long evolution histories. All violations were traced between adjacent revisions before we filtrated the *fixed* violations out of the *closed* ones by considering the types of source code changes that closed the violations. We identified the violation types with the highest and lowest fix rates and those that were fixed the most timely and least timely, and further investigated the possible underlying reasons for the differences in fix rate and fix time. Our findings is helpful to characterize and understand developers' considerations when fixing violations and provide practical implications for developers, tool builders and researchers to optimize the usage and design of ASATs.**

***Index Terms*—static analysis violations, violation fixes, evolution analysis, static analysis rules**

I. Introduction

Automatic static analysis tools (ASATs), such as SonarQube [1], PMD [2] or FindBugs [3], have been proposed to the software development community to help detect bugs, vulnerabilities or bad programming practices. The use of ASATs is a prominent approach to improve source code quality, as they reveal recurrent code violations.

With ASATs integrated into modern software development workflow and the widespread usage in both open source and industry software [4], [5], empirical studies have been conducted to explore the barriers, usage, cost, benefits, and needs when developers use ASATs. Some of the studies [6]–[8] investigated the overall relationships between violations and software development but overlooked the whole evolution history before the fixes of violations. They typically relied on inconsecutive versions and/or treated all *closed* violations, including those closed by file or code deletion, as *fixed*. Other studies matched the violations between adjacent revisions and tracked them over time [4], [9], [10] but they either did not make explicit distinction between real fixes and code deletions, or served for other purposes such as developer behavior characterization [9] or mining fixing patterns [11] and did not focus on understanding the fixes from the perspective of types of violations.

Despite of existing researches on violation fixes, empirical evidence is still missing for developers, ASAT tool builders, and researchers to understand what characteristics of source code changes are more likely to be violation fixes and which types of violations are mostly and timely fixed. The reason behind the facts that some types of violations are mostly and timely fixed whereas others are not is also worthy to be investigated.

To fill the gap in identifying and understanding the fixes of violations, we conducted a large-scale empirical study on 56,506,892 violations detected from over 32 thousand commits of 30 active, popular, and high-quality open-source Java projects, mainly from Apache Software Foundation (ASF) and Google, and identified 80,335 distinct violations by matching violations between adjacent revisions, among which 22,693 are closed. By analyzing the histories of all these violations, we answered the following three research questions.

- RQ1: Characteristics Analysis. What kinds of code changes (i.e., deletion or modification) typically indicate fixed violations? (Sec. IV-B)
- RQ2: Fix Rate Analysis. Which types of violations are fixed by developers the most and the least? (Sec. IV-C)
- RQ3: Fix Time Analysis. How long does it take developers to fix different types of violations? (Sec. IV-D)

The findings listed below serve as a summarization of the above research questions. In a sample of 300 cases of closed violations, over 93% of violations closed by code modifications are manually confirmed to be fixes, while only less than 28% of violations closed by code deletion are confirmed fixes. This indicates the feasibility that we use violations *closed* by code modifications to determine an actual *fix*. The violations that are easy to understand and can be detected in integrated development environments (IDEs) are fixed the most. Violations related to programming styles and certain code smells are much less fixed. The fix rate of bug violations is primarily related to the rate of false positives and whether the developer is aware of the violations. Moreover, 30% of all closed violations are fixed within a month, and 85% are fixed within a year. The top-10 violation types with the longest and shortest fixing time are mostly in the category of Code Smell and of various levels of severity, indicating

that the timeliness of violation fixes is not likely related to specific categories or severity levels but mainly depends on the developers' awareness and understandings of the violations. Our findings provide concrete empirical data for developers, ASATs builders, and researchers to investigate potential false alarms of ASATs and to optimize the usage and design of ASATs.

In summary, this work makes the following contributions.

- We manually confirmed that most violations closed by source code modifications can be determined as fixed.
- We conducted a large-scale empirical study to understand the characteristics and causes of more and less fixed violations and the timeliness of violation fixes.
- We provided practical implications of our findings to three audiences: developers, researchers, and tool builders.

II. Preliminaries

A. Automatic Static Analysis Tools (ASATs)

ASATs, such as SonarQube [1], is widely used as a guard for source code quality. ASATs detect source code violations to predefined rules by statically analyzing the source code or bytecode without running the program. Besides, ASATs can detect different categories of defects, such as bugs, vulnerabilities, and code smells. Each defect corresponds to a rule prompting developers with specific information. Each rule belongs to a category and has a level of severity. These defects are referred as *static analysis violations* [9], [11]–[13] or *warnings* [4], [14]–[16]. In this paper, we denote them as *violations* and use SonarQube as the infrastructure to detect violations.

SonarQube As one of the most common open source ASATs adopted both in academia and industry [17], SonarQube can analyze more than 30 different programming languages. Developers can directly use its service through the SonarCloud platform or download and configure it on a private server. In community edition version 8.9.1, SonarQube has 627 rules for Java by default, involving four *categories*: Code Smell, Bug, Vulnerability, and Security Hotspot. SonarQube claims that zero false-positives are expected for Code Smell and Bug, while for Vulnerability and Security Hotspot, the target is to have more than 80% of violations be true-positives[1]. Furthermore, SonarQube classifies the rules into five *severity* levels and evaluates rule severity from the worst impact on software quality and the likelihood of the worst happening.The five severity levels include Info, Minor, Major, Critical, and Blocker. In this paper, we adopt the 452 rules that are officially recommended by SonarQube that generally apply to most projects.

B. *Distinguishing* Fixed *Violations from* Closed *Violations*

Typically, we say that a violation was *closed* in Revision r if it was present in Revision $r-1$ but not present in Revision r. The reasons for closing violations varies. In some cases, developers fix the violations intentionally, but in some other cases, the source code containing the violation might be removed or changed for other purposes so that the violation is closed accidentally. Literatures [14], [18] have referred to the violations that developers would fix as actionable or true positives. The identification process of actionable violations can be simplified into three stages: fixed violation collection, feature recognition, and prediction. For the fixed violations collection, previous studies [11], [14], [15], [18]–[21] typically consider two characteristics to identify a fixed violation when it was closed: (1) a file or a method enclosing the violation was deleted; (2) specific lines of source code were changed (including code lines deletion). Violations that meet the first characteristic are regarded as unactionable or false positives, while those meeting the second one are regarded as fixed or actionable violations. In this paper, we also adopt this criteria to identify *fixed* violations out of *closed* violations.

