Which Violations do Developers and Reviewers Concern During Code Review? An Empirical Study

***Abstract*—Code reviews and static analysis tools (SATs) can detect possible defects in a project without running the code. However, code reviews can be time-consuming and tend to miss static analysis violations that need fixing but are hard to spot. SATs are also underutilized due to various problems. Although there have been attempts to use SATs to aid in code review and studies to use code review to facilitate the use of SATs, there is still a gap in understanding and describing the fact how developers and reviewers concern code quality during code review. To fill this gap, we first develop a tool named Com2Vio to relate reviewers' code quality concerns to review comments. With this tool, we conduct a large-scale empirical study on 724,096 violations and 240,341 code review comments from 52 open-source Java projects in order to understand the concerns of the developers and reviewers during the code review process. Our findings help characterize and understand developers' and reviewers' considerations during code review and provide practical implications for developers, reviewers, and tool builders to aid code review and the SATs promotion.**

I. Introduction

Static code analysis is a technique that can be utilized to identify potential issues in a project at an early stage without executing the code. Code reviews, one of the methods of static code analysis, enable developers to evaluate defects, styles, and other criteria before integrating code into the codebase, ensuring project quality [1]–[3]. However, code reviews can be time-consuming for reviewers, which is why modern code review (MDR), a variation of the traditional code review process, often utilizes static analysis tools (SATs) to assist developers [2]. With the widespread use of MDR as a software engineering practice in open-source and industrial projects, the use and research of SATs in the review process are also emerging. Many studies [1], [4]–[9] have been recently conducted to explore the overall relationship and interaction between CR and SATs. They investigated the combination of SATs and the code review process to improve the efficiency of code review and explored the usage of review comments to promote the use and development of SATs.

Specifically, the code review process helps to improve the overall quality of software, but it consumes a significant amount of time and effort for reviewers. To reduce the human effort in code review, Balachandran et al. [8] proposed a tool that integrates SATs with the code review process. To further explore how SATs can help with code review, Panichella et al. [7] and Han et al. [1] conducted empirical studies focused on the fixes of coding convention during code reviews, but they treated all disappearing violations, including those closed by file or code deletion, as fixed, which may threaten the violation-related analysis [10], [11]. Additionally, they overlooked the fact that review comments are related to violations, which is of interest to reviewers. Therefore, some studies [6], [9] have associated and analyzed violations with review comments evaluating how SATs can help CR and configure SATs better. However, in these studies, either CRCs were associated with violations manually or were limited to associating code style violations with CRCs, determining the scale of the analysis and the widespread understanding of code quality. There is still a gap in using review comments to describe and understand reviewers' concerns about code quality.

From the perspective of using SATs, it can report many static analysis violations as a guardian of source code quality. However, adopting SATs is often challenged due to the lack of effective selection of violation rules or violation instances that developers are concerned about [12]–[17]. Coincidentally, a wealth of data reflects developers' and reviewers' concerns about code quality hidden in code modification and code review comments (CRCs) during code review, which can be used to promote the use and development of SATs. Therefore, some studies [9], [18] have used CRCs to explore the possibilities of SATs in detecting defects and configuring better rules. However, little is known about the code quality concerns of developers and reviewers during the code review, which are essential for promoting SATs and helping with code reviews.

To fill the gaps and explore the potential benefits of combining SAT and CR, we conducted a large-scale empirical study on 724,096 violations and 240,341 CRCs from 52 open-source Java projects hosted on GitHub. Specifically, we first proposed a tool named Com2Vio mapping CRCs and violations to understand reviewers' quality concerns regarding violations. We then investigated Com2Vio's effectiveness in mapping CRCs and violations detailed in Sec. IV-B (RQ1: Effectiveness Analysis). Com2Vio achieves over 0.852 precision and 0.883 accuracy, indicating that Com2Vio can effectively identify whether a violation is associated with CRCs. We used the violation tracking and fix identification approaches, which integrated into ViolationTracker [11], to collect violations of concern to developers. Finally, using Com2Vio and ViolationTracker, we collect the violations concerned by developers and reviewers to answer the following three research questions of this paper.

- RQ2: Developer's Perspective. Which violations do developers care for or not? (Sec. IV-C)
- RQ3: Review's Perspective. Which violations do reviewers care for or not? (Sec. IV-D)
- RQ4: Commonality and Difference. What are the common-

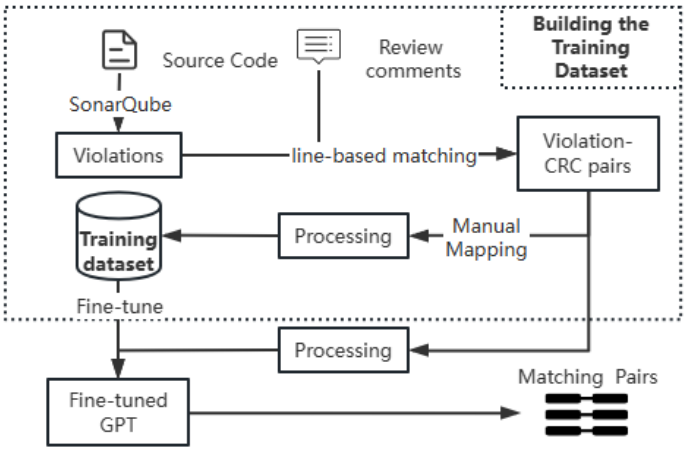


Fig. 1. An Overview of study method

ality and differences between the violations developers and reviewers focus on? (Sec. IV-E)

The major findings listed below are a summary of the above research questions. Developers tend to be more concerned about the quality of their code during code reviews and fix violations in their own submitted code. In addition, they tend to focus more on types that may cause bugs than those that don't affect code understanding. Reviewers are concerned not only with code quality at a high or abstract level but also with code reusability, comprehensibility, and correctness, ignoring deficiencies in canonical classes and best practices. For those reviewers concerned, however, existing violations are mainly caused by the following aspects: the developer's lack of detailed annotation to explain code implementation considerations; reviewers' misunderstanding and lack of double-checking for the revised code; false positives reported by AST and the lack of cross-file analysis ability.

Our findings provide empirical evidence and insights for developers and reviewers regarding their concerns and non-concerns with violations, thus better understanding each other for promoting code view and collaboration. Our findings also provide cases and capability requirements for tool builders to facilitate the development and usage of SATs.

In summary, this work makes the following contributions.

- We proposed a tool that can associate violations with CRCs and released a large-scale dataset of manually-mapped CRCs onto violations.
- We conducted a large-scale empirical study to understand the the violations concerned by developers and reviewers during code review.
- We provided practical implications of our findings to three audiences: developers, reviewers, and tool builders.

II. Com2Vio

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

The overall workflow of Com2Vio is depicted in Figure 1. The process commences with collecting raw data from existing datasets and GitHub[1], which includes source code and code review comments. The CRCs are then preprocessed, and SAT is employed to analyze the source code and identify violations. Additionally, the code comments are processed and represented in a suitable format. Next, a manual annotation process is carried out to determine the relationship between CRCs and violations. Finally, the manually annotated dataset is utilized for training the model for automatically matching CRCs with violations. In the subsequent sections, we provide a detailed description of each step.

