VioBench: Establishing a Benchmark of Static Analysis Violations for Promoting Postprocessing Techniques

Anonymous Author(s)

ABSTRACT

Although source code static analysis tools are widely used in industry to detect quality violations to the rules automatically, the detection results are typically underused due to the large amount of *unactionable* violations that developers would not act on. Postprocessing techniques have been proposed to help developers focus on the violations that are most likely to be actionable. However, deciding whether a violation is actionable is subjective and could be affected by the context of the violation. Therefore, most postprocessing techniques are difficult to be evaluated due to the lack of a curated benchmark of violations with reliable actionability labels and with sufficient evidence that supports the label. In this work, we propose a semi-automatic framework, VioBench, for establishing a benchmark of static analysis violations. VioBench automatically collects the violations with the full history and relates supportive data, such as commit messages gathered from the repository to the violations. It then provides a tool-supported manual verification procedure for violation annotators to annotate the violation. With VioBench, we constructed a benchmark containing 1,650 human-annotated violations with accurate and explainable labels of actionability. We report the data composition of this benchmark and demonstrate its usefulness in a preliminary case study evaluating the effectiveness of the state-of-the-art violation postprocessing techniques. The results also reveal possibilities of promoting new techniques to filter or prioritize the reported violations.

ACM Reference Format:
Anonymous Author(s). 2023. VioBench: Establishing a Benchmark of Static Analysis Violations for Promoting Postprocessing Techniques. In *Proceedings of 46th International Conference on Software Engineering (ICSE 2024).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Static analysis tools (SATs) are important in detecting quality threats in source code. However, studies have indicated that the violation detection results produced by the SATs are not well utilized partly because developers may be bothered by numerous false positives and unactionable violations that the developer won't or can't fix [3, 6, 12, 15, 21, 28–30, 32, 57].

To address this problem, many postprocessing techniques [42] are proposed for developers to focus on the violations that are more likely to be *actionable*. For example, Wang et al. [58] discovered 23 "Golden Features" representing contextual and historical characteristics of violations to determine whether a violation is actionable. Based on the "Golden Features", a number of machine learning techniques [60–62] have been proposed to automatically identify actionable violations and were reported to be highly effective.

However, recent research [23] observed that the strong performance of these approaches was largely caused by (1) data leakage when analyzing the violation context and defect likelihood (the "leakage features") of the violations and (2) unconfirmed (un)actionable violation oracle. The decision whether a violation is actionable or not is traditionally based on the *closed warning heuristic* (CWH) [58, 60]. While it is fully automatic, CWH has been criticized by recent studies [23, 66] for its inaccuracy. Some violations that have been accidentally closed by developers could be mistakenly decided as actionable, whereas some others that are still open for the time being could be wrongly decided as unactionable [23].

Moreover, the decision of actionability is subjective and may depend on many supportive data, including the source code context, the commit messages, and the related issue/bug reports. These data may provide evidence of whether the violation was closed by accident (closed but actually unactionable) or is an undisclosed bug (open but should be actionable). Therefore, a reliable benchmark of static analysis violations annotated with actionability requires human verification and should be accompanied by multi-sources of supportive data collected from the code repositories.

Although there have been a number of violation benchmark datasets built from a variety of open source projects of various sizes and languages [18, 36, 43, 50], most of them suffer from factual errors (e.g., inaccurate introduction and closure of violations without human validation) [2, 34, 66] and lack a detailed, repeatable data construction process [18]. The critical components of benchmark construction, such as well-defined data sources, required steps of data processing, clear guidance for data annotation, and human-friendly tool support, are unfortunately missing.

In this work, we propose a semi-automatic framework, VioBench, which enables collecting violations and supportive data (i.e., commit messages, code review comments, and issues) from code repositories. We first investigate the requirements for a benchmark of violations that are valid for facilitating and evaluating violation postprocessing techniques. Next, we present the details of the automatic and manual phases of VioBench, which contain (1) the automatic collection of violations and supportive data, (2) the criteria for selecting violations to be annotated, and (3) the process for manual annotating of violations. To simplify the annotating process, we developed ViolationNoter, an IntelliJ IDEA plugin, to facilitate violation visualization and annotation. With VioBench, we have constructed a curated benchmark from eight high-quality open-source projects containing 1,650 human-validated violations. Each violation is annotated for its actionability and associated with

ICSE 2024, April 2024, Lisbon, Portugal
© 2018 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

supportive data such as commit messages and bug/issue information (if any) that could be the supportive explanation of the actionability, priority, and repair decision. Finally, we conduct a preliminary case study to replicate the violation filtration and prioritization experiments with three state-of-the-art postprocessing techniques. For violation filtering, we find that using more accurate data may improve the effectiveness of the SVM-based techniques supported by the golden features, but chances still exist for further improvement. For violation prioritizing, we find that the prioritization results produced by SonarQube (with the default priority setting), a multiple-tool-based approach, and a history-aware-based approach exhibit disagreement, indicating chances to optimize the prioritization by further considering the context in which the violation is located and the potential quality problems it may cause.

We summarize the main contributions of this work as follows:

- A semi-automatic violation benchmark construction framework, VioBench, that formulates the collection and annotation process for the violations and the related data, with simple-yet-effective tool support for manual data inspection and annotation[1].
- A benchmark dataset constructed with the VioBench framework, containing 1,650 manually-validated violations collected from eight open-source projects, with the label of actionability and cross-validated supportive data that support the actionability decision.[2] The actionability of each violation is accompanied with detailed documentation describing the cause of the violation, detailed location information, introducing and fixing commits (if available), and supportive data such as commit messages, issues, and code review discussions.
- A preliminary case study with three state-of-the-art violation postprocessing techniques on violation filtration and prioritization, demonstrating the usefulness of the benchmark dataset and revealing the pitfalls and possible improvement directions of existing postprocessing techniques.

2 BACKGROUND

2.1 Violation Postprocessing Techniques

Postprocessing the reported violations to help developers to focus on actionable ones and to assist developers in their fixing actions are known as violation postprocessing techniques. Muske et al. categorized postprocessing techniques and presented six areas of postprocessing techniques, namely clustering, ranking, pruning, automated false positives elimination (AFPE), validation with code analyses, and simplification by manual inspection. While this classification provides an overview of all techniques for violation postprocessing, we observe the inherent needs of developers when they are confronted with a huge amount of violations reported by SATs from a user-centered view proposed by Nguyen et al. [11]. First, the long list of violations should be reduced and properly ordered. The developers expect to see actionable violations rather than unactionable ones or false positives. When the list is long, more important ones (e.g., bug-prone ones) are expected to be on the top part of the list. Second, when facing a violation to be fixed, a developer may need assistance to understand the rationale behind the violation and for making an appropriate fixing decision. Based on this observation, we categorize the postprocessing techniques into the following two categories: (1) violation filtering and prioritization and (2) violation fixing assistance. While the latter is important for developer to act on the violations, the former is somehow more important to present the actionable violations to the developers.

An accurate dataset of (un)actionable violations that can be verified by humans through related supportive data is critical and fundamental for the development of postprocessing techniques. However, there is currently no objective and accurate method for the decision of actionability of violations. Therefore, a human-validated benchmarking dataset of violations that can be repeatedly and publicly viewed and curated is the most fundamental infrastructure for promoting postprocessing techniques and for serving the developers in the first place.

2.2 Existing Violation Benchmarks and Datasets

Nunes et al. [44] proposed a benchmark for assessing and comparing static analysis tools in terms of their capability to detect web-related security vulnerabilities. The benchmark includes 134 off-the-shelf WordPress plugins organized into four vulnerability detection scenarios. However, it covers only web-related vulnerabilities, which is not applicable for other types of violations.

Yu et al. [66] constructed a benchmark dataset containing 500 manually validated static violations and 30 violation cases with thoroughly analyzed evolution history details for violation matching and tracing evaluation. However, it targeted violation tracking and did not provide details on whether the violations were actionable or unactionable.

Heckman et al. [18] proposed FAULTBENCH, containing 357 violations reported by FindBugs from six open-source projects, among which 67 were identified as true positives (TPs) and 290 false positives (FPs). Although FAULTBENCH can be used to evaluate false positive analysis techniques in violation filtering, it cannot be used to evaluate the identification of actionable violations because the violations are not rigorously reviewed for actionability. FAULTBENCH also included the priorities of the TPs as an evaluation benchmark for violation prioritization techniques. However, the prioritization information was merely based on the heuristics based on the number of closed violations, which have been challenged for accuracy by recent research [23, 65]. In addition, the introduction and fixing of the violations in FAULTBENCH are considered inaccurate due to its violation matching method [2, 34, 66], directly affecting the detection of violation closure.

There are also other benchmark datasets containing bugs and/or violations [4, 22, 36]. However, they are not specific to violations detected in code static analysis and thus cannot support violation postprocessing techniques.