[1]https://docs.sonarqube.org/latest/user-guide/rules/

III. Empirical Study Methodology

A. Study Design

The three research questions introduced in Section I bring values from different perspectives. The characteristics analysis in RQ1 analyzes the overall modification characteristics when closing violations. Our findings from RQ1 can identify the fixed violations of closed violations and promote the related research. The fix rate analysis in RQ2 analyzes the proportion of fixed violations in terms of category, severity, and type. Our findings in RQ2 can answer the factors that influence developers to fix violations and to what extent the category, severity, and type affect the rate of fixed violations. Furthermore, it can also answer whether the closing violation, proposed by Digkas et al. [7], is intentionally fixed by the developer or a side effect of source code evolution. The fix time analysis in RQ3 reports the overall distribution of time to fix violations and the fix time per type. Our findings from RQ3 can identify which violation types developers are more concerned about and the factors affecting the timeliness of violation fixes.

To answer these research questions, our study employs incremental analysis to build the history of violations consisting of four main phases: (1) identifying modified files (diffs) between revisions, (2) applying SonarQube to collect violations for diff files, (3) matching and tracking violations across the revision history, and (4) identifying code modifications of closed violations to distinguish fixed violations. Different from previous works [6], [7], we focus on fixed violations extracted from consecutive violation history by considering every commit. An overview of the methodology is shown in Fig. 1. Since the first phase is mainly based on the mechanisms provided by Git, we elaborate the following three phases in the next subsections.

B. Collecting Violations

To collect violations from a code base, we apply SonarQube to consecutive revisions of the associated project's source code. Since the cost of analyzing violations from the consecutive revision of project source code is high [6], without analyzing each version, we can't accurately identify

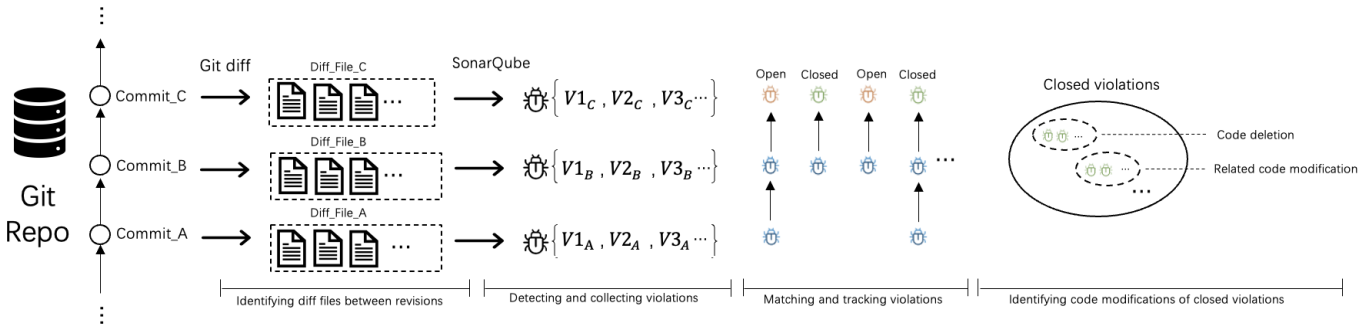

Fig. 1. An overview of the study methodology

```
ProjectName:rocketmq
StartCommit:515bc3538d71cbe3f0fd293f420644e907b41d10
EndCommit:8f788c3d09d8ae279959be0a56df7f4854b4bd87
FilePath:store/src/main/java/org/apache/rocketmq/
    store/dledger/DLedgerCommitLog.java
Type:Null pointers should not be dereferenced
Locations:
    location1:360-360
    location2:451-451
```

Fig. 2. An example of a two-location violation of type *Null pointers should not be dereferenced* detected in commit 515bc35 and closed in commit 8f788c3 in file DLedgerCommitLog.java of rocketmq

the revision and context when a violation is introduced or closed. Researchers [7], [14] have shown that most violations will be closed within two years. To this end, we analyze all revisions from the last four years of the code bases to strike a balance between reliable data and resources. For each violation instance, we record it as a five-tuple $(s_c, e_c, f, t, \mathcal{L})$, where s_c is the commit where the violation instance is detected, e_c is the commit where the violation is closed, f is the file where it belongs to, t is the type, and $\mathcal{L}$ is a set of locations. This definition is different from previous studies [9], [11] in that we opt to consider multiple *location*s. For each location, we record it as a two-tuple (s_l, e_l), where s_l and e_l are the start and end lines of the location. Fig. 2 shows an example of a violation with three locations in the collected dataset.

C. Tracking Violations

Violation tracking is the matching of similar violation instances between project revisions. ASATs often report the locations of the violation with line numbers. In many cases, the violation locations can be moved or changed through code changes, making matching not a simple task. However, suppose the matching is not accurate. In that case, there will be many false positives and negatives in identifying newly introduced and closed violations, affecting the analysis and understanding of fixed violations. Thus, previous work [4], [9], [10], [20]–[23] 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. The state-of-the-art approach was proposed by Avgustinov et al. [9] and integrated into the tool named Team Insight. This approach includes three different matching strategies (i.e., location-based violation matching, snippet-based violation matching, and hash-based violation matching) to match violations when changing files containing violations. However, this approach is less accurate when code refactoring and multiple similar violations exist. Based on this approach, Li et al. [4] introduce refactoring information and adopt the Hungarian algorithm [24] to improve the accuracy of violation matching. Other techniques for matching violations are not precise enough to be automatically applied to many violations in a long revision history [11]. In this paper, we follow the methods proposed by Avgustinov et al. [9] and adopt the improvements by Li et al. [4] to match the similar violations between revisions.