B. Raw Data Collection

In this step, the objective is to gather the source code to be reviewed (we call it *review code* for short) and the review data. In addition, we also collected *original code*, which is the version before the review code, to explore what the developers fixed previous violations in the review code, and *revised code*, which is the version after the review code, to analyze which violations were fixed and which remained after the review. Currently, Paixao et al. [19] provide a dataset called CROP with eight projects and 48,975 comments, which is not large. Furthermore, CROP only provides the code of two versions, the review and the revised versions; the original code version is missing. Therefore, to obtain all versions during code review, increase the universality of our method and reduce the impact of data, we plan to crawl raw data from GitHub based on the pull request model. Developers submit code for merging, and the repository management conducts code reviews and provides corresponding comments [2], [20]–[22]. Therefore, we prioritize selecting high-star projects with many pull requests to obtain as much source code with review comments as possible. We started by selecting high-quality projects from GitHub, mainly from Apache Software Foundation (ASF) and Google, similar to violation-related research [9]–[11], [17], [23]–[25]. This ensures that projects include diverse changes and rich code review information so that the data or conclusions analyzed from these projects can be widely accepted and have a high impact by the majority of developers and reviewers. We then utilized the GitHub API [2] to crawl the corresponding repositories' source code and review information.

C. Pre-Processing

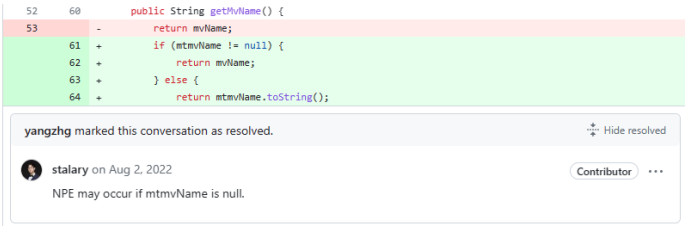
In this step, we conducted data preprocessing for the purpose of violation collection and CRCs representation. First, we employed SAT to analyze the changed files involved in code review and obtain corresponding violations. Then, the extracted CRCs from Github and the rules, so called violation types, and details were subjected to preprocessing. Finally, we characterized and stored the processed rules, details, and

[1]https://github.com/

[2]https://docs.github.com/en/rest?apiVersion=2022-11-28

```
ProjectName: apache/doris
PullRequestId: 11218
SV: review version
EV: revised version
FilePath: fe/fe-core/src/main/java/org/apache/
    doris/analysis/DropMaterializedViewStmt.java
Type: Null pointers should not be dereferenced
Detail: A "NullPointerException" could be thrown;
    "mtmvName" is nullable here.
Locations: Location1:61-61; Location2:64-64
```

(a) An example of a two-location violation detected in review version and closed in revised version in file DropMaterializedViewStmt.java



(b) An example of a CRC marked in file DropMaterializedViewStmt.java

Fig. 2. An example of violation-related-CRC pairs from project doris

code reviews. We used SonarQube, one of the most commonly used open-source SATs in academia and industry [15], to detect violations in the repositories. The SonarQube version was 8.9.1, with 627 Java rules covering four categories: code smells, bugs, vulnerabilities, and security hotspots. In this paper, we used the default 452 rules recommended by SonarQube, which usually apply to most projects. Figure 2(a) shows an example of a violation with two locations in the collected dataset. To process the CRCs, violation types, and details, we adopted a similar method by Zampetti et al. [9]. We employed stop word removal and stemming to process the violation types, violation details, and CRCs. There is a little bit of difference for violation types processing; we retained stop words as much as possible to maintain the semantic originality of the types. To represent the violation instances, we adopted a method proposed by Yu et al. [10], [11]. We recorded it as a octuple, as shown in figure 2(a). In order to depict CRCs, we utilize a six-tuple structure including the pull request ID, the file, the starting and ending line numbers, the comment content, and the reviewer.

D. Labeling Process

The primary objective of the labeling process is to manually establish associations between violations and CRCs to construct a benchmark dataset. We first employed a line number-based matching rule to filter out potentially relevant violations and CRCs to reduce the workload associated with manual review. Specifically, CRCs in several lines near the violation location may be related to the violation. Here, we first collect the possible correlation pairs of the violation and CRC in the same line to collect the data to be marked. Figure 2 shows an example of a violation-related CRC in the same line. By employing the line number-based method, we collected a total of 4380 line-related pairs for violations and CRCs from 38 projects. We then commence the manual labeling process by matching each violation instance with relevant CRCs. To determine the relationship between the two, each author, who has extensive experience in Java development and a thorough understanding of SonarQube's rules, independently judges whether the types or detailed information described in the code review comment correspond to those of the violation. In case of a disagreement between annotators, a third author with seniority in the matter issues a final conclusion. To facilitate the manual labeling process, we designed a web-based platform that displays data and documents the manual labeling information. Finally, we manually marked 4380 violations and CRCs line-related pairs. In total, 1406 pairs were violations associated with CRCs, and the remaining were not. To measure the degree of consistency between annotators with regards to the violation-to-CRC mapping, we compute Cohen's Kappa, which yields a value of 0.83, indicative of strong consistency [26].

E. Mapping CRCs and Violations

In this phase, we elucidate the process of automating the association of CRCs with violations. To the best of our knowledge, Com2Vio's most relevant tool is Auto-SCAT [9]. Unlike Auto-SCAT, which uses text similarity and activation thresholds to determine whether to activate a new project's checks, Com2Vio associates CRCs with violations based on the type and details of the violations. It determines whether the CRCs and violations express the same meaning, essentially a natural language processing (NLP) problem. Therefore, based on the current state-of-the-art NLP model, i.e., BERT [27] and GPT-3 [28], we trained them with labeled data to automate map CRCs and violations. We used 80% of the manually labeled data as the training set and the remaining 20% as the test set. The model training and data details are provided on an anonymous website[3]. We also experimented with conventional text-matching techniques, i.e., SimHash and cosine similarity, and other deep learning models. However, the results derived from these approaches did not prove satisfactory. Further discussion on the effectiveness of matching CRCs with violations is presented in Section IV.

III. Experiment Setup

A. Study Design

We aim to characterize and understand the commonalities and differences between the quality concerns of developers and reviewers during the code review process. In pursuit of this goal, we have formulated four research questions in Sec I.

RQ1 seeks to investigate the accuracy of the Com2Vio for matching CRCs with violations. The outcomes from addressing RQ1 can offer empirical evidence to support the data presented in RQ3 and RQ4. In this study, we investigate RQ2 from the developers' perspective by examining their focus on violations during the code submission process for

[3]https://sites.google.com/view/com2vio

review. Similarly, we approach RQ3 from the perspective of code reviewers, examining their concerns regarding violations during the code review. Furthermore, we aim to explore commonalities and differences in the focus on violations or code quality between developers and reviewers (RQ4). The results of this investigation can foster better mutual understanding and communication between developers and reviewers, promoting effective software development.