Based on these observations, the existing benchmark datasets in the literature have pitfalls and constrain the development of new violation postprocessing techniques. A desirable violation benchmark should be scalable, easily verifiable, and contain violations

[1]To get more developers/users involved, we have also developed a web-based tool for collaborative annotation. If this work is accepted, the web-based tool will be open-access to the public.
[2]The raw data, benchmark, and source code of plugin can be found online anonymously at https://sites.google.com/view/viobench/ during the double-blind review. To be publicly available for replication if this work is accepted.

with human-validated actionability and priorities with detailed supportive data that can be replicated and reviewed by the public.

3 THE VIOBENCH FRAMEWORK

In this section, we first sketch an overview of the VioBench framework for establishing a benchmark of static analysis violations. Then, we elaborate on each component of VioBench.

3.1 Framework Overview

The VioBench framework (Figure 1) consists of three main components: an automatic process for collecting violations and supportive data, a semi-automatic process for selecting violations to be included in the benchmark, and a manual process for annotating the violations.

In the first process, VioBench automatically collects all violations and their evolution histories from software repositories with off-the-shelf SATs and a state-of-the-art violation tracking tool. For each violation and related commit histories, VioBench also searches in the commit messages, code review comments, and related issue[3] descriptions of the supportive data that can be used for actionability decisions. Since the number of violations automatically collected could be huge, VioBench provides a mechanism for users to select violations that are appropriate for inclusion in the benchmark. This process is semi-automatic with automatic tool support for selecting commits and files where the violations survive and criteria for tuning the tools to include a reasonable number of violations to be manually annotated. Finally, VioBench employs a manual annotation process supported by an IDE plugin tool named ViolationNoter for comprehensive data annotation. All annotated violations are stored in the benchmark database and can be reviewed and further discussed[4].

We elaborate on these components as follows.

3.2 Automatically Collecting Violations and Supportive Data

3.2.1 Collecting Violations. We use SonarQube [52] to detect violations for each commit of the target software and ViolationTracker [66] to build complete evolutionary histories for the detected violations. SonarQube is one of the most common open-source SATs adopted both in academia and industry [32, 65]. It supports multiple languages with various sets of static analysis rules. In this work, we only focus on Java language with the default Java rule set. Other languages and rules can be extended with little effort. ViolationTracker is the state-of-the-art tool that builds precise evolution histories for violations. It is designed to be SAT-independant and able to work with any SATs. To be accompanied with the scope of this work, we use the detection results of SonarQube for history construction.

3.2.2 Collecting Supportive Data for the Violations. VioBench is able to collect supportive data, such as violation-related commit messages, code review comments, and issue reports[5]. With the help of ViolationTracker, the commits that touch the files and functions where the violations reside are recorded for each violation. Therefore, the commit messages and code review comments related to the violations are collected by Git APIs. Also, if a commit message mentions certain issue ids, the related issue descriptions are collected by APIs provided by corresponding issue management platforms (e.g., GitHub, Jira, and Bugzilla).

Finally the violations (along with the histories) and the corresponding supportive data are related and ready to be determined whether to be included in the benchmark.

3.3 Selecting Violations to be Annotated

The violations collected in the previous process could be numerous. It is infeasible for users to review and annotate all of them. Moreover, it has been reported that there could be a huge part of violations that are unactionable. Therefore, randomly selecting the violations might hit many unactionable cases but few actionable ones, which hurts the quality of the benchmark.

For the purpose of deciding the actionability of violations, we adopt a two-step selection process. First, we select violations that were introduced between two *releases* of the target software. Releases are the baseline commits that produce deployable and comparatively stable software. Since releases are for a broader audience to download and use [6], it is more likely that the violations should have been fixed before a release if the developers care about them. Although it is not always the fact, we apply this heuristic to narrow down the time frame from which we select violations that are introduced.

Second, we select violations that are reported from files that are structurally centric and bug-prone. After our first step, there could be still too many violations remaining. The violations reported from structurally-centric files typically have a higher possibility of affecting the quality of the software and, thus, are more representative and could be valuable for reviewing. Meanwhile, recent researches [11, 65] reported that developers usually focus more on bug-related violations. Hence bug-prone violations are also preferred in this step.

We apply the following heuristics and supporting tools to select potentially valuable violations from the raw set of violations.

3.3.1 Selecting Violations Introduced between Two Releases. The evolution history of a violation provides information on how developers deal with it. A violation with a very short history, however, lacks this information. Therefore, we opt to choose a release that is sufficiently distant in time from the present. Typically, a two-year period from the present is often used in previous studies [9, 23, 65] as literature shows that most violations were fixed within two years otherwise never fixed. This release is the end point of the time frame for violation selection.

Meanwhile, a violation with a very long history (e.g., 10 years) could be too "old" to be noticed by developers and thus have less

[3]In open-source projects, "issues" are typically tracked by Github Issue feature or other bug management systems such as Bugzilla, and can be related to the commit history by mentioning the issue ids in the commit message.
[4]The online editing and crowdsourced annotation are out of the scope of this paper but already under development.

[5]Issues can be tracked by many systems. For example, GitHub provides issue tracking feature as shown at https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues.
[6]https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases

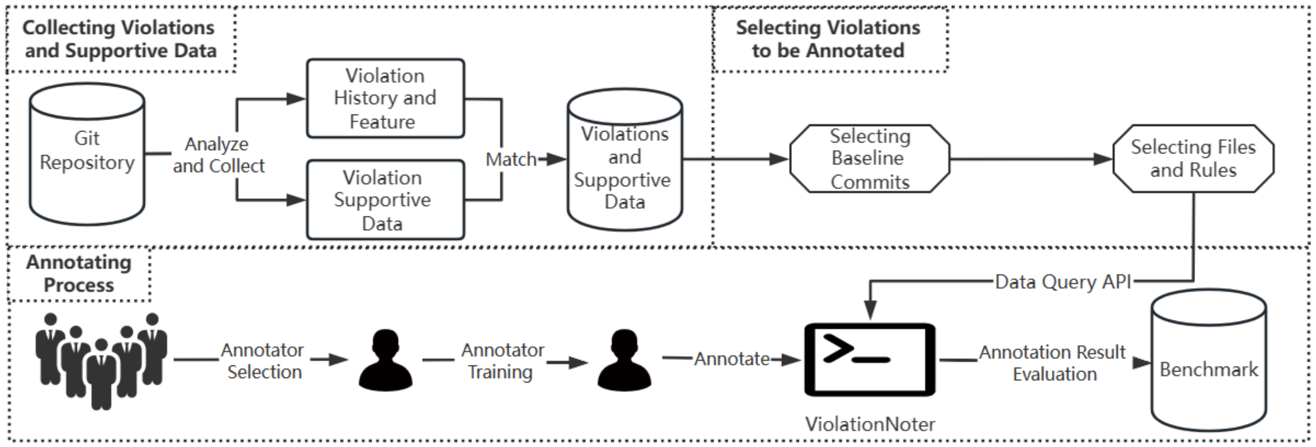

Collecting Violations and Supportive Data
Git Repository
Analyze and Collect
Violation History and Feature
Violation Supportive Data
Match
Violations and Supportive Data
Selecting Violations to be Annotated
Selecting Baseline Commits
Selecting Files and Rules
Data Query API
Annotating Process
Annotator Selection
Annotator Training
Annotate
ViolationNoter
Annotation Result Evaluation
Benchmark

Figure 1: An Overview of VioBench

value to be analyzed. Therefore, we choose a release before the two-year old release but not yet too old. Only the violations that were introduced between the two releases are selected for further analysis.

3.3.2 Selecting Violations Survived in Important *Files and Detected by* Bug-Related *Rules.* To find the *important* files, we adopted the *top-core method* proposed by Qu et al. [46] to select the most bug-prone files as the target of selected files. The *top-core method* is based on complex network theory and is able to efficiently predict potential bugs in files. The violations in such files are preferential for further analysis.

We also tend to select violations that may cause bugs or be related to errors. In practice, the violations against bug-related rules are relatively fewer than smell-related and others. Therefore, VioBench provides the violations collected as per rule, and we may supplement bug-related violations to the violation set to be further analyzed. If the proportion of rules selected in the first step is imbalanced, we will select additional files and rules from the whole history of repositories. After this process, the candidate violations for inclusion in the benchmark are selected and ready for manual annotation.

3.4 Annotating the Violations

3.4.1 Selecting and Training Annotators. The first thing is to ensure that all annotators are qualified with enough technical background and should be trained to understand the basic concepts, such as the actionability of violations and the purpose of the annotation task.

Annotator Selection. The annotators should meet the following criteria to be qualified for the annotation task: (1) The annotators should have participated in at least two software development projects so that they are capable of comprehending the changes in the source code; (2) The annotators should be familiar with the Git mechanism and understand the concepts of Issues and Pull Requests in the context of GitHub; and (3) The annotators should have used SonarQube or SonarLint [7] in their daily development work so that he understands most of the static analysis violations or knows how to find the meanings of the reported violations.

Annotator Training. We designed a two-hour training course for the annotators to get familiar with the purpose and the annotating environment. The criteria for judging whether a violation is purposefully fixed or is closed by accident are also taught in the training course. New annotators should pass annotation tests on random ground-truth violations. In the tests, the annotator should align her/his understandings and criteria for judging the actionability of the violation to the ground-truth[8].