D. Identifying Code Modifications of Closed Violations

Once the violation matching is completed, we can collect the closed violations and identify the code modification to pick the fixed ones. We generally divide the code modification leading to violation closure into two categories: deletion and modification, similar to previous studies [11], [14]. These studies divided deletion into file deletion and method deletion, which cannot describe some violations that are not located in the source code of methods. For example, the violations with type *"volatile" variables should not be used with compound operators* are generally located in fields, and those with type *"@SpringBootApplication" and "@ComponentScan" should not be used in the default package* are generally located in annotations of the method. Thus, to describe the code modification information more generally, we uniformly refer to the code entity to which the violation location belongs as **anchor**. Typically, if the location is in a method, the anchor is the method; otherwise, the anchor could be the class or field to which the location belongs. Note that we choose the **anchor** because the **anchor** is a code entity that can be tracked by version control systems or is comparatively stable for tracking. In this paper, we divide deletion (A, B, and C) and modification (D and E) into five subcategories. The detailed definitions and heuristic judgment criteria are as follows.

A: File deletion. The file containing the violation is deleted.

B: Anchor deletion. The file containing the violation is not deleted but at least half of the anchors to which the violation locations belong are deleted.

C: Code deletion. Not in the cases of A and B but all locations of the violation code are deleted.

D: Related code modification. At least one line at the locations of the violation code is changed.

E: Unrelated code modification. None of the source code at the violation location is changed.

We use a code differencing algorithm to classify code modification. There is a vast assemblage of work using text-based [25], [26] and tree-based [27], [28] approaches for the code differencing analysis. We adopt well-known text-based Myers diffing algorithms [25] to derive source code mappings for an individual file, which is also used as one of the algorithms in location-based violation matching adopted in state-of-the-art approach [9]. For the identification of fixed violations from closed ones, to the best of our knowledge, the recent study on this is from Kang et al. [14]. They have been mentioned that it may not be appropriate to identify violations closed by file changed as fixed if the commit of closing violations cannot be accurately determined. Other researchers directly regard violations closed by file changes as fixed without carefully checking if it's reasonable. Thus, instead of assuming which code changes can accurately identify closed violation as fixed, we put this part of the verification content in Section IV-B to elaborate.

E. Data Preparation

Since most of the previous work [5], [6], [11], [14] is carried out in Java language projects, and the implementation for Java in violation matching algorithm is relatively mature, we choose the project with Java as the primary language. Furthermore, we need projects that include various code changes to generalize our conclusions as much as possible. Thus, we randomly select 30 high-star projects from the Apache Foundation and Google, all of which are recently active, have a rich history (i.e., the project has at least 500 commits), and have a not-small project scale (i.e., the project contains at least 100 Java files). We select the default main branch for each project to analyze all commits for at least the last four years.

For RQ1, we aim to explore which code modification would indicate that the closed violation is fixed. To strike a balance between the choice of project and rule type, we first selected the top six projects with the highest number of violations. We then randomly selected ten different rule types from each item, with each rule type corresponding to at least five closed violations, excluding those cases that were closed due to file deletion. As a result, we selected 60 violation types and 300 closed violations. After that, we mark a closed violation as fixed if found in a revision but not in the next revision due to the code changes related to fixing the violation. Whether the code changes are related to the fixing of the violation is determined independently by the 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 experiences decides whether the closed violation is actually fixed.

TABLE I
OPEN-SOURCE PROJECTS USED IN THIS STUDY

Projects	30
Commits	32,691
Violations Instances	56,506,892
Distinct violations (Closed)	80,335 (22,693)
Violations types (Closed)	339 (270)

For RQ2 and RQ3, we first use code modification to pick out fixed violations based on the findings in RQ1. We disregard the closed violations due to file and anchor deletions when answering RQ2 and RQ3. This is because violations closed by file or anchor deletions are most likely not really fixes but only a side effect of software maintenance. In other words, the violation contained by the source code might not be fixed even if the source code had not been deleted. Then, we calculate the violation fix rate and fix time with all violations and for each violation type. The violation fix time is the elapsed days from the commit time of violation introduction to the commit time of violation fix, which is also used in the previous work [6], [7]. If the violation is fixed in the same day of introduction, the fix time of the violation is denoted as 0.

IV. RESULTS

A. Statistics on Detected Violations

We have totally detected 56,506,892 violations involving 339 SonarQube rules from the target projects[2]. After applying the violation tracking method, 80,335 distinct violations are identified, involving 270 SonarQube rules. Among all distinct violations, we find only 22,693 closed violations, with a close violation rate of 28%. Project shenyu has the highest violation close rate, which is 82%, while project opennlp has the lowest close rate only arriving at 1%. In 23 (77%) projects, the close violation rates are less than 40%, indicating that most violations still exist in the projects.

B. Characteristics Analysis (RQ1)

We investigate the different code modification actions developers took when they closed the violations and verify whether the closed violation is *fixed*. We computed Cohen's Kappa to measure the annotator agreement on the fixed violation judgment and obtained a value of 0.82, considered a strong agreement [29]. Table II shows the precision of identifying fixed violations from closed violations. The numerator in brackets in the table represents the number of violations we artificially determined to be fixed, and the denominator is the number of closed violations. The results show that the Anchor Deletion does not indicate that the closed violation has been fixed. Still, the Related Code Modification and Unrelated Code Modification can indicate that the violation has been fixed to a large extent. Besides, we found that it is possible to use

[2]Our dataset for replication are released at https://github.com/FudanSELab/sq-violation-dataset

TABLE II
PRECISION OF IDENTIFYING FIXED VIOLATION FROM CLOSED VIOLATION

Projects	Deletion		Modification		Total
	Anchor Deletion	Code Deletion	Related Code Modification	Unrelated Code Modification	
hudi	0%(0/8)	46%(6/13)	95%(19/20)	89%(8/9)	66%(33/50)
guava	0%(0/6)	40%(2/5)	93%(26/28)	81%(9/11)	74%(37/50)
logging-log4j2	0%(0/5)	0%(0/4)	97%(28/29)	100%(12/12)	80%(40/50)
nomulus	0%(0/5)	50%(5/10)	100%(21/21)	93%(13/14)	78%(39/50)
nutch	0%(0/2)	47%(8/17)	89%(17/19)	83%(10/12)	70%(35/50)
shenyu	0%(0/6)	38%(5/13)	100%(18/18)	92%(12/13)	70%(35/50)
Total	0%(0/32)	42%(26/62)	96%(129/135)	90%(64/71)	73%(219/300)

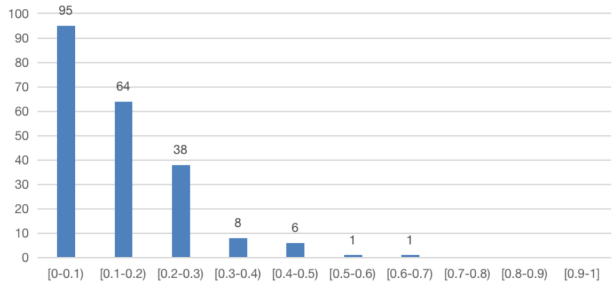

Fig. 3. Distribution of fix rate for all violation types

Code Deletion to determine whether a violation is fixed for specific rules. This is primarily related to the dominant fixing pattern of these violation types. For example, violations of type *Deprecated code should be removed* and *Unnecessary imports should be removed* is mainly fixed by code deletion. There are other ways, such as reusing discarded code or using packages that have been imported but not used before, to fix these violations, while these ways are a side effect of software evolution. Other ways to fix these violations include reusing discarded code or using imported but previously unused packages. In fact, fixing violations in these ways can be regarded as a side effect of software evolution.