We constructed our dataset through a three-step process, which includes the collection of raw data, tracking and collecting violations, and associating violations with code review comments. We collect source code and CRCs, detect violations the same as in the Sec. II-B.

B. Collecting and Tracking Violations

We analyzed the source code of three versions: the original code, the code submitted for review, and the revised code. After analyzing the code of each version, we proceeded with matching and tracking violations. Violation tracking involves matching similar instances of violations between project revisions. SATs typically report the positions of violations in terms of line numbers. However, in many cases, code changes can also move and modify the violation locations, matching a non-trivial task. Inaccurate matching can result in numerous false positives and negatives when identifying newly introduced and closed violations, affecting the analysis and understanding of identified violations [11], [29], [30]. Therefore, previous research on violation tracking [29], [31]–[34] has primarily focused on the problem of matching violations using code matching techniques and software change history to match the code and its contextual code where violations are located. The state-of-the-art method proposed by Yu et al. [11] has been integrated into a tool named ViolationTracker, which uses an anchor-based heuristic violation similarity matching strategy to match and track violations with high accuracy. Moreover, since ViolationTracker is open-source, we utilized it in this study to identify the introduced and closed violations.

C. Distinguish between Violations of Concern

After the violations are collected and matched, we begin to identify violations that are of concern to developers or reviewers. To identify the violations concerns of the developers, we used the similar approach as in the previous study: to distinguish fixed violations from closed ones [17], [33]–[37]. Determining whether closed violations are intentional fixes, so-called actionable violations, is a common research topic in many studies [11], [17], [37]–[39]. The current state-of-the-art method proposed by Yu et al. categorizes code modification characteristics when violations are closed into 5 types: file deletion, code deletion, anchor deletion, related code modification, and unrelated code modification [10]. They found that related code modification, and unrelated code modification characteristics can be used to determine whether a violation has been truly fixed. In this paper, we consider the violations fixed by the developers as the ones they are concerned about, while the existing violations are considered the ones they neglect or disregard. To identify violations of interest to reviewers, previous research has shown that violation types extracted from CRCs are of interest to reviewers [9]. Therefore, we first construct the matching pairs of violations and CRCs. To handle the case where CRCs and violations are not on the same line but still relevant, we collect the CRCs on the three lines above and below the line where the violations are located to build the matching pairs to be processed. [4] Then, we use Com2Vio to identify the association between the CRCs and violations. Finally, we consider the violations associated with CRCs as the ones that reviewers are concerned about, while those not associated with CRCs are considered the ones they neglect.

D. Data Preparation and Evaluation Criteria

Since the majority of previous work has been conducted in Java projects, and the implementation of Java violation matching algorithms is relatively mature, we have selected Java as the primary language for our project. Additionally, we require projects with various code changes and rich code review comments to generalize our conclusions as much as possible. Therefore, we randomly selected 52 high-star projects mainly from ASF and Google, which are recently active, have a rich collaboration history, i.e., containing at least 50 pull requests, and are of substantial project size. Table I shows the basic information about the projects we collected.

For RQ1, we evaluated the matching status of CRCs and violations against 20% of the unused benchmark datasets. For each matching status, we define the precision (P), recall (R), Accuracy (A), and F1-score (F1) in relation to a specific matching pairs (S), as follows:

- $P = TP/(TP + FP)$;
- $R = TP/(TP + FN)$;
- $F1 = 2PR/(P + R)$;
- $A = (TP + TN)/(TP + NP + TN + FN)$;

TP is the number of violation-CRC pairs correctly identified as related, and FP is the number of unrelated violation-CRC pairs identified as related. TN is the number of violation-CRC pairs correctly identified as unrelated, and FN is the number of related violation-CRC pairs identified as unrelated.

To the best of our knowledge, there are no other existing automated methods for directly associating CRCs with violations. The most similar method to Com2Vio is the Auto-SCAT tool proposed by Zampetti et al. [9], which uses CRCs and text in a pre-collected knowledge base for rule selection, but with a different purpose to ours. Therefore, we adopt a variant of the method used in Auto-SCAT, which associates CRCs with violation types and details by judging the text similarity between them. We also attempt to fine-tune the BERT and Sentence-BERT [27], [40] model as an alternative approach to evaluate the effectiveness of the association between CRCs and violations.

[4] We have not observed any relevant CRCs beyond three lines in practice; hence, we choose three as the threshold for line relevance for efficiency in matching.

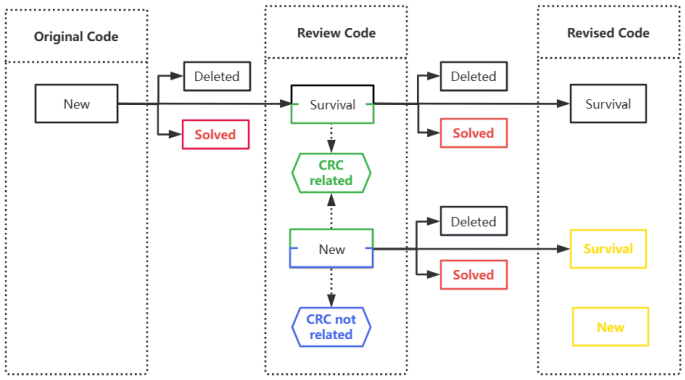


Fig. 3. The evolution of the violations

TABLE I
OPEN-SOURCE PROJECTS USED IN THIS STUDY

Projects	52
Pull Requests	11,258
Files (Java)	13,255
CRCs	240,341

TABLE II
STATISTICS ON VIOLATIONS OF THREE VERSIONS (TYPE)

Version	Total	NEW	CLOSED	SOLVED
Original	214,696 (281)	–	–	–
Review	257,866 (298)	54,756 (277)	11,586 (210)	4,571 (156)
Revised	251,534 (294)	7,178 (184)	13,510 (229)	3,570 (160)

Figure 3 shows the evolution process of violations in the three versions of a pull request. The data of RQ2, 3, and 4 in this paper are color-coded in the figure. The dotted boxes in the figure show the violations in each version, and the solid lines show the evolution of the violations. *New* indicates that the violation occurred for the first time in the version, *Survival* indicates that the previous violation still exists in the version, *Deleted* indicates that the violation closed because of deletion, and *Solved* indicates that the violation closed because of repair. Regarding RQ2, the violations that concern developers come from two parts: one is the violations introduced in the review version and repaired in the revised code, and the other is the violations that existed in the original code but were later fixed (red labeled). The violations ignored or neglected by developers also come in two parts. One is the violations introduced in the code of the review version but still exist in the revised version, and the other is the violations newly introduced in the revised version (yellow labeled). Since we cannot determine whether developers can notice the violations introduced in the original code but still exist in the revised version, we do not consider these violations the ones developers neglect. For RQ3, we select the violations that appear in the code of the review version and are associated with CRCs to explore the violations that reviewers concern about (green labeled). The violations ignored by reviewers are those newly introduced in the code of the review version but are not associated with CRCs (blue labeled). We do not consider the violations that existed in the original code but are unrelated to CRCs because we cannot determine whether reviewers noticed these original violations during the review process. For RQ4, the data is based on the intersection, union, and difference of the data in RQ2 and RQ3 to explore the commonalities and differences between the violations that developers and reviewers concern about.