3.4.2 Criteria for Annotating Violations. All violations are judged for the following properties: (1) Authenticity (TP/FP); (2) Actionability (Actionable/Unactionable); (3) Priority; (4) Way and Code of Fixing.

Authenticity. Annotators should decide whether a violation is True-Positive (TP) or False-Positive (FP). Some existing work [23, 42, 58] defined FPs to include both technically-erroneous results and those that developers would not or cannot fix, which may cause conceptual confusion. We define FPs with strict criteria to only include the technically-erroneous results. It conforms with the common sense that an FP is unactionable, but an unactionable violation may not be an FP. An FP violation does not cause the reported problem in the program context of the case for complex reasons. A typical reason for technical errors is the limitation in program analysis techniques that fail to trace control or data flows in the scope of analysis. Therefore, if a violation is decided as FP, the annotator should also give a reason why the violation does not cause the reported problem.

Actionability. We define *actionable/unactionable* violations within the scope of True-Positives (TPs). Most studies treat violations that disappeared from history due to non-file deletion as actionable [23, 34, 58, 60]. However, there are cases where a disappeared violation which was accidentally closed by other source code changes is unactionable and a violation which is not discovered by developers

[7]https://www.sonarsource.com/products/sonarlint/

[8]This is a cold-start problem, i.e., who creates the first ground-truth. In this work, the ground-truth violations are manually annotated and collaboratively discussed by the team of authors and reviewed by experts from academia and industry. It is actually the core of the benchmark that we will present in Section 4

is actionable. We argue that the actionability of a violation should be determined by the fact whether it was or could be deliberately fixed and whether it is explicitly ignored by developers. Supportive information includes the related commit messages, code review comments, and descriptions of the issues mentioned in the commit messages or review comments. Therefore, annotators should label a violation with the following criteria:

(1) Label a closed violation as actionable if there is concrete evidence that the violation is deliberately fixed.
(2) Label an open violation as unactionable if there is concrete evidence from code review comments, commit messages, or issue descriptions that the violation is proactively ignored.
(3) Label a closed violation as unactionable if it does not meet Criterion (1) and there is concrete evidence from code review comments, commit messages, or issue descriptions that the violation is proactively ignored.
(4) Label an open violation as actionable if it does not meet Criterion (2), and there is concrete evidence that the reported problem exists and there is an obvious way to fix it with reasonable effort. (5) Otherwise, the actionability is uncertain and makes no labels.

Note that some violations that developers do not care about or won't fix could be labeled as actionable by this criteria. We argue that the actionability and priority should be labeled separately. These violations typically do not cause quality issues of the program so the fixing priorities are low even if they are actionable. Therefore, we design further criteria of priority for the violations.

Priority.

In this work, only actionable violations are assigned a level of priority. The principle for violation prioritization is to arrange the violations more likely to cause quality issues to the upper part of the list. We design the six priority levels with the following criteria. A violation goes to the priority level at the first criterion match.
P6: A violation is associated with bugs or reported issues.
P5: A violation is explicitly mentioned as fixed in commit messages or code review comments.
P4: A violation is most likely to cause malfunctioning or quality issues.
P3: A violation is found to be mentioned in commit messages or code review comments, regardless the final decision is ignore it or not. This means that the developers at least considered the violation.
P2: A violation that is in a frequently modified file in recent time. The underlying rationale is that frequently modified code should be given more attention as it is more likely to cause errors [40, 47, 55, 56].
P1: All remaining actionable violations.

Way and Code of Fixing The way in which a violation is fixed is also important supportive data for understanding the actionability of violations. Thanks to the ViolationTracker tool, the commit in which the violation is closed is precisely recorded. The annotators are supposed to annotate two types of information. First, they should determine whether the fixing way is the same as what the SAT recommends. Second, they should identify the specific code lines related to the fix.

The annotators should filter out code modifications unrelated to the violation and focus on modifications to data flow and control flow related to the violation. Although this task requires advanced knowledge and understanding of the program, the annotations are beneficial for other research areas, such as automatic violation fixing or fixing pattern mining and SAT builders, to provide more useful fixing suggestions. The fixing code has been automatically collected, so the annotators should only review the recorded fixing code and make confirmation.

3.4.3 *ViolationNoter: The Tool Support.*

A previous study [11] has noted that developers prefer a graphical user interface for violation analysis, such as an Integrated Development Environment (IDE). In addition, displaying information about the violations in the same user interface would reduce developers' inspection efforts. Therefore, we develop a simple yet effective IntelliJ IDEA plugin, ViolationNoter, for annotating violations.

Each violation is displayed in the file/method where it resides. The violation is shown at the introduced commit. The annotator may refer to the evolution history of the violations shown by the plugin tool. The introduced commit, the closed commit (if exists), and all intermediate commits are accessible to the annotators. Meanwhile, the plugin tool also provides code review comments and related issues (if available) so that the annotators may understand the purpose of the code changes. The running ViolationNoter plugin is shown in Figure 2(a). The UI is divided into two parts. The upper part is the source code segments before and after the commit provided by the IntelliJ IDEA code editor. The lower part is the violation information panels provided by ViolationNoter, which is the first tab page among a total of three tab pages. The left part of the tag page displays all the violation information in the current file, including line number, type, and details. Annotators use this UI to annotate the violations according to the given properties (i.e., authenticity, actionability, etc.) and corresponding criteria. The right part shows historical information obtained through ViolationTracker, including the introduction version, elimination version, and survival time of violations. The second tab page (Figure 2(b)) shows code review information related to this file, and the third page (Figure 2(c)) displays the list of related issues or bugs for the annotator's references.

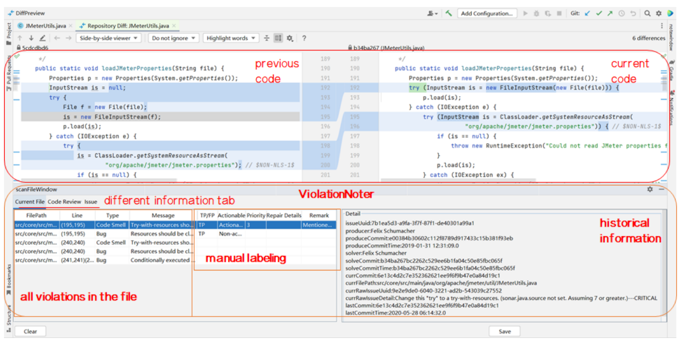
To improve the reliability of the annotation, each violation is independently reviewed and annotated by at least two annotators. Different annotations results will be separately recorded and later resolved by a third senior annotator or by majority-voting mechanisms if used in a crowdsourcing environment.

4 ESTABLISHING A VIOLATION BENCHMARK

In this section, we describe our attempt to build a violation benchmark with the VioBench framework. We begin with selecting representative projects and then describe the constructed benchmark. Finally we discuss the usefulness of the benchmark.

4.1 Selecting Representative Projects

Selecting representative projects from a large pool of open-source projects is not trivial. Previous studies typically selected high-quality, large-scale open-source projects with long maintenance histories [9, 23, 33, 37, 53, 65, 66, 68]. Such software projects usually represents the strict quality control of source code, especially before the release of a stable commit. We follow the same criteria in selecting representative projects for violation collection. We believe



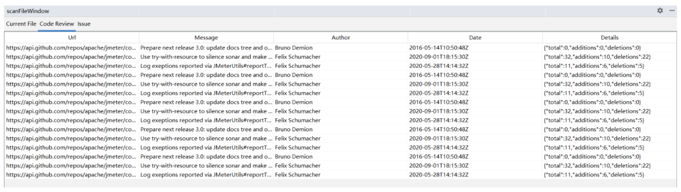
```
    */
    public static void loadJMeterProperties(String file) {
        Properties p = new Properties(System.getProperties());
        InputStream is = null;
        try {
            File f = new File(file);
            is = new FileInputStream(f);
            p.load(is);
        } catch (IOException e) {
            try {
                is = ClassLoader.getSystemResourceAsStream(
                        "org/apache/jmeter/jmeter.properties"); // $NON-NLS-1$
                if (is == null) {
```

```
    */
    public static void loadJMeterProperties(String file) {
        Properties p = new Properties(System.getProperties());
        try (InputStream is = new FileInputStream(new File(file))) {
            p.load(is);
        } catch (IOException e) {
            try (InputStream is = ClassLoader.getSystemResourceAsStream(
                    "org/apache/jmeter/jmeter.properties")) { // $NON-NLS-1$
                if (is == null) {
                    throw new RuntimeException("Could not read JMeter properties f
                }
                p.load(is);
            } catch (IOException ex) {
```

FilePath	Line	Type	Message	TP/FP	Actionable	Priority	Repair Details	Remark
src/core/src/m...	(195,195)	Code Smell	Try-with-resources sho...	TP	Actiona...	3		Mentione...
src/core/src/m...	(195,195)	Bug	Resources should be cl...	TP	Non-ac...			
src/core/src/m...	(240,240)	Code Smell	Try-with-resources sho...					
src/core/src/m...	(240,240)	Bug	Resources should be cl...					
src/core/src/m...	(241,241)(2...	Bug	Conditionally executed ...					