Related Code modification and Unrelated Code Modification can be used to identify whether violations are fixed. At the same time, Code Deletion can be used for a small number of specific rules.

C. Fix Rate Analysis (RQ2)

With respect to RQ2, we focus on the following sub-questions.

- RQ2-1: What is the distribution of the fixed violations overall?
- RQ2-2: What types of violations are mostly fixed in different projects?
- RQ2-3: What types of violations remain unfixed in different projects?

1) RQ2-1: We sum up all fixed violations without differentiating among projects to investigate the distribution. Overall, we have identified 9,091 violation instances involving 214 types that developers have fixed. Besides, only 20% (43) of the violation types correspond to 75% (59,731) of the violations, and only 20% of the violation types are associated with 78% of fixed violations. This indicates that a few types correspond to most violations occurring and fixed, similar to previous studies [6], [7].

To explore the fix rate of different violation types, we first filter out those types with no more than ten violation instances to prevent too few samples from affecting the analysis. The overall distribution of the fix rate is shown in Fig. 3. The abscissa of the histogram is the fix rate, and the ordinate is the number of violation types in the fix rate interval. The only two violation types with an overall fix rate greater than 0.5 are *Unused "private" fields should be removed* and *A conditionally executed single line should be denoted by indentation*. Both two types are easy to understand and fix. It is worth mentioning that the main fixing pattern of type *Unused "private" fields should be removed* should be Code Deletion, but most of these violations were fixed by code changes, and a small part of these violations are closed by Code Deletion. This indicates that some developers will first complete the design of some fields in the development process and then use these fields in later development.

The fix rate of most violation types (i.e., 197, 92.5%) is less than 0.3, and 25 (7%) violation types have no fixed cases. We list the top 10 unfixed violation types by the number of instances as shown in Table III Nine violation types with no fixed cases belong to Code Smell, of which four are Critical. Most of these violation types are related to specific programming habits (i.e., *Control structures should use curly braces* and *Statements should be on separate lines*). Three of these violation types are related to *override* methods in class *Object*, which are used to implement a specific function. Only one type *Locks should be released (Locks)* in the category "Bug" that has no fixed cases. We manually examined ten violations of type *Locks*, all due to the inclusion of conditional judgments before the lock release, causing SonarQube to report the violation. However, because of the specific functionality, the developers do not want to fix those violations. An example is shown in Fig. 4, where the developer annotated the code as *No need to signal if timed out* to wrap the lock release code within a conditional judgment.

Among 25 violation types without fixed cases, 21 are Code smells, three are Bugs, and only one is Vulnerability. Although Code Smell accounts for half the number (i.e., 56.6%) of configured violation types, it has the highest number without

TABLE III
VIOLATION TYPES WITHOUT FIXED CASES

Violation types	occurrences	Category	Severity
Control structures should use curly braces	97	Code Smell	Critical
Statements should be on separate lines	97	Code Smell	Major
”switch” statements should not contain non-case labels	55	Code Smell	Blocker
”clone” should not be overridden	49	Code Smell	Blocker
Package names should comply with a naming convention	45	Code Smell	Minor
The Object.finalize() method should not be **overridden**	36	Code Smell	Critical
Fields should not be initialized to default values	27	Code Smell	Minor
Classes that **override** ”clone” should be ”Cloneable” and call ”super.clone()”	26	Code Smell	Minor
Locks should be released	26	Bug	Critical
Inner classes should not have too many lines of code	24	Code Smell	Major

TABLE IV
FIX RATE PER CATEGORY AND SEVERITY

Category	Info	Minor	Major	Critical	Blocker	Total
Code Smell	9%(633/6,683)	15%(3,264/21,153)	14%(3,592/25,745)	12%(1,072/9,312)	12%(73/602)	14%(8,634/63,495)
Bug	0	9%(91/977)	11%(232/2,183)	9%(11/123)	15%(29/191)	10%(363/3,474)
Vulnerability	0	0	0	5%(2/41)	9%(4/43)	7%(6/84)
Security Hotspot	0	18%(50/273)	0%(0/3)	8%(36/437)	11%(2/19)	12%(88/732)

```
public boolean enterWhenUninterruptibly(
      Guard guard, long time, TimeUnit unit) {
      ......
    } finally {
      // No need to signal if timed out
      if (!satisfied) {
        lock.unlock();
    }}
      ......}
```

Fig. 4. An unfixed violation purposefully ignored by a developer

fixed cases. The severity distribution is relatively even, and the number of severity levels from highest to lowest is 3, 9, 6, 6, 1. To further explore whether category and severity affect developer fixes to violations, we categorized the fixed violations according to their category and severity, as shown in Table IV. Vulnerability has the lowest fix rate, while Code Smell has the highest fix rate. However, in terms of purpose and positioning mentioned in Section II-A, Vulnerability should have a higher fix rate than Code Smell. Thus, we analyze the reasons for the difference in fix rates among different categories in Section IV-C2 and IV-C3. According to the fix rate of violations of the same category with different severity, the classification of severity shows ineffectiveness in fix rate. The violation fix rate does not increase with the severity, indicating much room for optimization of the default severity of violations. We do not compare the severity of different categories because different categories have different criteria for severity [30], and the comparison is meaningless.