IV. RESULTS

A. Statistics on Violations and CRCs

In total, we detected 724,096 violations involving 298 SonarQube rules across 52 projects. By applying violation tracking methods, we identified 276,630 distinct violations. Additionally, across all projects, there were 11,258 pull requests (PRs) containing 240,341 comments in 80,220 conversations. With the application of Com2Vio, 2,540 violations were associated with CRCs involving 161 rules, indicating that most of the rules reviewers were connected.

For each PR, we analyzed three versions, with the violation data for each version presented in Table II. The column *Total* indicates the total number of detected violations for the specific version, *NEW* represents the number of newly detected violations relative to the previous version, *CLOSED* indicates the number of violations closed in the version, and *SOLVED* represents the number of closed violations that were resolved (i.e., of interest to developers). The value in parentheses indicates the number of related SonarQube rules. The – indicates no corresponding data for this version. Overall, the developers closed/fixed quite a few violations in the original code but introduced many violations (i.e., 54,756) in the review code. After the review, the number of violations decreased, and developers closed nearly twice as many violations as they introduced. Consider that in the previous study [24], the violation close rate was only 8.77% on projects that actually used SonarQube, whereas here, the violation close rate was 24% during code review. In this sense, developers tend to fix more violations during code reviews and pay more attention to code quality than usual.

B. Effectiveness Analysis (RQ1)

Table III presents each approach's precision, recall, accuracy, and F1 score. Overall, the GPT-based approach outperformed the other methods regarding the F1 score and accuracy. This may be attributed to the GPT-based model's ability to precisely capture the semantic meaning of violation detail

TABLE III
EFFECTIVENESS OF DIFFERENT APPROACHES

Approach	P	R	F1	A
Com2Vio(BERT)	0.777	0.797	0.787	0.846
Com2Vio(SBERT)	0.769	0.772	0.770	0.836
Com2Vio(GPT)	0.852	0.813	0.832	0.883
Auto-SCAT(Text similarity)	0.584	0.693	0.634	0.715

TABLE IV
STATISTICS ON VIOLATION CATEGORIES FROM DEVELOPER PERSPECTIVE

Ca	Concern		Ignore		Default
	Instance	Type	Instance	Type	
CS	7,103 (87.2%)	127 (70.6%)	45,173 (90.5%)	184 (67.6%)	256 (56.6%)
B	642 (7.9%)	43 (23.9%)	2,544 (5.1%)	67 (24.6%)	140 (31.0%)
SH	394 (4.8%)	9 (5.0%)	2150 (4.3%)	14 (5.1%)	29 (6.4%)
V	2 (0.0%)	1 (0.6%)	39 (0.1%)	7 (2.6%)	27 (6.0%)

TABLE V
VIOLATIONS FOCUSED OR IGNORED BY DEVELOPERS

Rule	O/F	Ca
Inappropriate "Collection" calls should not be made	8/5	B
Return values from functions without side effects should not be ignored	6/4	B
Boxing and unboxing should not be immediately reversed	21/13	B
"catch" clauses should do more than rethrow	48/24	CS
Double-checked locking should not be used	14/6	B
Zero should not be a possible denominator	20/7	B
Exception types should not be tested using "instanceof" in catch blocks	12/5	CS
Redundant casts should not be used	68/28	CS
"toString()" should never be called on a String object"	30/11	CS
Null checks should not be used with "instanceof"	46/18	CS
Unnecessary imports should be removed	2,519/0	CS
Static non-final field names should comply with a naming convention	1,637	CS
Unused "private" fields should be removed	1,408/0	CS
"public static" fields should be constant	1,277/0	CS
Class variable fields should not have public accessibility	953/0	CS
Non-primitive fields should not be "volatile"	584/0	B
Private fields only used as local variables in methods should become local variables	450/0	CS
"Public constants and fields initialized at declaration should be "static final" rather than merely "final"	178/0	CS
Methods and field names should not be the same or differ only by capitalization	144/0	CS
Classes from "sun.*" packages should not be used	137/0	CS

information and CRCs. The text similarity-based approach was the least effective of all approaches. One reason for this difficulty may be that it cannot accurately capture semantic meaning, making it difficult to cover most cases through similarity threshold configurations.

The Com2Vio (GPT) achieves an accuracy of 88.3%, indicating that Com2Vio(GPT) can effectively identify whether a violation is associated with CRCs.

C. Developer's Perspective (RQ2)

1) Overall statistics: The developers fixed 8,141 violations (Related Code Modification and Unrelated Code Modification), half of which (i.e., 13,068, 52%) were introduced in the original code, indicating that developers were concerned about violations in the original code. Of the 3,570 violations fixed in the accepted code, 3,160 were introduced in the review code, indicating that developers are more concerned about violations in their committed code during the review process. Considering that 54,756 new violations were introduced in the review code, which also shows that most of the violations were either unconcerned or unaware of.

To understand the rules developers focus on, we first analyzed the overall category distribution of violations they focus on and ignore. Table IV shows the distribution of violation categories in terms of the number of violations and rules, with the overall proportion in parentheses. The first column is Category (C), and there are four of them: Code Smell (CS), Bug (B), Security Hotspot (SH), and Vulnerability (V). There is little difference in the distribution of categories that developers' concerns and non-concerns regarding the proportion of violations and rules. Regarding the number of violations and the number of related rules, developers are more focused and more likely to ignore violations in the Code Smell category. Regarding the proportion of instances and rules for the same category, developers pay more attention to and tend to ignore violations in the Bug category, followed by Security Hotspot. For example, the Bug category accounts for 7.9% of instances, while 23.9% of rules are involved. Both Bugs and Security Hotspot are related to the correct program operation and therefore are of greater concern to developers.

2) Specific rules: To further investigate the rules that developers focus on, we used the fluctuation ratio method proposed by Liu et al. [30] to evaluate the degree to which developers focus on different rules. The fluctuation ratio assesses the difference in ranking between introducing and fixing a given rule. Specifically, we define the total set as violations that developers concerns and non-concerns, with ALL representing the number of all violations and ALL_s being the total number of solved violations, which are developers' concerns. Given a type T, T_i is the number of type T violations, and T_s is the number of fixed. The fluctuation ratio T_f of type T is calculated as follows: $T_f = \frac{T_s}{ALL_s} \div \frac{T_i}{ALL}$. The larger the T_f value, the more developers focus on that rule, with a higher fix and lower occurrence rates. It is worth noting that, according to the calculation method of volatility, when the fix rate (T_s/T_i) of violations is the same, the fluctuation ratio is also the same. Therefore, when the rules have the same fluctuation ratio and a fixed count greater than 0, we consider those with higher fixed counts to be the rules that developers are more concerned about. When the number of fixes for a rule is 0 (and therefore, its volatility is also 0), we consider the rule with higher occurrences to be those developers are more likely to ignore.