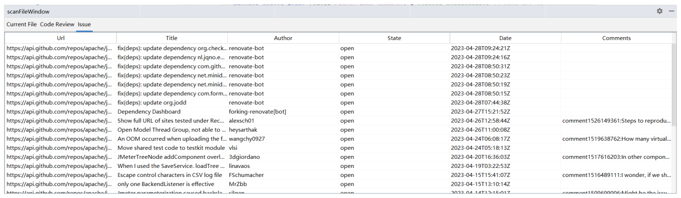
```
Detail
issueUuid:7b1ea5d3-a9fa-3f7f-87f1-de40301a99a1
producer:Felix Schumacher
produceCommit:e00384b30602c112f8789d917433c15b381f93eb
produceCommitTime:2019-01-31 12:31:09.0
solver:Felix Schumacher
solveCommit:b34ba267bc2262c529ee6b1fa04c50e85fbc065f
solveCommitTime:b34ba267bc2262c529ee6b1fa04c50e85fbc065f
currCommit:6e13c4d2c7e352362621ee9f6f9b47e0a84d19c1
currFilePath:src/core/src/main/java/org/apache/jmeter/util/JMeterUtils.java
currRawIssueUuid:9e2e9de0-6040-3221-ad2b-543039c27552
currRawIssueDetail:Change this "try" to a try-with-resources. (sonar.java.source not set. Assuming 7 or greater.)---CRITICAL
lastCommit:6e13c4d2c7e352362621ee9f6f9b47e0a84d19c1
lastCommitTime:2020-05-28 06:14:32.0
```

(a) Violations' Information



Url	Message	Author	Date	Details
https://api.github.com/repos/apache/jmeter/co...	Prepare next release 3.0: update docs tree and o...	Bruno Demion	2016-05-14T10:50:48Z	{"total":0,"additions":0,"deletions":0}
https://api.github.com/repos/apache/jmeter/co...	Use try-with-resource to silence sonar and make ...	Felix Schumacher	2020-09-01T18:15:30Z	{"total":32,"additions":10,"deletions":22}
https://api.github.com/repos/apache/jmeter/co...	Log exeptions reported via JMeterUtils#reportT...	Felix Schumacher	2020-05-28T14:14:32Z	{"total":11,"additions":6,"deletions":5}
https://api.github.com/repos/apache/jmeter/co...	Prepare next release 3.0: update docs tree and o...	Bruno Demion	2016-05-14T10:50:48Z	{"total":0,"additions":0,"deletions":0}
https://api.github.com/repos/apache/jmeter/co...	Use try-with-resource to silence sonar and make ...	Felix Schumacher	2020-09-01T18:15:30Z	{"total":32,"additions":10,"deletions":22}
https://api.github.com/repos/apache/jmeter/co...	Log exeptions reported via JMeterUtils#reportT...	Felix Schumacher	2020-05-28T14:14:32Z	{"total":11,"additions":6,"deletions":5}
https://api.github.com/repos/apache/jmeter/co...	Prepare next release 3.0: update docs tree and o...	Bruno Demion	2016-05-14T10:50:48Z	{"total":0,"additions":0,"deletions":0}
https://api.github.com/repos/apache/jmeter/co...	Use try-with-resource to silence sonar and make ...	Felix Schumacher	2020-09-01T18:15:30Z	{"total":32,"additions":10,"deletions":22}
https://api.github.com/repos/apache/jmeter/co...	Log exeptions reported via JMeterUtils#reportT...	Felix Schumacher	2020-05-28T14:14:32Z	{"total":11,"additions":6,"deletions":5}
https://api.github.com/repos/apache/jmeter/co...	Prepare next release 3.0: update docs tree and o...	Bruno Demion	2016-05-14T10:50:48Z	{"total":0,"additions":0,"deletions":0}
https://api.github.com/repos/apache/jmeter/co...	Use try-with-resource to silence sonar and make ...	Felix Schumacher	2020-09-01T18:15:30Z	{"total":32,"additions":10,"deletions":22}
https://api.github.com/repos/apache/jmeter/co...	Log exeptions reported via JMeterUtils#reportT...	Felix Schumacher	2020-05-28T14:14:32Z	{"total":11,"additions":6,"deletions":5}
https://api.github.com/repos/apache/jmeter/co...	Prepare next release 3.0: update docs tree and o...	Bruno Demion	2016-05-14T10:50:48Z	{"total":0,"additions":0,"deletions":0}
https://api.github.com/repos/apache/jmeter/co...	Use try-with-resource to silence sonar and make ...	Felix Schumacher	2020-09-01T18:15:30Z	{"total":32,"additions":10,"deletions":22}
https://api.github.com/repos/apache/jmeter/co...	Log exeptions reported via JMeterUtils#reportT...	Felix Schumacher	2020-05-28T14:14:32Z	{"total":11,"additions":6,"deletions":5}

(b) List of Code Reviews



Url	Title	Author	State	Date	Comments
https://api.github.com/repos/apache/j...	fix(deps): update dependency org.check...	renovate-bot	open	2023-04-28T09:24:21Z	
https://api.github.com/repos/apache/j...	fix(deps): update dependency nl.jqno.e...	renovate-bot	open	2023-04-28T09:24:16Z	
https://api.github.com/repos/apache/j...	fix(deps): update dependency com.gith...	renovate-bot	open	2023-04-28T08:50:31Z	
https://api.github.com/repos/apache/j...	fix(deps): update dependency net.minid...	renovate-bot	open	2023-04-28T08:50:23Z	
https://api.github.com/repos/apache/j...	fix(deps): update dependency net.minid...	renovate-bot	open	2023-04-28T08:50:19Z	
https://api.github.com/repos/apache/j...	fix(deps): update dependency com.form...	renovate-bot	open	2023-04-28T08:50:15Z	
https://api.github.com/repos/apache/j...	fix(deps): update org.jodd	renovate-bot	open	2023-04-28T07:44:38Z	
https://api.github.com/repos/apache/j...	Dependency Dashboard	forking-renovate[bot]	open	2023-04-27T15:21:52Z	
https://api.github.com/repos/apache/j...	Show full URL of sites tested under Rec...	alexsch01	open	2023-04-26T12:58:44Z	comment1526149361:Steps to reprodu...
https://api.github.com/repos/apache/j...	Open Model Thread Group, not able to ...	heysarthak	open	2023-04-26T11:00:08Z	
https://api.github.com/repos/apache/j...	An OOM occurred when uploading the f...	wangchy0927	open	2023-04-24T06:08:17Z	comment1519638762:How many virtual...
https://api.github.com/repos/apache/j...	Move shared test code to testkit module	vlsi	open	2023-04-24T05:18:13Z	
https://api.github.com/repos/apache/j...	JMeterTreeNode addComponent overl...	3dgiordano	open	2023-04-20T16:36:03Z	comment1517616203:In other compon...
https://api.github.com/repos/apache/j...	When I used the SaveService. loadTree ...	linavaos	open	2023-04-19T03:22:53Z	
https://api.github.com/repos/apache/j...	Escape control characters in CSV log file	FSchumacher	open	2023-04-15T15:41:07Z	comment1516489111:I wonder, if we sh...
https://api.github.com/repos/apache/j...	only one BackendListener is effective	MrZbb	open	2023-04-15T13:10:14Z	

(c) List of Issue Descriptions

Figure 2: Screenshots of the ViolationNoter tool

that such projects are representative for developers to understand how violations are introduced and fixed.

We also try to cover a variety of business domains with the projects because we wish to include as many types of violations as possible. If the project is only from a few business domains, some rule types of the violations may not be present. Hence, we extract topics from the *descriptions* of the GitHub projects and select projects with different topics to achieve business variety.

We deliberately choose a mix of projects that did and did not use SonarQube because we plan to use SonarQube as the primary analysis tool in our study and wish to see if the violations detected can be universally applicable to other projects and other SATs. There are specific configuration files in open-source projects that act as a clue for whether the project has used certain SATs. The descriptions in the commit message or code review discussions, such as "Just fix warnings of Sonar" may also reveal the fact that the developers were using certain SATs. These clues enable us to find projects with different ways of code quality control.

Finally, eight open-source Java projects are selected based on the above criteria. Among the eight projects, four primarily use SonarQube as the code quality control tool, one uses FindBugs, and one uses PMD. The details of the selected projects are listed in Table 1. We adopt the 452 rules officially recommended by SonarQube that generally apply to most projects. After analyzing all commits for these projects, we collected a total of 82,283 distinct violations, of which 95.7% belong to the category Code Smell (CS), 3.5% belong to Bug (B), 0.7% belong to Security Hotspot (SH), and 0.1% belong to Vulnerability (V). A total of 269 rules are involved in all violations.

4.2 The Violation Benchmark

We choose in our research team two postgraduate students and one doctoral candidate, majored in Software Engineering, as the annotators. They all have at least two years of Java development experience and are familiar with the mechanism of static analysis and the SonarQube tool. With one-month effort of the three students, an initial version of the violation benchmark containing 1,650 fully-validated violations is established. The Cohen's Kappa coefficient values of their annotations were 0.88 for violation priority, 0.94 for actionability, and 0.98 for authenticity, showing the strong consistency of the violation annotation between different annotators.