2) RQ2-2: We choose violation types with no less than ten instances and show the distribution of the fix rate of each project in Fig. 5. It shows the violation fix rate is closely related to the project, and 29 (97%) projects' median violation fix rate is less than 0.4. In a small percentage of projects, all violation types have a fix rate of less than 0.2, but most projects have quite a few violations with a fix rate higher than 0.5. This indicates that the fix rate of the same violation type varies significantly between projects, considering that only two types are greater than 0.5 in sum-up data.

To explore the characteristics of high fix rate violation types in different projects, we inspect those fix rates as more than 0.5 in at least one project and have at least ten instances in total. As a result, there are 39 violation types in Code Smell, 4 in Bug, 1 in Security Hotspot, and none in Vulnerability. The top ten violation types per category are shown in Table V. The type *Delivering code in production with debug features activated is security-sensitive* mainly occurred in project shenyu from AFS. This type usually occurs during the test stage of development for debugging purposes and generally eliminates associated violations once the functionality is stabilized, so the fix rate is high. The first three types in Table V of the category Bug are easy to recognize and related to potential exceptions. The last type can cause erroneous data acquisition when enabling multi-thread. For all ten violation types in the category Code Smell, most belong to Minor, and the rest belong to Major and Info. These easy-to-recognize and fix violations tend to be fixed at a higher rate, given that ASATs are not used on some projects, and developers primarily tend to fix obvious ones. Besides, violation types associated with simplified code have a high fix rate. Since we are using some high-quality and popular projects, developers may be iterating and optimizing code continuously. To further explore the reason related to developers of high fix rates, we checked the main repairers of these violations. We found that individual developers fixed most project violations, indicating that the violation repair may have something to do with the developer's quality awareness and competence. For example, violations of type *Method names should comply with a naming convention* are fixed by only one developer in maven from AFS.

3) RQ2-3: To answer this research question, we only inspect violation types with at least ten instances same con-

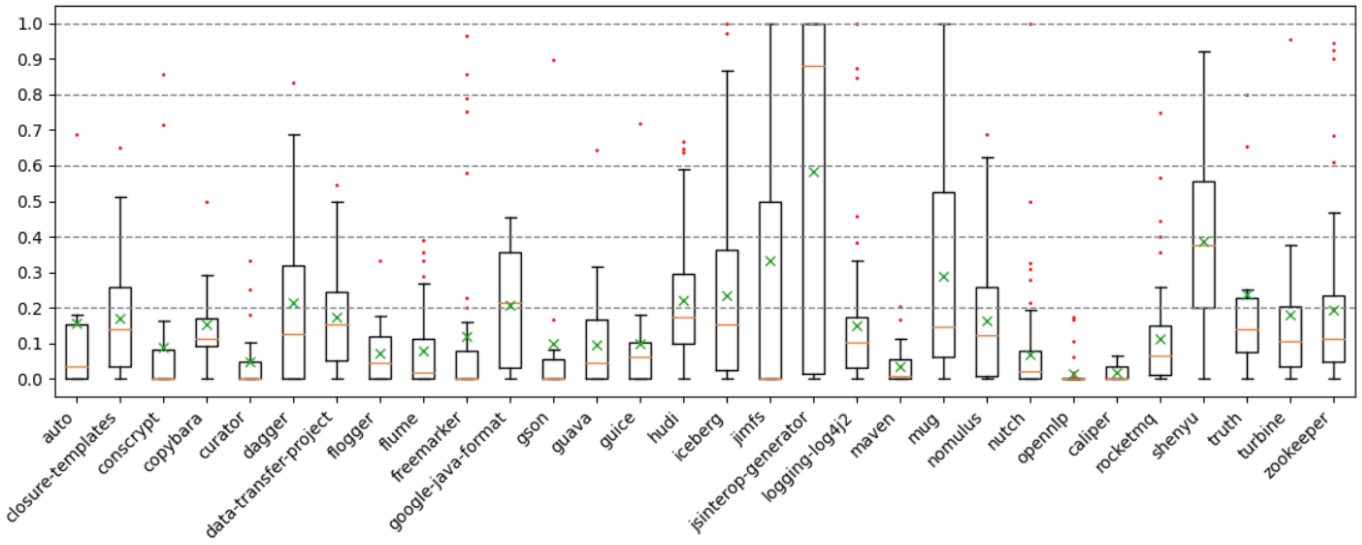

Fig. 5. Distribution of fix rate per project

TABLE V
THE HIGHEST FIX RATE OF VIOLATION TYPES AMONG PROJECTS

Violation types	Fix Rate	Projects	Category	Severity
Delivering code in production with debug features activated is security-sensitive	76.5% (26/34)	1	Security Hotspot	Minor
Jump statements should not occur in ”finally” blocks	70.0% (7/10)	3	Bug	Critical
Null pointers should not be dereferenced	64.9% (24/37)	8	Bug	Major
Optional value should only be accessed after calling isPresent()	63.6% (14/22)	2	Bug	Major
Non-primitive fields should not be ”volatile”	53.8% (7/13)	2	Bug	Minor
”public static” fields should be constant	100.0% (16/16)	1	Code Smell	Minor
Method names should comply with a naming convention	100.0% (16/16)	1	Code Smell	Minor
Static non-final field names should comply with a naming convention	97.4% (37/38)	3	Code Smell	Minor
Deprecated code should be removed	95.2% (20/21)	2	Code Smell	Info
Class variable fields should not have public accessibility	95.1% (39/41)	3	Code Smell	Minor
Array designators ”[]” should be on the type, not the variable	92.9% (92/99)	2	Code Smell	Minor
”@Override” should be used on overriding and implementing methods	92.6% (25/27)	4	Code Smell	Major
”entrySet()” should be iterated when both the key and value are needed	91.7% (11/12)	2	Code Smell	Major
Only static class initializers should be used	91.7% (11/12)	3	Code Smell	Major
Modifiers should be declared in the correct order	90.5% (218/241)	7	Code Smell	Minor

ditions in section IV-C2. The top 10 violation types with the lowest fix rates for each category are shown in the Table VI. We do not list the types of Code Smell, for these are listed in the Table III. Only two types of Vulnerabilities and seven types of Security Hotspot meet the above conditions. The two types of Vulnerability are challenging to recognize or understand for developers without knowledge of the relevant domain (i.e., XXE attacks). Besides, type *Server certificates should be verified during SSL/TLS connections* are not fixed in flume and nutch from AFS due to specific domains. Most of the violation types in Security Hotspots do not cause bad results without attacks and are more like a warning. Most violation types in Bug category are easy to understand, but it is difficult for developers to be aware of due to complex code logic if there is no help from tools. In addition, quite a few violations are false alarms. An example of *Zero should not be a possible denominator* that is not fixed due to a false alarm is shown in Fig. 6. The variable *denominator* cannot be zero but SonarQube reports this violation in the last line of the method (shaded in the figure).

```
static long multiplyFraction(long x, long numerator, long denominator) {
    ......
    long commonDivisor = gcd(x, denominator);
    x /= commonDivisor;
    denominator /= commonDivisor;
    // We know gcd(x, denominator) = 1
        // x * numerator / denominator is exact,
    // so denominator must be a divisor of numerator.
    return x * (numerator / denominator);
}
```