Table V lists the top ten rules developers focus on (top half) and the top ten rules they ignore(bottom half). The *O/F* column represents the number of violation occurrences (O) and fixes (F) regarding the rules. Of the top ten rules developers care about, half are belong to Bug, and the others are Code Smell, and developers care more about rules of Bug than Code Smell. The number of rules in the Bug category is much lower than in Code Smell, indicating developers are more concerned about program correctness than maintenance violations during code review. This is different from the conclusions of previous studies [10], [23], [24], [30], most of which have found that developers are not more concerned with Bugs. Regarding

TABLE VI
VIOLATIONS FOCUSED BY DEVELOPERS AMONG PROJECTS

Rule	O/F	P	Ca
String literals should not be duplicated	1,170/246	19	CS
Delivering code in production with debug features activated is security-sensitive	274/78	19	SH
Sections of code should not be commented out	278/67	17	CS
The diamond operator (<>) should be used	509/133	15	CS
Track uses of "TODO" tags	1,186/207	14	CS
Null pointers should not be dereferenced	346/108	14	B
Unused assignments should be removed	438/110	13	CS
Generic exceptions should never be thrown	377/48	12	CS
Unused local variables should be removed	142/45	12	CS
Instance methods should not write to "static" fields	47/17	8	CS

TABLE VII
VIOLATIONS FOCUSED OR IGNORED BY REVIEWERS

Rule	O/C	Ca
Exceptions should not be created without being thrown	1/1	B
Interface names should comply with a naming convention	2/1	CS
"Stream" call chains should be simplified when possible	2/1	CS
Classes should not be empty	15/6	CS
Class names should not shadow interfaces or superclasses	3/1	CS
"toString()" and "clone()" methods should not return null	3/1	B
Variables should not be self-assigned	3/1	B
Related "if/else if" statements should not have the same condition	3/1	B
Loops should not be infinite	16/5	B
Constants should not be defined in interfaces	14/4	CS
Formatting SQL queries is security-sensitive	112/0	SH
"Iterator.next()" methods should throw "NoSuchElementException"	52/0	B
Methods returns should not be invariant	51/0	CS
Arrays should not be copied using loops	47/0	CS
"static" members should be accessed statically	39/0	CS
Exceptions should not be thrown in finally blocks	38/0	CS
A field should not duplicate the name of its containing class	38/0	CS
Labels should not be used	33/0	CS
"@Override" should be used on overriding and implementing methods	32/0	CS
Functional Interfaces should be as specialised as possible	31/0	CS

rules, two of the rules in Code Smell are related to exception handling, and the rest are related to code simplification. Of the violations that developers don't care about, only one is in the Bug category, i.e., Non-primitive fields should not be "volatile," which is related to multi-threading but has little impact on the developer's productivity. For the others in the Code Smell category, most of these are related to naming and maintenance and are generally not difficult to understand, while some are specific functionality implementations. To further analyze why developers do not fix these violations, we reviewed the source code and found that the original code had the same types of violations, so there may not be uniform rules in the original project, and developers are used to ignoring these violations.

3) Among projects: To explore the characteristics of violations that developers focus on in different projects and to prevent data bias caused by extensive violation data in individual projects, we selected the top 30 rules based on their fluctuation ratio in each project. Then, We calculated the number of projects in which these rules appeared. Finally, we ranked these rules by the number of projects among the top 30 most focused-on violations.

Table VI lists the rules developers focus on the most in different projects. Column *P* indicates that the rule belongs to the top 30 rules in how many projects. Rule *Null pointers should not be dereferenced* of Bug category and *Delivering code in production with debug features activated is security-sensitive* of Security Hotspot category have received attention in many projects. Both rules belong to Common Weakness Enumeration[5], indicating that developers in many projects also pay attention to security problems. Of the violations in the Code Smell category, most are related to the development process. For example, the fix pattern of type *Unused local variables should be removed* recommended by the SonarQube is code deletion, but here the developer solved it by code changes, indicating that the developers later used these variables. The rest rules relate to code simplification and optimization, given that we used high-quality and popular projects, indicating that developers may be iterating and optimizing code continuously.

[5]https://cwe.mitre.org/

TABLE VIII
VIOLATIONS FOCUSED BY REVIEWERS AMONG PROJECTS

Rule	O/C	P	Ca
Sections of code should not be commented out	1,051/170	23	CS
Delivering code in production with debug features activated is security-sensitive	266/45	20	SH
Track uses of "TODO" tags	953/99	18	CS
Deprecated code should be removed	330/45	13	CS
Unused local variables should be removed	207/31	13	CS
Generic exceptions should never be thrown	585/31	13	CS
Unused "private" fields should be removed	616/94	11	CS
Unnecessary imports should be removed	540/80	11	CS
Standard outputs should not be used directly to log anything	255/62	11	CS
Unused "private" methods should be removed	206/35	11	CS

> Developers tend to be more concerned about the quality of their code during code reviews and tend to fix violations in their own submitted code. Developers tend to focus more on security-related violations, which may cause bugs, than those that don't affect code understanding. In addition, developers tend to iterate and continuously optimize their code on high-quality projects.

D. Review's Perspective (RQ3)

1) Specific rules: Concerning RQ3, We first analyzed the overall distribution of the categories of reviewers' concerns and non-concerns. We found little difference between reviewers' and developers' overall distribution of concerns and non-concerns regarding violation categories. To analyze the

reviewers' attention to different rules, we cannot use the fluctuation ratio proposed by Liu et al. [30] to measure since reviewers do not fix the violations. We adopt a new metric T_r to measure. The calculation formula is as follows: $T_r = T_c \div T_a$. Here, T_c is the number of CRCs associated with type T. T_a is the sum of T_c and the number of type T violations introduced in the review code but not associated with CRCs. When T_r values are the same and greater than 0, i.e., type T is associated with at least one CRC, we select the rule with a higher value of T_c as the one that reviewers are more concerned about. If T_c is 0, i.e., type T is unrelated to any CRCs, we select the rule with a higher value of T_a, i.e., more occurrences of the violation, as the one that reviewers are more likely to ignore.

Table VII displays the top 10 rules that the reviewers paid the most attention to and those that the reviewers most easily ignored. The O/C column represents the number of violation occurrences (O) and related CRCs (C) regarding the rules. The rules which the reviewers focus on are half Bug and half Code Smell. Although these violations occur only a few times, they are mentioned highly in CRCs. These rules are more concerned with the code quality from a high or abstract level. For example, four rules are related to class or interface. In addition, reviewers are also concerned about code reusability, comprehensibility, and correctness. Three rules are usually caused by mistakes or copy-paste errors, which are easily ignored by developers, i.e., the type *Related "if/else if" statements should not have the same condition*. Code smells dominate the categories of violations that reviewers ignore, most of which are code specifications or best practices that have no impact on the code works and understands it. Some violations are easy to spot, yet reviewers don't mention them. To further examine why reviewers missed these easy-to-spot violations, we manually reviewed the code and found that most of the violations were false positives or didn't require attention. For example, in file *hbase-server/src/main/java/org/apache/hadoop/hbase/master/HMaster.java* of Project Hbase[6], there are several violations of type *Methods returns should not be invariant*. Still, the comments on the code associated with these violations explain the code implementation considerations. One of the comments is *"maybe this will not happen any more, but anyway, no harm to add a check here."*. This shows that SonarQube's ability to analyze violations in specific scenarios has yet to be improved.