These violations cover 156 different rules and encompass all violation categories. We describe our dataset from the following five dimensions, as shown in Table 2. Each dimension includes the distribution proportion and quantity of violations. The corresponding classifications are shown in parentheses. For example, CS in Rules represents how many rules exist in the category of Code Smell. The dataset and more detailed statistics are publicly available at our website[9].

We find the dataset meets the following requirement [18, 36, 51] as a benchmark.

- Representativeness. We collected violations from real-world high-quality open-source projects. The violations are all manually validated based on concrete evidence in the development process. The number of validated violations is now 4.6 times the previous benchmark dataset [18], presenting a variety of violations in real-world software development.
- Diversity. The violations are of various rules, categories, and severity/priorities.
- Portability. The violations and the accompanied labels and information is used for facilitating and evaluating various violation postprocessing techniques.
- Accessibility. The violations, along with the original projects, static analysis tools, and other sources of information (i.e., commit histories and issue descriptions), are all publicly available.
- Scalability/Fairness. Although the benchmark is currently not at scale, the VioBench framework supports continuous collection and annotation of violations. The representation of the violations is not language specific and thus supports violations collected from projects of other programming languages. The framework can be extended to the crowd-sourcing mode with annotation quality control such that the bottleneck of manual validation can be broken.

[9]https://sites.google.com/view/viobench

Table 1: Projects used to construct the Benchmark

Projects	Stars	History	Commits	Size(LOC)	Developers	Topic	Issues	PR	SATs
pdfbox	1,960	2008/2/10-2023/5/6	11,221	167,514	6	content,library	5,590	155	SonarQube
calcite	3,798	2012/4/20-2023/5/5	5,196	718,782	310	sql,big data,hadoop	5,598	3,174	SonarQube
jmeter	6,930	1998/9/3-2023/5/5	17,807	147,496	59	performance,test	5,054	749	SonarQube
dolphin	10,320	2019/3/1-2023/5/5	7,777	142,672	456	work/air flow,job-scheduler	6,321	7,111	SonarQube
curator	2,944	2011/7/14-2023/5/6	2,794	60,565	121	consensus,zookeeper	670	459	N/A
gson	22,042	2008/9/1-2023/5/4	1,813	31,023	141	(de)serialization,json	1,525	787	N/A
maven	3,585	2003/9/1-2023/5/5	11,804	88,975	169	build-management	7,773	1,050	FindBugs
jetcache	3,885	2013/9/10-2023/4/26	1138	19,164	25	spring,cache,redis	721	52	PMD

Table 2: Statistics of the Benchmark

Dimension	Proportion	Quantity
TP/FP	0.96/0.04	1510/70
Actionable/Non-actionable	0.56/0.44	887/693
Priority(6/5/4/3/2/1)	0.03/0.07/0.14/0.55/0.07/0.14	25/60/122/486/66/128
Survival/Closed	0.31/0.69	470/1040
Category(CS/B/SH/V)	0.74/0.19/0.05/0.02	1171/300/86/23
Rules(CS/B/SH/V)	0.7/0.19/0.07/0.04	109/30/11/6

4.3 Discussions

In this subsection, we discuss the potential applications of VioBench, its relationship with other language ecosystems, and some of its key limitations.

4.3.1 Potential Applications VioBench and the Benchmark. We see several opportunities to apply VioBench in future static analysis violation postprocessing research. First, a large number of violation postprocessing techniques are designed to reduce or eliminate the effort of developers to review violations. However, evaluating them is currently limited by small data sets and inconsistent standards. VioBench constructs datasets that are easy to scale, and standard annotating processes facilitate the construction of strongly consistent data, making them ideal for postprocessing techniques evaluation.

Second, the raw data collected by VioBench contains a complete history of violations and violation-supportive data that can be reused to support various postprocessing technologies. For example, mining the characteristics of actionable violations, predicting the actionability of violations, mining the repair mode of violations, and exploring under what circumstances the same type of violations will cause real problems to optimize the priority of violations, etc.

Finally, VioBench can also be used to deeply study and analyze the relationship between static analysis violations and other software development processes in real projects (i.e., exploring how to use SATs to support the review process by reviewing the relationship between violations and code review comments), and supporting various empirical studies, such as the developer's characteristics of fixing violations in practice.

4.3.2 Supporting Other Programming Languages. Although we use open-source Java projects for establishing the benchmark, VioBench is not dependent on programming languages. Our static analysis tool of choice, SonarQube, is not limited to Java but also analyzes other programming languages, such as C#. In addition, the violation benchmark dataset we built has a uniform representation that can be adapted to the output of other SATs. The intergrated ViolationTracker tool is also language-independent and can be used for violation history tracing in other languages. On the other hand, our dataset constructed by VioBench is inevitably threatened by the chosen projects, limiting the ability to generalize conclusions to projects in different programming languages. However, the fact that the eight projects we selected are large, popular, active, domain-rich, and containing related supportive data partially mitigates the threats to generalization.

4.3.3 Limitations. We identify the following limitations in the initial version of the benchmark.

- The benchmark dataset is currently not considered large. To mitigate this limitation, we provide detailed documentation of the supporting tools (e.g., an IDE plugin) and criteria for manual validation. We also make the code and data open-source so that future research can effectively expand upon our work.
- The violation annotation process may require strong software development skills and could be affected by the subjective judgment of the annotators. Therefore, we provide training and testing guidelines for annotators to minimize the subjective impact. Double annotation and consistency checking mitigate the threat of human-related errors in the annotation.

5 PRELIMINARY CASE STUDY

In this section, we introduce a preliminary case study by applying state-of-the-art postprocessing techniques on the benchmark dataset constructed by VioBench. We explore the effectiveness of two series of violation postprocessing techniques: violation filtering (i.e., identifying (un)actionable violations) and violation prioritizing (i.e., ranking violations before reporting to developers), which are two typical application scenarios in real-world development process. This case study replicates the experiments with the SOTA techniques [23, 34, 58, 60] in violation filtering and prioritization. We also discuss the quality of our benchmark and potential improvement and try to answer the following two research questions.

- RQ1: How effective are the SOTA violation filtering techniques in identifying actionable and unactionable violations with the benchmark?
- RQ2: How effective are the SOTA violation prioritizing techniques perform in determining violation priorities with the benchmark?

By answering RQ1, we aim at the first-hand results of the performance of the SOTA techniques. Existing work has reported strong effectiveness in violation filtering. For example, recent studiess [60–62] with machine learning techniques (e.g., support vector machine or SVM) have achieved up to 98% precision and 96% recall. The near-perfect results, however, are challenged by the inaccurate labels of (un)actionable in the dataset [23]. Our replication study enables us to investigate the possible reasons for any deviation between our result and the literature's result and thus may help identify potential improvements in violation filtering techniques.

By answering RQ2, we aim to understand any disagreement among various violation prioritizing techniques [42]. The state-of-the-art prioritizing techniques follow two technical strategies. One is to leverage multiple static analysis tools and synthesize the ranking of these tools [13, 26, 35, 39, 45, 48, 49, 59], which we call the Multiple-Tools strategy. The other [13, 26, 35, 39, 45, 48, 49, 59] is based on analyzing the evolutionary history of the violations with predefined heuristics (such as higher priority with recent changes). We call this strategy History-Aware. Our replication study is intended to identify differences in these approaches, identify possible weaknesses, and improve chances for violation prioritizing.

5.1 Selecting Target Postprocessing Techniques

5.1.1 Violation Filtering. The violation filtering techniques, including clustering, pruning, and automatic false positives elimination (AFPE), mostly rely on machine learning models [16, 19, 20, 25, 31, 38, 41, 54, 60, 62, 64, 67, 69]. Wang et al. [58] identified 23 "Golden Features" that later widely used for identifying actionable violations [60–62]. In our replication case study, we use the linear SVMs recommended by previous work [60] to recognize actionable violations. For the features that lead to data leakage among the 23 features, we reimplement these features using the method recommended by Kang et al. [23]. Instead of using the data provided in the previous studies, we use our manually annotated benchmark dataset. We do not employ a deep-learning model for comparison simply because our violation data are intrinsically very simple and not yet of large enough scale for a heavy-weight learning model.

5.1.2 Violation Prioritizing. In this work, we use the Severity attribute provided by SonarQube as the priority as a baseline. We also replicate violation prioritizing techniques with the Multiple-Tools and History-aware strategies.

For the Multiple-Tools strategy, we use different static analysis tools to combine and rank the results for the rules. The Multiple-Tools strategy merging results enables cross-validation of results from different tools, greatly increasing or decreasing confidence in false positives and negatives [42]. In this paper, we rank the priority of rules based on their severity or priority in different SATs for the same rule. We use four commonly used Java SATs, i.e., SonarQube, FindBugs, checkStyle, and PMD, to rank the priority of rules. In contrast, we also evaluated the default priority that SonarQube assigns to the rule.