Fig. 6. An unfixed violation due to a false alarm

To explore why some of the violations of low fix rates were fixed, we manually examined the instances marked as fixed in our data set. We find the fact that a number of violations were fixed by one developer in one commit. It is a sign that some violation fixes were a side effect of code maintenance. The developer did not fix them intentionally but accidentally while maintaining the software. We also find that some of the fixed cases were not actually fixed but only inaccurate matches between revisions.

TABLE VI
THE LOWEST FIX RATE OF VIOLATION TYPES AMONG PROJECTS

Violation types	Fix Rate	Projects	Category	Severity
Server certificates should be verified during SSL/TLS connections	0%(0/16)	2	Vulnerability	Critical
XML parsers should not be vulnerable to XXE attacks	5.4%(2/37)	7	Vulnerability	Blocker
Using publicly writable directories is security-sensitive	5.4%(2/37)	13	Security Hotspot	Critical
Using pseudorandom number generators (PRNGs) is security-sensitive	6.3%(14/222)	15	Security Hotspot	Critical
Using slow regular expressions is security-sensitive	8.2%(6/73)	18	Security Hotspot	Critical
Expanding archive files without controlling resource consumption is security-sensitive	8.3%(1/12)	6	Security Hotspot	Critical
Using weak hashing algorithms is security-sensitive	8.7%(2/23)	11	Security Hotspot	Critical
Hard-coded credentials are security-sensitive	10.5%(2/19)	9	Security Hotspot	Blocker
Configuring loggers is security-sensitive	15.7%(11/70)	6	Security Hotspot	Critical
Locks should be released	0.0%(0/26)	2	Bug	Critical
Zero should not be a possible denominator	0.0%(0/17)	4	Bug	Critical
”volatile” variables should not be used with compound operators	0.0%(0/15)	5	Bug	Major
Loops with at most one iteration should be refactored	0.9%(3/335)	9	Bug	Major
Strings and Boxed types should be compared using ”equals()”	1.1%(2/188)	9	Bug	Major
”ThreadLocal” variables should be cleaned up when no longer used	2.8%(4/144)	17	Bug	Major
Blocks should be synchronized on ”private final” fields	3.4%(2/59)	12	Bug	Major
Return values should not be ignored when they contain the operation status code	3.7%(6/162)	17	Bug	Minor
”Random” objects should be reused	5.4% (2/37)	8	Bug	Critical
Overrides should match their parent class methods in synchronization	6.1%(2/33)	12	Bug	Major

We find very differnt fix rates for different violation types and find violation fix rates are mainly related to the following factors: (1) the difficulty of understanding rules and fixing the corresponding violations; (2) the harm of violations; (3) the context in which the violation resides; (4) the function and effect of rules; and (5) the development activities of specific developers.

D. Fix Time Analysis (RQ3)

To answer this research question, we investigate the fix time of violations. We filter out the violations introduced in the first commit for data accuracy since most of the projects were not analyzed from the initial commit. In other words, all subsequent statistics are based on consecutive commits, which is different from previous studies [6], [7]. Besides, we only consider violation types that occur at least ten times and have fixed cases. Overall, there are 154 violation types. 30%(1448) of violations are fixed within a month, and 85% are fixed within a year.

To explore the variations of violation fix time, we use the approach proposed by Giger et al. [31] for classifying the fix time, which is also used in previous studies [6]. This approach uses the median time to distinguish between fast (fix time $\leq$ median time) and slow status (fix time $>$ median time). We only select the violation types that were fixed more than ten (10) times in order to prevent any bias caused by few violation fix cases on the experiment.

Table VII shows the top-ten and bottom-ten violation types, sorted by median time of fixes. The column **Difference** is the top quartile fix time minus the bottom quartile difference. The smaller the difference, the more concentrated the violation fix. The column **Fixed** is the number of violations of the corresponding type that were fixed. Regarding categories, the nine types in Table VII that take the shortest fix time and the longest fix time belong to Code Smell. At the same time, the distribution of the severity of violations is relatively balanced. It indicates that the category and severity of violations are not directly related to the timeliness of violation fix. The difference between the upper and lower quartiles of the fix time of these violations is relatively large. Some violations are fixed in one day (e.g., type *Type parameter names should comply with a naming convention*), indicating that the timeliness of violation fixes is partly due to developer awareness or team management. Most violation types that take a short time to fix are easily fixed and related to coding habits. These violation types are marked by many development tools such as IntelliJ IDEA. In addition, the violation types that take a long time to fix are mostly trivial.

We analyze the distribution of violation fix time, 30% of violation instances are fixed within a month, and 85% are fixed within a year. In addition, we find that the factors affecting the timeliness of violation fixes are as follows: (1) the difficulty of understanding and fixing; (2) the importance of violation; (3) the hints of developing tools; and (4) the management of developers and teams.

V. DISCUSSIONS

A. Implications

In this subsection, we discuss some implications from the perspectives of developers, tool builders, and researchers.

