ViolationBase: Building Precise History of Violations For Effective Violation Evolution Manipulation

Ping Yu*
Fudan University
China
pyu17@fudan.edu.cn

Yijian Wu*†
Fudan University
China
wuyijian@fudan.edu.cn

Jian Zhang*
Fudan University
China
zhangjian16@fudan.edu.cn

Rui Song*
Fudan University
China
19212010058fudan.edu.cn

Xin Peng*
Fudan University
China
pengxin@fudan.edu.cn

Wenyun Zhao*
Fudan University
China
wyzhao@fudan.edu.cn

ABSTRACT

In this work, we introduce a tool, ViolationBase, to build precise histories of static analysis violations for effective violation evolution manipulation. It employs code entity anchoring heuristics for violation matching and considers merge commits that were ignored in existing research. In addition, ViolationBase supports incrementally parallelizing multiple versions of a single code repository, which can greatly improve the speed of building violation histories. We built a benchmark consisting of 500 manually-validated violation instances and 30 violation cases with detailed evolution history details. ViolationBase achieves over 93.6% precision and 98.0% recall on violation matching, outperforming the state-of-the-art approach and rebuilding the histories of all violation histories at a precision of 99.4%. Preliminary application on actionable violation identification shows that ViolationBase is useful for identifying actionable violations. In addition, an empirical study on 30 well-known open-source Java projects with a total of 80,335 violation cases reveals its usefulness in characterizing and understanding developers' fixing actions and providing practical implications to developers, tool builders, and researchers. The source code of ViolationBase and the benchmark dataset are available at https://github.com/FudanSELab/violationTracker , the empirical study dataset is available at https://github.com/FudanSELab/sq-violation-dataset, and the demonstration video is available at https://youtu.be/Nj0E_yPi0og.

KEYWORDS

mining code repository, static analysis violations, software maintenance

*Affiliated with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China.
†Corresponding author

ESEC/FSE 2023, 11 - 17 November, 2023, San Francisco, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Ping Yu, Yijian Wu, Jian Zhang, Rui Song, Xin Peng, and Wenyun Zhao. 2023. ViolationBase: Building Precise History of Violations For Effective Violation Evolution Manipulation. In *Proceedings of The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Automatic static analysis tools (ASATs) are widely utilized in open-source and industrial software projects to assess the quality and detect potential defects. However, the adoption of ASATs is often hindered by a variety of issues, such as improper rule configurations, a large number of violations that developers do not care about, and invalid rule priorities [7, 21, 23]. To address these issues, developers have proposed many techniques that employ violation data analysis. Specifically, violation rule configuration, e.g., filtering the rule set using code review comments [31]; violation filtering, e.g., using machine learning to filter out actionable violations which are those developers would fix [3, 5, 6, 10, 12, 16, 26, 27, 30]; violations prioritizing, e.g., using various characteristics of violations, source code, repair history to rank violations [11, 14, 15, 17–20, 25].

While the research works for static analysis violations are emerging, the community still lacks a violation evolution fact base to manipulate them in a systematic way. Studies have shown that tracking violation history can effectively help solve the aforementioned issues, as the facts of violation repair may reflect developers' subjective expectations for violation detection results [1]. Constructing violation history relies on accurate matching and meticulous handling of the evolution process. However, as code changes, violations may not only be moved, but related code and context may also be modified, making the precise construction of violation history not simple. In addition, existing violation history construction methods perform poorly in many cases, such as when dealing with violations with multiple locations, and ignore the construction of the entire evolution history, making them unsuitable for constructing longer violation evolution histories directly [1, 3, 4, 14, 22].