For History-Aware strategy, we reimplemented the approach proposed by Liu et al. [34], which is to rank violations by violation types. The key in this approach is the Fluctuation Ratio (FR) measurement method, which evaluates the ranking differences between introducing and fixing a violation of a specific violation type. Specifically, given a violation type T, we denote the number of introductions of T as T_i, the number of fixes as T_s, the total number of introductions of all types as ALL_i, and the total number of fixes of all types as ALL_s. The FR of type T (T_f) is calculated as follows:

$$T_f = \frac{T_s}{ALL_s} \div \frac{T_i}{ALL_i} \quad (1)$$

The higher the calculated value of T_f, the higher the priority of the violation. In this paper, we used the history of the violation in our benchmark dataset to calculate T_f. It is worth noting that, instead of ranking violation *types*, there are quite a few studies for ranking *violations*. However, these studies are reported to be based on problematic original data or proved to be inaccurate [5, 17, 24, 27], or only valid when real errors and user feedback are available. Considering the applicability of these approaches, we opt to employ violation *type* ranking in this work.

5.2 Evaluation Metrics

5.2.1 Violation Filtering. To analyze the effectiveness of the SVM-based method for identifying violations with VioBench, we use Precision (P), Recall (R), F1-score (F1), Accuracy (A), and Area Under Curve (AUC), which are also used in previous studies [23, 58, 60, 62]. These values are specialized in our specific context of (un)actionable violation evaluation, as follows:

- $P_A = TAV/(TAV + FAV)$; $R_A = TAV/AAV$; $F1_A = 2P_AR_A/(P_A + R_A)$;
- $P_N = TNV/(TNV + FNV)$; $R_N = TNV/ANV$; $F1_N = 2P_NR_N/(P_N + R_N)$;
- $A = (TAV + TNV)/(AAV + ANV)$

where:

- TAV is the number of actionable violations correctly predicted as actionable (True actionable violations);
- FAV is the number of unactionable violations wrongly predicted as actionable (False actionable violations);
- TNV is the number of unactionable violations correctly predicted as unactionable (True unactionable violations);
- FNV is the number of actionable violations wrongly predicted as unactionable (False unactionable violations).
- AAV and ANV refer to the numbers of All actionable violations and All unactionable violations that are the manually labeled ground truth in the benchmark.

The final indicator AUC is a measure of the predictive ability of a machine learning method to distinguish between true and false values. The AUC ranges from 0 (worst discrimination) to 1 (perfect discrimination). The value of AUC represents the area under the true-positive-rate-versus-false-positive-rate curve. The AUC of a straw classifier that always outputs a single label is 0.5. Similar to previous studies [23, 60, 62], we use a training and testing set and train a model for each project. To train the SVM model, the ratio of the training set to the testing set was 1 to 1 in the previous study; however, in this case, the violations were all predicted to be actionable or unactionable. After trying the proportion of various training sets and testing sets, we found that 80% of the data as the training set and 20% of the data as the testing set had the best effect. And the result can be referred to Sec 5.3.2

5.2.2 Violation Prioritizing. In the benchmark, we only mark the violation instance's specific priority, not the rule's priority. For the priority ground truth values (GTVs) of specific rules in a project, we calculate the average priority of a rule by summing the priorities of all instances of the same rule, dividing by the number of instances, and ordering the average priority of rules. This reflects the average priority of a rule, and violations with higher priorities are placed at the front of the queue.

We use the Spearman rank correlation coefficient between the ranking results and the ground-truth ranking, the same measure as previous research [18], to evaluate the effectiveness of violation ranking. The Spearman rank correlation measures the strength of the monotonic relationship between two sequences, reflecting what extent two sequences maintain consistent trends in increasing or decreasing. The distance between the same violation in the two ranking results is calculated by the ranking or position of the violation in the corresponding ranking result.

5.3 Case Study Results

We report the results of the preliminary case study as follows.

5.3.1 RQ1: Violation Filtering. There are 1250 violations in our training data for eight projects, of which 700 are actionable. For the testing set, there are 311 violations, 178 of which are actionable. The remaining violations, i.e., 19, were not used because we could not capture all 23 features.

Table 3 shows the percentage of actionable violations in the dataset and the prediction effectiveness of the evaluation metrics. The first two columns *Act.% in Training* and *Act.% in Testing* refer to the proportion of actionable violations in the training and testing set, respectively. The last two columns F_{da} and F_{dn} are the *F1 score*s of the *dummy baseline*, which blindly predicts all violations actionable or unactionable, respectively.

Overall, the linear SVM model trained on the benchmark achieves an accuracy of 0.688. The accuracy of the strawman method is based on the proportion of the ground-truth (un)actionable violations. Therefore, for the always-actionable dummy predictor, the accuracy should be 0.524 (the same as the Act.% in Testing); for the always-unactionable dummy predictor, the accuracy is 0.476 (=1-0.524). We also find slightly-higher F1 scores ($F1_A$ and $F1_N$) in the result of the SVM-predictor than that of the dummy-predictors (F_{da} and F_{dn}). We see that the result of the SVM predictor is better than that of the strawman baseline, indicating that the SVM predictor with gold features did play a role in identifying the actionability of violations. However, the improvement is very weak.

In addition, the 0.709 F1 score for actionable violation identification was slightly superior to Kang et al. 's 0.64 for the results of training with clean data, i.e., removal of unconfirmed actionable violations [23]. Part of the reason may be the adoption of different projects and the distribution ratio of training and testing sets. More importantly, our extraction of violation features depends on real evolutionary history, which suggests that more accurate data may improve the effectiveness of SVMs supported by golden features. However, using the golden features of violations and machine learning algorithms to identify actionable violations requires further exploration and improvement.

5.3.2 RQ2: Violation Prioritizing. Table 4 presents the Spearman rank correlation values between the two violation prioritization techniques and the ground truth values (GTVs) for the VioBench project. A positive correlation indicates that the specified priority is similar to the GTVs priority, while a negative correlation indicates that the specified priority is opposite to the GTVs priority. The closer the correlation is to 1 or -1, the stronger the match or opposition of the specified priority. Cells with one asterisk (*) have significant correlations at the 0.05 level, while cells with two asterisks (**) have significant correlations at the 0.01 level.

Although the effectiveness of the three methods on violation priority shows a certain positive correlation for specific projects (i.e., the History-aware method shows a strong positive correlation with GTVs in the curator project), these methods cannot reflect the GTVs on the whole. There was little difference between the history-aware-based ranking method and the SonarQube default priority in sorting rules, i.e., history-aware outperformed the SonarQube default priority in four projects. The SonarQube default rule priority is slightly better than the multiple-tool-based in violation priority ranking. Given that SonarQube prioritized violations based on how harmful and likely they are to cause problems, closer to the developer's need. Therefore, SonaQube's violation priority is more reliable than other tools. However, SonaQube only considers the possibility that a certain rule may cause a problem and ignores the context in which the violation instance is located, so the default priority differs quite a bit from GTVs.

The history-aware-based violation ranking method reflects the priority of violation repair to some extent, but it only considers the number of violations introduced and fixed, ignoring the time. This can result in rules with many violation introductions and fixes but with a long fix time, having a high priority in history-aware-based ranking. If the rule's priority is higher, it will be fixed, *and* more quickly. However, the reverse is different. Previous studies [65] have shown that simple and easily perceived violations will be fixed more often. Therefore, judging the priority of violations based on the number of violations introduced and repaired is not precise enough in prioritizing rules. Therefore, when prioritizing violations, it is worth further exploring the chances of automating violation ranking by considering the time of violation fixing on top of the number of violations introduced and fixed, and further by considering the source code context of the violations.

5.4 Threats to Validity of the Case Study

The main threat to the validity of our experiment is twofold. First, the number of violations we use for training and testing for actionable violation identification is not large. Labeling violations are time-consuming and involve understanding the project, rules, code, relevant review comments, and possibly associated issues. However, previous studies [60] claimed that training on a small data set could achieve good results, and the amount of data we used is equal to the amount of data trained by Kang et al. [23].