2) Among Projects: Many rules in Table VII appear less frequently, but the correlation rate is relatively high, which may be affected by individual projects or reviewers. To explore the characteristics of violations that reviewers focus on in different projects and to prevent data bias caused by extensive violation data in individual projects, we selected the top 30 rules based on T_r in each project. Then, we calculated the number of projects in which these rules appeared. Finally, we ranked these rules by the number of projects among the top 30 most focused-on violations, as shown in Table VIII. Only one of these violations falls into the Security Hotspot category, while the rest fall into the Code Smell category. However, two Code Smells, *Generic exceptions should never be thrown*[7] and Standard outputs should not be used directly to log anything[8], are also associated with security vulnerabilities, indicating that many project reviewers are concerned about software security and the protection of sensitive information. Furthermore, half of these violations relate to *unused*, indicating most project reviewers are concerned about code maintainability and possible technical debt.

> Reviewers are concerned not only with code quality at a high or abstract level but also with code reusability, comprehensibility, and correctness, ignoring deficiencies in canonical classes and best practices.

E. Commonalities and Differences (RQ4)

For RQ4, we used similar data from RQ2 and RQ3, i.e., the top 30 most concerned and ignored rules by developers and reviewers, to investigate commonalities and differences from different perspectives. We used R_c (D_c) to represent the 30 most concerned rules for reviewers (developers) and R_i (D_i) to represent the 30 most ignored.

1) Commonality: There are nine rules for the intersection of R_c and D_c, which both reviewers and developers are concerned about. Most of these types are related to code simplification or logic optimization, such as type *Null checks should not be used with "instanceof"*. In addition to focusing on exception handling related rules, they are also concerned with code readability and understandability, taking into account the need for reviewers to understand the intent of code changes. In addition, we found that some of the violations that reviewers and developers were concerned about survived in the accepted code. We further examined the reasons for their retention and found that misstatements in specific areas by SonarQube were to blame. For example, developers rarely fix type *Loops should not be infinite* violations in code related to thread wait and wake up, asynchronous processing, and so on, indicating that SAT detection capabilities in this area are still lacking.

There are seven for the intersection of R_i and D_i, that is, rules that neither reviewers nor developers care about. Both developers and reviewers tend to ignore those violations that might require a deep understanding of a specific knowledge area, i.e., *Server certificates should be verified during SSL/TLS connections* requires one knows network security well. Type *"Preconditions" and logging arguments should not require evaluation* requires one to know the limitation of using com.google.common.base.Preconditions.

We also investigate if R_i to D_c and R_c to D_i have the same rules. We find that the intersection is an empty set, indicating

[6]https://github.com/apache/hbase

[7]https://cwe.mitre.org/data/definitions/397.html

[8]https://wiki.sei.cmu.edu/confluence/display/java/ERR02-J.+Prevent+exceptions+while+logging+data

no violation that the reviewers (developers) are concerned about but the developers (reviewers) ignore.

2) Difference: For D_c and R_c difference sets, which are the different rules that developers and reviewers concern with, we found developers tend to focus on rules applied on a smaller-scale of code blocks, require less context, i.e., *Inappropriate "Collection" calls should not be made* and *Boxing and unboxing should not be immediately reversed*. Reviewers tend to be concerned about rules on the class level, which require an overall understanding, i.e., *Class names should not shadow interfaces or superclasses*. Reviewers also focus on removing deprecated code, which is not a concern for developers since its presence does not block developers' development tasks, indicating that reviewers place a greater emphasis on the maintainability of the code.

For Di and Ri difference sets, most of the rules developers ignore are related to naming conventions, variable modifiers, and unused code that won't affect their development tasks. Most types ignored by reviewers usually occur in a long block of code that might have loops, try/catch/finally blocks, or multiple if/else conditions, i.e., *Methods returns should not be invariant*. These violations are difficult to discover by reading through the code, as they require reviewers to keep too much information in their heads at one time. As mentioned above, both developers and reviewers care about the removal of unused code; an interesting finding is that *Unused "private" fields should be removed*, *Unnecessary imports should be removed* and *Unused "private" methods should be removed* appear in the top ten reviewers focused violation among projects but not in developers. Furthermore, two appear in the top ten that developers tend to ignore. All three rules are related to useless code at the class level and won't block the developer's development, so the developers easily overlook them. However, reviewers usually need to go through the entire context during code review, making it easier for them to identify such violations.

In addition, we identified the rules that reviewers concerns, but, developers ignore by examining the violations that are associated with CRCs but still exist in the revised version. In total, most of the violations ,i.e., 1,613 64.5%, mentioned by the reviewers were fixed by the developers, with 927 violations involving 119 rules. By inspecting survival but CRCs-related violations, we've come up with four reasons. (1) The developers and reviewers have agreed on the violations' existence, i.e., the developers explained code implementation considerations to the reviewers, and the developers promised to fix the violations in a subsequent version. (2) The reviewers did not check the revised code repeatedly. For example, the reviewers mentioned violations, and the developers revised the relevant code, but the violations were not corrected in the revised version, and the reviewers accepted. (3) Neither reviewers nor SAT conducted cross-file analysis. Specifically, the violations do not exist; the reviewer only reviewed the modified code and did not look at the code in the relevant file, thus marking the erroneous comments, and the SAT's lack of cross-file analysis capability caused the violation to be reported. (4) SAT misidentification. The violation was fixed, but SAT still reported it.

> Our detailed study of commonalities and differences between developers and reviewers in violation concerns and non-concerns provides data and insights to understand each other better and improve SATs promotion.

V. Threats to Validity

External validity. The main external threats to validity involve the generalizability of the dataset collection and the use of the SAT adopted. The selected projects inevitably threaten the results of this study since not all projects use SonarQube or other SATs, and analyzing the selected projects to study violations that developers and reviewers care about or neglect may limit the ability to generalize the conclusions to projects in different programming languages, ecosystems, or domains. However, the 52 projects are large, active, diverse, and have ample collaboration history, which to some extent mitigates the threat of generalization. On the other hand, adopting SonarQube as the SAT and the configured rules may threaten the validity of our study. As one of the most popular static analysis detection tools currently, SonarQube has more rules and better language detection capabilities than other tools used in previous studies, such as FindBugs and CheckStyle. Additionally, since each rule is specific to a particular problem, this may pose a slight threat to our conclusions. However, SonarQube and other SATs have considerable overlap in violation types. Although the names of the rules may differ, the functionality is the same.