Developers. This study identifies the violation types which developers fix and ignore. The results provide some suggestions for type configuration and violation repair. Regarding rule configuration, developers should not adopt the ASATs' default rule set and severity. Developers should configure specific rule sets and severity based on actual project needs and adopt those with high fix rates in the same area projects. Before software delivery or release, developers should configure rules related to development schedules to avoid leaving out

TABLE VII
TOP 10 VIOLATION TYPES WITH THE LONGEST AND SHORTED MEDIAN FIX TIME (DAYS)

Violation types	Median	Difference	Fixes	Category	Severity
Type parameter names should comply with a naming convention	453	0 (453-453)	107	Code Smell	Minor
Fields in non-serializable classes should not be "transient"	420	387 (485-98)	15	Code Smell	Minor
Unused method parameters should be removed	344	1058 (1091-33)	116	Code Smell	Major
Fields in a "Serializable" class should either be transient or serializable	321	252.5 (321-68.5)	26	Code Smell	Critical
Parentheses should be removed from a single lambda input parameter	313	261 (341-80)	56	Code Smell	Minor
Deprecated code should be removed	277	473.75 (500-26.25)	60	Code Smell	Info
Restricted Identifiers should not be used as Identifiers	261	336.5 (395.5-59)	101	Code Smell	Major
"throws" declarations should not be superfluous	253	799 (973-174)	11	Code Smell	Minor
Anonymous inner classes containing only one method should become lambdas	246	418.5 (434-15.5)	30	Code Smell	Major
Resources should be closed	232	709 (766-57)	18	Bug	Blocker
"for" loop stop conditions should be invariant	0	128 (128-0)	15	Code Smell	Major
Redundant casts should not be used	3	2 (3-1)	20	Code Smell	Minor
Public constants and fields initialized at declaration should be "static final"	4	0 (4-4)	16	Code Smell	Minor
Field names should comply with a naming convention	4	0 (4-4)	15	Code Smell	Minor
"@Override" should be used on overriding and implementing methods	5	139 (141-2)	21	Code Smell	Major
Methods and field names should not be the same or differ only by capitalization	5	6.25 (6.25-0)	16	Code Smell	Blocker
Printf-style format strings should be used correctly	6	248 (249-1)	33	Code Smell	Major
Abstract classes without fields should be converted to interfaces	9	143 (143-0)	17	Code Smell	Minor
Local variable and method parameter names should comply with naming convention	13	321 (322-1)	13	Code Smell	Minor
Synchronized classes Vector, Hashtable, Stack and StringBuffer should not be used	15	146.25 (152.25-6)	28	Code Smell	Major

unfinished functionality. The development team should agree on a common coding specification and configure the relevant rules. Developers should configure essential rules that can cause code exceptions. Concerning violation repair, developers can learn violation types to improve their ability and quality awareness to fix violations faster and better. Developers should configure static analysis plugins in development tools to fix violations early.

Tool Builders. Our findings in studying violation fix rate and fix time shed light on the areas where tool builders can benefit developers the most. We found that most unfixed cases for some violation types with high fix rates were due to false positives. Tool developers can use these cases to test automatic analysis tools to improve tool detection accuracy. Since the severity of the violation is not effective in fixing the violation, we recommend that tool developers investigate more effective default severity. Tool builders can recommend a more suitable default rule set based on the actual project domain since the fix rate and fix the time of different violation types vary significantly from project to project. In addition, tool builders can integrate violation tracing and violation ignoring capabilities into static analysis tools to provide developers with specific violation ignoring capabilities.

Researchers. This study highlights the need for researchers to develop better violation matching and tracking algorithms to identify closed violations more quickly and accurately. Besides, this study verifies the feasibility of using different code modification characteristics to distinguish fixed violations from closed ones. Developers can combine violation traceability with code modification characteristics to collect fixed violations. Based on the collected, fixed violations, developers can further study and identify the characteristics of actionable violations to make more accurate actionable violation predictions and technical debt analyses. Given recent advances in fixing pattern mining and automated program repair, we believe automatic remediation of violations is highly possible.

B. Threats to Validity

Threats to internal validity. The limitation of the underlying algorithms used in this study (i.e., violation matching and code differencing) is among the threats. Violation matching may produce false positive closed violations. The original algorithm does have quite a few mismatches and misses, especially for projects with a long history. Therefore, we create some heuristic algorithms in addition to adopting the algorithm improved by Li et al. [4] to reduce this threat. Code differencing algorithms are used to classify code modifications. Prior research [32] has shown that even developers familiar with the project often disagree on the origin of code sources, so there may not be an exact definition. Thus, we choose the same algorithm in the violation matching algorithm. In addition, the number of closed violations we validated in RQ1 is not very large. This is because such a manual validation is very time-consuming, involving understanding code changes and violation fixing patterns, verifying violations are actually closed, and data statistics. Therefore, we deliberately selected closed violation samples from both project and violation type dimensions and filtered the closed cases due to file deletion to generalize our conclusions as much as possible.

Threats to external validity. The adoption of SonarQube as the ASAT and configured violation types may threaten the validity of our study as each rule is specific to a particular problem. Nevertheless, SonarQube and other ASATs have quite a few overlapping violation types; while they have different names, their functionality is the same. For example, type *resources should be closed* in SonarQube aims to remind developers to close resources properly, for it may cause a resource leak. The same violation type in FindBugs is

OBL_UNSATISFIED_OBLIGATION, *CloseResource* in PMD and *memleak-resourceLeak* in TscanCode [33]. The research results are inevitably threatened by selected projects, as not all projects use SonarQube, and the sample of projects is analyzed to study the characteristics of fixed violations and which factors influence developers to fix violations, thus limiting the ability to generalize the conclusions to projects of different programming languages, ecosystems, or domains. However, the fact that 30 projects from ASF and Google are large, prevalent, and active, which have been studied, partially mitigates threats to generalization.

VI. RELATED WORK

A. Identification of Fixed Violations

Identifying whether a closed violation is intentionally fixed, so-called actionable violation, is a commonly reported research content in many violation analysis studies [6], [7], [11], [23]. Common identification strategies include finding text related to violation types from commit messages, tasks, or issue management platforms (i.e., Jira or Bugzilla) [23], [34], [35]. However, since there is usually not much relevant information about whether the project uses tools; many studies judge whether the violation is fixed by identifying the code modification characteristics when the violation is closed [11], [14], [15], [18], [19]. To identify actionable violations, many researchers [20], [21], [36]–[41] consider different aspects of violations to study the characteristics that are important in determining actionable violations. Wang et al. [18] conducted a systematic evaluation of all the available features and selected 23 "Golden Features" that were most effective for identifying actionable violations. Using these features, some studies [15], [19], [42] found that any machine learning technique, e.g., linear SVMs, can achieve good performance. However, Kang et al. [14] found that the strong performance of the "Golden Features" was primarily the reason for data leakage and data duplication issues. These issues are mainly caused by the inability to identify fixed violations accurately. Although violation matching can be used to determine the context in which a violation is closed, determining a fixed one requires careful context analysis. Most of the work [11], [14], [15], [18], [19] is based on file deletion and method deletion to determine whether the violation is fixed and does not verify the reliability of the results. Our work builds on previous work and goes further by analyzing the characteristics of code changes when a violation is closed and verifying whether a violation is fixed in multiple ways.