Second, for multi-tool-based violation prioritization, since many rules are only present in SonarQube (which has more rules than FindBugs and PMD), SonarQube's violation priority may significantly impact prioritization more than other tools. However, their priority differences do not vary significantly for those violation types present in all SATs. In addition, we did not use the entire history for history-based violation prioritization. But more importantly, we have to ensure the accuracy of the history data, so

Table 3: The Result of Violation Filtering

Projects	Act.% in Training	Act.% in Testing	P_A	R_A	$F1_A$	P_N	R_N	$F1_N$	A	AUC	F_{da}	F_{dn}
pdfbox	0.763	0.773	0.842	0.941	0.889	0.667	0.400	0.500	0.818	0.671	0.872	0.370
calcite	0.364	0.381	0.800	0.500	0.615	0.750	0.923	0.828	0.762	0.712	0.552	0.765
jmeter	0.650	0.780	0.771	0.844	0.806	0.167	0.111	0.133	0.683	0.477	0.877	0.360
dolphinscheduler	0.333	0.318	0.333	0.286	0.308	0.688	0.733	0.710	0.591	0.510	0.483	0.811
curator	0.430	0.455	0.462	0.600	0.522	0.556	0.417	0.476	0.500	0.508	0.625	0.706
gson	0.590	0.632	0.800	0.667	0.727	0.556	0.714	0.625	0.684	0.691	0.774	0.538
maven	0.457	0.450	0.727	0.889	0.800	0.889	0.727	0.800	0.800	0.808	0.621	0.710
jetcache	0.200	0.182	0.200	0.250	0.222	0.824	0.778	0.800	0.682	0.514	0.308	0.900
Total	0.472	0.524	0.692	0.727	0.709	0.682	0.644	0.663	0.688	0.661	0.688	0.645

Table 4: Spearman Rank Correlation

Projects	SonarQube	Multiple-Tools	History-aware
pdfbox	-0.0009	0.1792	0.0273
calcite	0.1769	0.0306	-0.3593
jmeter	0.3134*	0.3238*	0.1493
dolphin	-0.2665	0.1676	0.1934
curator	0.2370	-0.0390	0.5347**
gson	0.1819	-0.1323	-0.6432**
maven	0.6944*	0.2374	-0.1793
jetcache	-0.0524	-0.3231	0.1098

we constructed the priority based on the history of violations in VioBench.

6 RELATED WORK

As a benchmark suite of static analysis violations, VioBench is related to several lines of research.

Bug/Violation Dataset. Bug datasets are the foundation for various tasks, providing experience and experimental foundations. Existing research on bug datasets can be roughly classified into actual bugs [7, 8, 10, 14, 22, 36], which may cause projects to fail some test cases, and static analysis violations [4, 9, 18, 34, 36, 37, 44, 66] which may not result in test failures. The dataset most relevant to our study is the benchmark used for evaluating violation-related techniques. Section 2.2 outlines the existing benchmark datasets related to violations. In comparison to these benchmark suites, our VioBench benchmark suite has the following advantages: (1) It clearly records the introduction and repair information of violations, as well as the reasons for data labeling; (2) All violations come from real and popular open-source repositories, and violations types are rich and complete; (3) It provides a convenient plugin to label, easily expands the data set, and is constructed in a manner derived from the majority consensus of existing research; (4) It includes raw data, analysis, and all data after labeling for easy reuse and review.

Evaluation of Post-processing Techniques. Several studies [1, 18, 33, 63] have evaluated violation post-processing techniques. Allier et al. [1] proposed a framework to compare different violation ranking algorithms to determine the best approach for ranking violations. For comparison, they selected six algorithms from the literature and used a benchmark dataset covering two programming languages (Java and Smalltalk) and three rule checkers (FindBugs, PMD, SmallLint). However, they used FAULTBENCH as the evaluation dataset, whose primary data for constructing the data set is inaccurate. In another study, Liang et al. [33] proposed an automatic method based on "generic error-correlated lines" to construct an effective warning priority training set. Specifically, they first identified generic bug-related lines using an Issue-Tracking System, combined SATs to find violations related to actual errors, and selected three categories of influencing factors as input attributes for the training set. Consistent with the first step in constructing violation priorities in VioBench, the most urgently fixed violations are those related to bugs. Their empirical evaluation showed that, with the help of the constructed training set, there is high accuracy in prioritizing top n violations for the project. In contrast to the above studies, we focus on constructing a public dataset for supporting violation post-processing methods. We evaluated three different violation post-processing techniques, revealing their respective strengths and weaknesses while demonstrating the applicability of VioBench.

7 CONCLUSION AND FUTURE WORK

We propose a semi-automated framework VioBnech, which can automatically collect violations and supportive data and easily facilitate the expansion of data sets. We solve the possible impact of annotation bias in the manual annotation process through standardized processes, such as the selection of annotators, training, unified annotation action in a plugin tool, annotation result verification, etc. Using VioBench, we constructed a benchmark with eight projects and labeled 1,650 violations. Based on this benchmark, we conducted a preliminary case study of three violation postprocessing techniques, highlighting potential improvement opportunities for violation filtering and prioritization.

Evaluations of other violation post-processing techniques are planned in future research and can be supported by the enlarging benchmark dataset constructed by the VioBench framework. We are also developing a web-based tool and planning to open it to the public to involve more developers/users for benchmark construction and to explore better violation post-processing approaches.

REFERENCES

[1] Simon Allier, Nicolas Anquetil, André C. Hora, and Stéphane Ducasse. 2012. A Framework to Compare Alert Ranking Algorithms. In *WCRE*. IEEE Computer Society, 277–285.

[2] Pavel Avgustinov, Arthur I. Baars, Anders Starcke Henriksen, R. Greg Lavender, Galen Menzel, Oege de Moor, Max Schäfer, and Julian Tibble. 2015. Tracking Static Analysis Violations over Time to Capture Developer Characteristics. In *ICSE (1)*. IEEE Computer Society, 437–447.

[3] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the State of Static Analysis: A Large-Scale Evaluation in Open Source Software. In *SANER*. IEEE Computer Society, 470–481.

[4] G Boetticher. 2007. The PROMISE repository of empirical software engineering data. *http://promisedata. org/repository* (2007).

[5] Cathal Boogerd and Leon Moonen. 2006. Prioritizing Software Inspection Results using Static Profiling. In *SCAM*. IEEE Computer Society, 149–160.

[6] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *ASE*. ACM, 332–343.

[7] Cristina Cifuentes, Christian Hoermann, Nathan Keynes, Lian Li, Simon Long, Erica Mealy, Michael Mounteney, and Bernhard Scholz. 2009. BegBunch: benchmarking for C bug detection tools. In *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*. 16–20.

[8] Valentin Dallmeier and Thomas Zimmermann. 2007. Extraction of bug localization benchmarks from history. In *ASE*. ACM, 433–436.

[9] Georgios Digkas, Mircea Lungu, Paris Avgeriou, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. 2018. How do developers fix issues and pay back technical debt in the Apache ecosystem?. In *SANER*. IEEE Computer Society, 153–163.

[10] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empir. Softw. Eng.* 10, 4 (2005), 405–435.

[11] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. 2022. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Trans. Software Eng.* 48, 3 (2022), 835–847.

[12] Frank Elberzhager, Jürgen Münch, and Vi Tran Ngoc Nha. 2012. A systematic mapping study on the combination of static and dynamic quality assurance techniques. *Inf. Softw. Technol.* 54, 1 (2012), 1–15.

[13] Lori Flynn, William Snavely, David Svoboda, Nathan M. VanHoudnos, Richard Qin, Jennifer Burns, David Zubrow, Robert Stoddard, and Guillermo Marce-Santurio. 2018. Prioritizing alerts from multiple static analysis tools, using classification models. In *SQUADE@ICSE*. ACM, 13–20.

[14] Claire Le Goues, Neal J. Holtschulte, Edward K. Smith, Yuriy Brun, Premkumar T. Devanbu, Stephanie Forrest, and Westley Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Trans. Software Eng.* 41, 12 (2015), 1236–1256.

[15] Andrew Habib and Michael Pradel. 2018. How many of all bugs do we find? a study of static bug detectors. In *ASE*. ACM, 317–328.

[16] Quinn Hanam, Lin Tan, Reid Holmes, and Patrick Lam. 2014. Finding patterns in static analysis alerts: improving actionable alert ranking. In *MSR*. ACM, 152–161.

[17] Sarah Smith Heckman. 2007. Adaptively ranking alerts generated from automated static analysis. *ACM Crossroads* 14, 1 (2007).

[18] Sarah Smith Heckman and Laurie A. Williams. 2008. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *ESEM*. ACM, 41–50.

[19] Sarah Smith Heckman and Laurie A. Williams. 2009. A Model Building Process for Identifying Actionable Static Analysis Alerts. In *ICST*. IEEE Computer Society, 161–170.

[20] Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. 2017. Machine-learning-guided selectively unsound static analysis. In *ICSE*. IEEE / ACM, 519–529.

[21] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *ICSE*. IEEE Computer Society, 672–681.

[22] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *ISSTA*. ACM, 437–440.

[23] Hong Jin Kang, Khai Loong Aw, and David Lo. 2022. Detecting False Alarms from Automatic Static Analysis Tools: How Far are We?. In *ICSE*. ACM, 698–709.

[24] Sunghun Kim and Michael D. Ernst. 2007. Which warnings should I fix first?. In *ESEC/SIGSOFT FSE*. ACM, 45–54.

[25] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. 2017. Learning a classifier for false positive error reports emitted by static code analysis tools. In *MAPL@PLDI*. ACM, 35–42.

[26] Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, and Ming Zhu. 2007. ISA: a source code static vulnerability detection system based on data fusion. In *Infoscale (ACM International Conference Proceeding Series, Vol. 304)*. ACM, 55.

[27] Ted Kremenek and Dawson R. Engler. 2003. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *SAS (Lecture Notes in Computer Science, Vol. 2694)*. Springer, 295–315.

[28] Rahul Kumar and Aditya V. Nori. 2013. The economics of static analysis tools. In *ESEC/SIGSOFT FSE*. ACM, 707–710.