Internal validity. The main internal threats to validity are the accuracy of the underlying algorithms used in this study,i.e., violation matching and fixed violation identification, and the accuracy of the data labeling. False-positive fixes may be introduced during the violation matching and fixed violation identification, which may bias our understanding of the violations developers care about. To address this issue, we performed additional reviews to confirm whether the collected fixed violations were truly fixed. The matching and fixed identification of violations aligned with the expected results in previous papers [10], [11]. Additionally, the data labeling process is inevitably subject to the subjective experience of the labelers. To mitigate this threat, we developed a page to display violations and CRCs information, making it easier for data labelers to view code and review comments. We also adopted cross-validation, and the Cohen kappa coefficient reached 0.83, indicating strong consistency. Furthermore, the benchmark database constructed and the data on the violation-to-CRC association pairs verified in RQ1 are not very large, despite being 2.3 times the manual verification in similar work. This is because such manual verification is very time-consuming and involves understanding the code's intent, the meaning of violations, and data statistical processes.

VI. Implications

In this section, we will discuss the results obtained according to the RQs and present possible practical implications of our research.

Developers. This study identifies the rules developers and reviewers are concerned about and non-concerned about, as well as the commonalities and differences. Since reviewers focus on some design violations at a high level, developers should develop a holistic awareness to avoid design problems. Considering that reviewers are concerned about the readability and understandability of the code and may misunderstand the meaning of the code, developers should increase their focus on code readability and maintainability to reduce reviewers' workload and misunderstanding of the code. Moreover, effective and adequate comments can reduce communication costs. Developers can use the SAT plugin, i.e., SonarLint, within the IDE or directly use SAT to review code before submission, improving code quality and reducing reviewer effort. In addition, we found that the rules that developers and reviewers pay attention to have a certain overlap between projects. Therefore, regarding SAT rule configuration and repair priority, developers can configure the corresponding rules and determine the order of violation repair by referring to the violations that developers and reviewers pay attention to in similar projects.

Reviewers. Our detailed analysis is useful for reviewers to understand developers' quality concerns. In addition, our findings provide some suggestions for code review. We found that the rules in the original code were more likely to occur in the newly committed code, so the reviewers could improve the overall quality of the project by strengthening the code inspection. Reviewers can require developers to configure uniform rules to reduce their effort. During the code review process, reviewers can use violation tracking tools and SAT to double-check that the violations they care about are fixed before merging the code. Some of the violations ignored by reviewers are difficult to perceive but can be fixed. Reviewers can use related rules to help them improve their ability to review.

Tool Builders. Our findings in studying why reviewers are concerned but violations remain and which rules developers and reviewers ignore reveal areas where tool builders can benefit users most. We found that the default rule configuration is unsuitable for most projects, and only a few rules were of concern to users. Although users in different projects care about the same rules, the overlap is not large, so domain-specific rule recommendations are necessary. In addition, the rules that users care about have clear commonalities, i.e., maintenance, security, and unused, so the default rule category may not be enough for users to screen out the rules they want quickly. It is necessary to enrich the rule labels. Many of the violations that users don't care about are false positives or domain-specific implementations, so tool builders can use these unfixed cases to improve and refine SAT capabilities. In addition, developers are more focused on newly introduced violations during code reviews. The SAT can reduce the number of violations to be reviewed by users by integrating violation-tracking capabilities.

VII. Related Work

A. Mapping CRCs and violations

At present, similar work mainly focuses on two aspects: using SATs to help CR and using CRCs to evaluate or improve the capacity of SATs. The idea of combining the SATs with code reviews to explore the benefits of each other is not new. The current work [1], [7]–[9], [41]–[43] primarily focuses on two areas: (1) using the SATs to aid code review and (2) using code review data to evaluate or improve SATs capabilities. Currently, few works directly match CRCs with specific violation instances. To the best of our knowledge, there are two studies [6], [9] most relevant to Com2Vio. Zampetti et al. [9] employed an automated method (Auto-SCAT) to recommend which SATs rule to enable using (statement-level) CRCs. Specifically, Auto-SCAT matches new CRCs with text in a knowledge base to enable relevant CheckStyle rules instead of directly correlating with the violation instances. Singh et al. [6] matched 274 CRCs with violations identified by PMD [44] to evaluate how static analysis tools can reduce code review effort. Due to their analysis being conducted on a minimal dataset, they relied entirely on manual judgment to determine whether a CRC and a violation referred to the same issue. In contrast, our benchmark dataset is manually labeled 16 times more CRCs than they did. Moreover, we incorporated this mapping process into our automated tool, Com2Vio, which demonstrated high accuracy in matching CRCs with violations.

B. Empirical Study on the Combination of SATs and CR

There are many empirical studies on code reviews or violations [1]–[3], [8], [22], [45]–[48]. The difference between our study and the empirical study focused on code review is that our investigation of between review and static analysis rules light on the commonalities and differences in the perspectives of reviewers and developers on the quality of the project. Furthermore, the difference between our study and focusing on the evolutionary history of violations is that we examine the changes in software quality during the code review process. Given the presence of manual review processes, developers and reviewers are more likely to pay greater attention to the code; thus, the conclusions drawn from our study may be more representative than studies that lack such manual review processes.

To the best of our knowledge, a few empirical studies have correlated CRCs with violations, mainly for the following three researches [1], [6], [7]. Panichella et al. [7] analyzed the closed violation in the code review processes of six Java open-source projects using CheckStyle and PMD. They aimed to investigate whether static analysis violations were addressed during code review and whether certain violations were more likely to be eliminated than others. Building on this work, Han et al. [1] furthered the investigation by considering the effects of rebasing and distinguishing between violations

resolved through reviewer comments versus those accidentally removed.

Singh et al. [6] conducted an empirical study involving 92 pull requests from 23 open-source Java projects to understand how static analysis tools can help reduce the manual effort associated with code reviews. They found that PMD can reduce the workload of code reviewers and distilled a set of four possible future static rules that cover nearly 17% of the comments. Unlike these studies, our research not only considers violation resolution from the developers' perspective but also investigates the quality concerns of reviewers. Additionally, we differentiate between fixed violations in the original code versus those introduced during the review process, providing a more comprehensive perspective on code quality.

VIII. CONCLUSION

This paper presents a large-scale empirical study to characterize and understand the concerns and non-concerns of developers and reviewers regarding violations. Our findings provided practical for developers, reviewers, and tool builders. In the future, we plan to investigate how to screen suitable rules for specific projects and how to use SATs to help better code review. Our dataset with the more complete analysis results is released at an anonymous website https://sites.google.com/view/com2vio.

REFERENCES

[1] D. Han, C. Ragkhitwetsagul, J. Krinke, M. Paixão, and G. Rosa, "Does code review really remove coding convention violations?" in *SCAM*. IEEE, 2020, pp. 43–53.

[2] N. Davila and I. Nunes, "A systematic literature review and taxonomy of modern code review," *J. Syst. Softw.*, vol. 177, p. 110951, 2021.

[3] J. Czerwonka, M. Greiler, and J. Tilford, "Code reviews do not find bugs. how the current code review best practice slows us down," in *ICSE (2)*. IEEE Computer Society, 2015, pp. 27–28.