B. Empirical Studies on Fixed Violations

The most similar research came from Liu et al. [11]. They collect and track many fixed and unfixed violations across revisions of open-source Java projects to mine violation fixing patterns and then conduct an empirical study. We all analyze fixed violations, using the same violation matching algorithm proposed by Avgustinov et al. [9] and identifying fixed violations based on code modifications. However, Li et al. [4] proved the algorithm for matching violations inaccurate and only achieves 63.8% true positive in identifying closed violations. Thus, we used an improved algorithm to achieve 88.9% true positives in identifying closed violations [4]. Besides, we have proved that it is unreasonable to use code deletion and unrelated code modification, which are not filtered out by previous studies, to identify fixed violations for most violation types. The ASAT we use is SonarQube which is different from FindBugs, which they operate. SonarQube analyzes the source code directly, whereas FindBugs requires compiled bytecode. Due to the low compilation success rate of most open source projects [43], a complete violation evolution history is not easy to obtain. Considering the differences in completeness and accuracy of violations caused by the above factors, we focus on the commonality and difference of fixed violations from different types of fix rates, fix time, as well as fixed and unfixed cases to provide further enlightenment for different audiences. They investigate recurrences of violations as well as their fixing changes to provide insights into prioritizing violations and fixing patterns.

Our work intersects with the work of Marcilio et al. [6], and Digkas et al. [7]. Digkas et al. focus on the amount of technical debt repaid and the fixed violation types in the Apache Ecosystem. Marcilio et al. [6] focus on the developers responding characteristics of reported violations. Marcilio et al. surveyed the team leaders of the analyzed projects and found that most believed that ASAT violation messages were relevant to overall software improvements. The violation data used by Marcilio et al. comes from the private dataset and open-source software that used SonarQube. They both use SonarQube as the experiment tools to analyze violations and regard the closed violation as fixed, and the findings of their reported fixed violation distribution analyses were partially similar. For example, the violation fixing rate was low, with one-third of violations being fixed within one year, and a fraction of violation types corresponded to a large proportion of fixed violations. The difference between our study and theirs is twofold. First, we focus on the factor affecting the possibility of fixing violations and the reason for fixing a violation or not. Second, our data came from a complete consecutive violation history and screened out the violations that were actually fixed. Thus, our work can provide more reliable insights for fixed violations than these works.

VII. CONCLUSION

This paper presents a large-scale empirical study to characterize and understand the fixes of violations. We find empirical evidence to support identifying fixed violations from closed ones and understanding which violations are mostly and timely fixed. In the future, we plan to investigate how to configure the static analysis rules better to support effective violation detection and how to prioritize the detected violations according to the likelihood of actionable violations.

ACKNOWLEDGMENT

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

REFERENCES

[1] S. S.A, "Sonarqube," [Online], 2022, available: https://www.sonarqube.org/, Last Accessed on: April. 2022.

[2] P. 6.45.0, "Pmd source code analyzer," [Online], 2022, available: https://pmd.github.io, Last Accessed on: April. 2022.

[3] F. 3.0.1, "Findbugs," [Online], 2015, available: http://findbugs.sourceforge.net/, Last Accessed on: April. 2022.

[4] 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.

[5] 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

[6] 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

[7] 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.

[8] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. D. Penta, A. D. Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 403–414. [Online]. Available: https://doi.org/10.1109/ICSE.2015.59

[9] 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.

[10] 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.

[11] 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.

[12] R. D. Venkatasubramanyam and S. Gupta, "An automated approach to detect violations with high confidence in incremental code using a learning system," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 472–475.

[13] R. Bavishi, H. Yoshida, and M. R. Prasad, "Phoenix: automated data-driven synthesis of repairs for static analysis violations," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 613–624. [Online]. Available: https://doi.org/10.1145/3338906.3338952

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

[25] W. Miller and E. W. Myers, "A file comparison program," *Software: Practice and Experience*, vol. 15, no. 11, pp. 1025–1040, 1985.

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

[27] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, I. Crnkovic, M. Chechik, and P. Grünbacher, Eds. ACM, 2014, pp. 313–324. [Online]. Available: https://doi.org/10.1145/2642937.2642982

[28] B. Fluri, M. Würsch, M. Pinzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Software Eng.*, vol. 33, no. 11, pp. 725–743, 2007. [Online]. Available: https://doi.org/10.1109/TSE.2007.70731

[29] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[30] S. S.A, "Sonar rules," [Online], 2022, available: https://rules.sonarsource.com/java/RSPEC-2259, Last Accessed on: April. 2022.

[31] E. Giger, M. Pinzger, and H. C. Gall, "Predicting the fix time of bugs," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, RSSE 2010, Cape Town, South Africa, May 4, 2010*, R. Holmes, M. P. Robillard, R. J. Walker, and T. Zimmermann, Eds. ACM, 2010, pp. 52–56. [Online]. Available: https://doi.org/10.1145/1808920.1808933

[32] 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.

[33] Tencent, "Tscancode," [Online], 2017, available: https://github.com/Tencent/TscanCode, Last Accessed on: April. 2022.

[34] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. D. Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, J. M. González-Barahona, A. Hindle, and L. Tan, Eds. IEEE Computer Society, 2017, pp. 334–344. [Online]. Available: https://doi.org/10.1109/MSR.2017.2

[35] 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.

[36] 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

[37] 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

[38] 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

[39] 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

[40] 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

[41] 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

[42] 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

[43] C. Zhang, B. Chen, L. Chen, X. Peng, and W. Zhao, "A large-scale empirical study of compiler errors in continuous integration," in *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, M. Dumas, D. Pfahl, S. Apel, and A. Russo, Eds. ACM, 2019, pp. 176–187. [Online]. Available: https://doi.org/10.1145/3338906.3338917