[29] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for static analysis of Java reflection: literature review and empirical study. In *ICSE*. IEEE / ACM, 507–518.

[30] Lucas Layman, Laurie A. Williams, and Robert St. Amant. 2007. Toward Reducing Fault Fix Time: Understanding Developer Behavior for the Design of Automated Fault Detection Tools. In *ESEM*. ACM / IEEE Computer Society, 176–185.

[31] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. 2019. Classifying False Positive Static Checker Alarms in Continuous Integration Using Convolutional Neural Networks. In *ICST*. IEEE, 391–401.

[32] Valentina Lenarduzzi, Francesco Lomio, Heikki Huttunen, and Davide Taibi. 2020. Are SonarQube Rules Inducing Bugs?. In *SANER*. IEEE, 501–511.

[33] Guangtai Liang, Ling Wu, Qian Wu, Qianxiang Wang, Tao Xie, and Hong Mei. 2010. Automatic construction of an effective training set for prioritizing static analysis warnings. In *ASE*. ACM, 93–102.

[34] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. 2021. Mining Fix Patterns for FindBugs Violations. *IEEE Trans. Software Eng.* 47, 1 (2021), 165–188.

[35] Bailin Lu, Wei Dong, Liangze Yin, and Li Zhang. 2018. Evaluating and Integrating Diverse Bug Finders for Effective Program Analysis. In *SATE (Lecture Notes in Computer Science, Vol. 11293)*. Springer, 51–67.

[36] Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. 2005. Bugbench: Benchmarks for evaluating bug detection tools. In *Workshop on the evaluation of software defect detection tools*, Vol. 5. Chicago, Illinois.

[37] Diego Marcilio, Rodrigo Bonifácio, Eduardo Monteiro, Edna Dias Canedo, Welder Pinheiro Luz, and Gustavo Pinto. 2019. Are static analysis violations really fixed?: a closer look at realistic usage of SonarQube. In *ICPC*. IEEE / ACM, 209–219.

[38] Laura Alejandra Martínez-Tejada, Yasuhisa Maruyama, Natsue Yoshimura, and Yasuharu Koike. 2020. Analysis of Personality and EEG Features in Emotion Recognition Using Machine Learning Techniques to Classify Arousal and Valence Labels. *Mach. Learn. Knowl. Extr.* 2, 2 (2020), 99–124.

[39] Na Meng, Qianxiang Wang, Qian Wu, and Hong Mei. 2008. An Approach to Merge Results of Multiple Static Analysis Tools (Short Paper). In *QSIC*. IEEE Computer Society, 169–174.

[40] Audris Mockus and David M. Weiss. 2000. Predicting risk of software changes. *Bell Labs Tech. J.* 5, 2 (2000), 169–180.

[41] Tukaram Muske. 2014. Improving Review of Clustered-Code Analysis Warnings. In *ICSME*. IEEE Computer Society, 569–572.

[42] Tukaram Muske and Alexander Serebrenik. 2023. Survey of Approaches for Postprocessing of Static Analysis Alarms. *ACM Comput. Surv.* 55, 3 (2023), 48:1–48:39.

[43] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *PLDI*. ACM, 308–319.

[44] Paulo Jorge Costa Nunes, Iberia Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. 2018. Benchmarking Static Analysis Tools for Web Security. *IEEE Trans. Reliab.* 67, 3 (2018), 1159–1175.

[45] Paulo Jorge Costa Nunes, Ibéria Medeiros, José Fonseca, Nuno Neves, Miguel Correia, and Marco Vieira. 2019. An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. *Computing* 101, 2 (2019), 161–185.

[46] Yu Qu, Qinghua Zheng, Jianlei Chi, Yangxu Jin, Ancheng He, Di Cui, Hengshan Zhang, and Ting Liu. 2021. Using K-core Decomposition on Class Dependency Networks to Improve Bug Prediction Model's Practical Performance. *IEEE Trans. Software Eng.* 47, 2 (2021), 348–366.

[47] Xiaoxia Ren, Ophelia C. Chesley, and Barbara G. Ryder. 2006. Identifying Failure Causes in Java Programs: An Application of Change Impact Analysis. *IEEE Trans. Software Eng.* 32, 9 (2006), 718–732.

[48] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. 2018. Ranking Source Code Static Analysis Warnings for Continuous Monitoring of FLOSS Repositories. In *OSS (IFIP Advances in Information and Communication Technology, Vol. 525)*. Springer, 90–101.

[49] Athos Ribeiro, Paulo Meirelles, Nelson Lago, and Fabio Kon. 2019. Ranking warnings from multiple source code static analyzers via ensemble learning. In *OpenSym*. ACM, 5:1–5:10.

[50] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing Test Cases For Regression Testing. *IEEE Trans. Software Eng.* 27, 10 (2001), 929–948.

[51] Susan Elliott Sim, Steve M. Easterbrook, and Richard C. Holt. 2003. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *ICSE*. IEEE Computer Society, 74–83.

[52] SonarSource. 2022. SonarQube. [Online] Available: https://www.sonarqube.org/. Last Accessed on: April. 2022.

[53] Alexander Trautsch, Steffen Herbold, and Jens Grabowski. 2020. A longitudinal study of static analysis warning evolution and the effects of PMD on software quality in Apache open source projects. *Empir. Softw. Eng.* 25, 6 (2020), 5137–5192.

[54] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Y. Aravkin. 2014. ALETHEIA: Improving the Usability of Static Security Analysis. In *CCS*. ACM, 762–774.

[55] Ekincan Ufuktepe, Tugkan Tuglular, and Kannappan Palaniappan. 2021. The Relation between Bug Fix Change Patterns and Change Impact Analysis. In *QRS*. IEEE, 1089–1099.
[56] Ekincan Ufuktepe, Tugkan Tuglular, and Kannappan Palaniappan. 2022. Tracking Code Bug Fix Ripple Effects Based on Change Patterns Using Markov Chain Models. *IEEE Trans. Reliab.* 71, 2 (2022), 1141–1156.
[57] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C. Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empir. Softw. Eng.* 25, 2 (2020), 1419–1457.
[58] Junjie Wang, Song Wang, and Qing Wang. 2018. Is there a "golden" feature set for static warning identification?: an experimental evaluation. In *ESEM*. ACM, 17:1–17:10.
[59] Achilleas Xypolytos, Haiyun Xu, Bárbara Vieira, and Amr M. T. Ali-Eldin. 2017. A Framework for Combining and Ranking Static Analysis Tool Findings Based on Tool Performance Statistics. In *QRS Companion*. IEEE, 595–596.
[60] Xueqi Yang, Jianfeng Chen, Rahul Yedida, Zhe Yu, and Tim Menzies. 2021. Learning to recognize actionable static code warnings (is intrinsically easy). *Empir. Softw. Eng.* 26, 3 (2021), 56.
[61] Xueqi Yang and Tim Menzies. 2021. Documenting evidence of a reproduction of 'is there a "golden" feature set for static warning identification? - an experimental evaluation'. In *ESEC/SIGSOFT FSE*. ACM, 1603.
[62] Xueqi Yang, Zhe Yu, Junjie Wang, and Tim Menzies. 2021. Understanding static code warnings: An incremental AI approach. *Expert Syst. Appl.* 167 (2021), 114134.
[63] Kwangkeun Yi, Hosik Choi, Jaehwang Kim, and Yongdai Kim. 2007. An empirical study on classification methods for alarms from a bug-finding static C analyzer. *Inf. Process. Lett.* 102, 2-3 (2007), 118–123.
[64] Jongwon Yoon, Minsik Jin, and Yungbum Jung. 2014. Reducing False Alarms from an Industrial-Strength Static Analyzer by SVM. In *APSEC (2)*. IEEE, 3–6.
[65] Ping Yu, Yijian Wu, Jiahan Peng, Jian Zhang, and Peicheng Xie. 2023. Towards Understanding Fixes of SonarQube Static Analysis Violations: A Large-Scale Empirical Study. In *SANER*. IEEE, 569–580.
[66] Ping Yu, Yijian Wu, Xin Peng, Jiahan Peng, Jian Zhang, Peicheng Xie, and Wenyun Zhao. 2023. ViolationTracker: Building Precise Histories for Static Analysis Violations. In *ICSE*. IEEE, 2022–2034.
[67] Ulas Yuksel and Hasan Sözer. 2013. Automated Classification of Static Code Analysis Alerts: A Case Study. In *ICSM*. IEEE Computer Society, 532–535.
[68] Fiorella Zampetti, Saghan Mudbhari, Venera Arnaoudova, Massimiliano Di Penta, Sebastiano Panichella, and Giuliano Antoniol. 2022. Using code reviews to automatically configure static analysis tools. *Empir. Softw. Eng.* 27, 1 (2022), 28.
[69] Yuwei Zhang, Ying Xing, Yunzhan Gong, Dahai Jin, Honghui Li, and Feng Liu. 2020. A variable-level automated defect identification model based on machine learning. *Soft Comput.* 24, 2 (2020), 1045–1061.