[4] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, and H. C. Gall, "Context is king: The developer perspective on the usage of static analysis tools," in *SANER*. IEEE Computer Society, 2018, pp. 38–49.

[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 *SANER*. IEEE Computer Society, 2016, pp. 470–481.

[6] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, "Evaluating how static analysis tools can reduce code review effort," in *VL/HCC*. IEEE Computer Society, 2017, pp. 101–105.

[7] S. Panichella, V. Arnaoudova, M. D. Penta, and G. Antoniol, "Would static analysis tools help developers with code reviews?" in *SANER*. IEEE Computer Society, 2015, pp. 161–170.

[8] V. Balachandran, "Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation," in *ICSE*. IEEE Computer Society, 2013, pp. 931–940.

[9] 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.

[10] P. Yu, Y. Wu, J. Peng, J. Zhang, and P. Xie, "Towards understanding fixes of sonarqube static analysis violations: A large-scale empirical study," in *SANER*, 2023, to appear. Preprint available at: https://github.com/FudanSELab/sq-violation-dataset/blob/master/Understanding

[11] P. Yu, Y. Wu, X. Peng, J. Peng, J. Zhang, P. Xie, and W. Zhao, "Violationtracker: Building precise histories for static analysis violations," in *ICSE*, 2023, to appear. Preprint available at: https://conf.researchr.org/details/icse-2023/icse-2023-technical-track/12/ViolationTracker-Building-Precise-Histories-for-Static-Analysis-Violations.

[12] 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 *ICSE*. IEEE Computer Society, 2013, pp. 672–681.

[13] 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.

[14] A. Habib and M. Pradel, "How many of all bugs do we find? a study of static bug detectors," in *ASE*. ACM, 2018, pp. 317–328.

[15] V. Lenarduzzi, F. Lomio, H. Huttunen, and D. Taibi, "Are sonarqube rules inducing bugs?" in *SANER*. IEEE, 2020, pp. 501–511.

[16] 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.

[17] H. J. Kang, K. L. Aw, and D. Lo, "Detecting false alarms from automatic static analysis tools: How far are we?" in *ICSE*. ACM, 2022, pp. 698–709.

[18] S. Mehrpour and T. D. LaToza, "Can static analysis tools find more defects?" *Empir. Softw. Eng.*, vol. 28, no. 1, p. 5, 2023.

[19] M. Paixão, J. Krinke, D. Han, and M. Harman, "CROP: linking code reviews to source code changes," in *MSR*. ACM, 2018, pp. 46–49.

[20] G. Gousios, A. Zaidman, M. D. Storey, and A. van Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *ICSE (1)*. IEEE Computer Society, 2015, pp. 358–368.

[21] G. Gousios, M. D. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: the contributor's perspective," in *ICSE*. ACM, 2016, pp. 285–296.

[22] W. Zou, J. Xuan, X. Xie, Z. Chen, and B. Xu, "How does code style inconsistency affect pull request integration? an exploratory study on 117 github projects," *Empir. Softw. Eng.*, vol. 24, no. 6, pp. 3871–3903, 2019.

[23] 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 *SANER*. IEEE Computer Society, 2018, pp. 153–163.

[24] 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 *ICPC*. IEEE / ACM, 2019, pp. 209–219.

[25] 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.

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

[27] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds. Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: https://doi.org/10.18653/v1/n19-1423

[28] L. Floridi and M. Chiriatti, "GPT-3: its nature, scope, limits, and consequences," *Minds Mach.*, vol. 30, no. 4, pp. 681–694, 2020. [Online]. Available: https://doi.org/10.1007/s11023-020-09548-1

[29] P. Avgustinov, A. I. Baars, A. S. Henriksen, R. G. Lavender, G. Menzel, O. de Moor, M. Schäfer, and J. Tibble, "Tracking static analysis violations over time to capture developer characteristics," in *ICSE (1)*. IEEE Computer Society, 2015, pp. 437–447.

[30] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. L. Traon, "Mining fix patterns for findbugs violations," *IEEE Trans. Software Eng.*, vol. 47, no. 1, pp. 165–188, 2021.

[31] J. Spacco, D. Hovemeyer, and W. W. Pugh, "Tracking defect warnings across versions," in *MSR*. ACM, 2006, pp. 133–136.

[32] S. Kim and M. D. Ernst, "Which warnings should I fix first?" in *ESEC/SIGSOFT FSE*. ACM, 2007, pp. 45–54.

[33] S. S. Heckman and L. A. Williams, "On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques," in *ESEM*. ACM, 2008, pp. 41–50.

[34] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: improving actionable alert ranking," in *MSR*. ACM, 2014, pp. 152–161.

[35] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in *MSR*. IEEE Computer Society, 2007, p. 27.
[36] S. S. Heckman and L. A. Williams, "A model building process for identifying actionable static analysis alerts," in *ICST*. IEEE Computer Society, 2009, pp. 161–170.
[37] J. Wang, S. Wang, and Q. Wang, "Is there a "golden" feature set for static warning identification?: an experimental evaluation," in *ESEM*. ACM, 2018, pp. 17:1–17:10.
[38] J. Yoon, M. Jin, and Y. Jung, "Reducing false alarms from an industrial-strength static analyzer by SVM," in *APSEC (2)*. IEEE, 2014, pp. 3–6.
[39] R. D. Venkatasubramanyam and S. Gupta, "An automated approach to detect violations with high confidence in incremental code using a learning system," in *ICSE Companion*. ACM, 2014, pp. 472–475.
[40] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing, EMNLP-IJCNLP 2019, Hong Kong, China, November 3-7, 2019*, K. Inui, J. Jiang, V. Ng, and X. Wan, Eds. Association for Computational Linguistics, 2019, pp. 3980–3990. [Online]. Available: https://doi.org/10.18653/v1/D19-1410
[41] A. Carvalho, W. P. Luz, D. Marcilio, R. Bonifácio, G. Pinto, and E. D. Canedo, "C-3PR: A bot for fixing static analysis violations via pull requests," in *SANER*. IEEE, 2020, pp. 161–171.
[42] S. Gunawardena, E. D. Tempero, and K. Blincoe, "Concerns identified in code review: A fine-grained, faceted classification," *Inf. Softw. Technol.*, vol. 153, p. 107054, 2023.
[43] S. Mehrpour and T. D. LaToza, "Can static analysis tools find more defects?" *Empir. Softw. Eng.*, vol. 28, no. 1, p. 5, 2023.
[44] P. 6.45.0, "Pmd source code analyzer," [Online], 2022, available: https://pmd.github.io, Last Accessed on: April. 2022.
[45] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *ICSE*. IEEE Computer Society, 2013, pp. 712–721.
[46] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," in *ICSME*. IEEE Computer Society, 2015, pp. 81–90.
[47] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: how developers see it," in *ICSE*. ACM, 2016, pp. 1028–1038.
[48] V. Lenarduzzi, V. Nikkola, N. Saarimäki, and D. Taibi, "Does code quality affect pull request acceptance? an empirical study," *J. Syst. Softw.*, vol. 171, p. 110806, 2021.