In this work, we propose a tool, ViolationBase, to build precise histories of static analysis violations for effective violation manipulation. ViolationBase takes a code repository as input and violation evolution histories with detailed change status as output. Technically, it contains a violation life-cycle model that captures

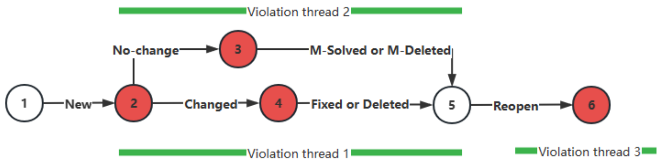


Figure 1: An Examples of a Violation Case.

the inducing and fixing of violations and tracks violation cases and violation threads. To this end, our approach addresses the technical challenges of (1) quickly obtaining historical violations of the reversion to be analyzed, (2) accurately matching the same violations between adjacent revisions, (3) merging the same violations on different branches, and (4) accurately identifying the changing status of violations.

We implement the ViolationBase tool with SonarQube for Java projects. To evaluate the effectiveness, we built manually-validated benchmark datasets of 500 violation cases introduced or fixed in the history and 159 threads of 30 violation cases. Compared to the state-of-the-art (SOA) violation tracking techniques [1], our approach achieved over 93.6% precision in violation matching and identified nearly all violation cases with the history. We also evaluated the effectiveness of the ViolationBase in identifying actionable violations and compared it with *closed-warning heuristic* method used in many studies [8, 24, 26, 27]. The results showed that the average F1 score reached 0.89, superior to *closed-warning heuristic*. Furthermore, we conducted a large-scale empirical study of 80,335 violation cases in 30 active, popular, and high-quality open-source Java projects. Our analysis of various fixed violation characteristics within violation data sets delved into why violations are fixed, which types of violations are frequently fixed, and how long it takes to fix violations following their introduction. Our findings help characterize and understand developers' repair behaviors and provide practical implications for developers, tool builders, and researchers.

2 TOOL DESIGN

We implement ViolationBase as a general framework supporting different languages with the help of multiple ASATs. In practice, ViolationBase can be integrated with version control tools and IDE or services (e.g., DevOps, GitHub), accumulating violation facts incrementally with the evolution of software. It takes a git repository as input, and the output is a set of *violation case*s. A *violation case* represents a group of similar violation instances and their history. The evolutionary history of a violation case can be seen in Figure 1. The red node represents the violation instance in this revision while the white node indicates it doesn't exist. A *violation thread* captures the life-cycle of the source code quality issues, from creation to changes and disappearance. It starts with a NEW instance and ends with a FIXED or DELETED instance, or the last commit in the maintenance history. A violation case may include multiple violation threads, as shown by the green line in Figure 1. More details can be referred to in [29]. Figure 2 shows an overview of the ViolationBase design. The ViolationBase framework consists of three high-level modules:

Violation Data Preparation. We apply the ASATs to consecutive source code revisions to collect violations from a code repository. Since it's time-consuming and resource-consuming to analyze the historical version of the repository to obtain violations of each revision, ViolationBase first makes multiple copies of the repository to prepare violations concurrently. The number of repository copies can be determined dynamically based on server resources and the number of versions that need to be analyzed. Then, it checkout each copy to the corresponding version based on the commit order that needs to be analyzed. Finally, it leverages git metadata to identify the files that have changed between versions and analyze each version concurrently by specifying the modified files to collection violations.

Violation Matching and Tracing. ViolationBase traverses the commit history and matches violation instances per pair of revisions with a parent-of-relation. Thanks to the version control system, violation instances in the files that were not changed will be automatically matched, and it only does matching for those in the changed files. After this step, each violation instance is assigned an original matching status regarding the parent and child revisions. Then, ViolationBase mainly creates the violation cases and updates the matching status per violation case. Finally, The original violation matching status is updated according to the maintenance history and persists in the data.

Violation history Manipulation. ViolationBase provides an intuitive user interface based on the constructed violations history dataset.
(1) Violation Change History Backtracking. Given the unique identifier or filter criteria for a violation, ViolationBase queries all historical versions and displays diff information.
(2) Project Quality Monitoring. ViolationBase continuously monitors project quality from different perspectives by counting the number of violations added, closed, and remained per revision at the project, file, and package levels.
(3) Developer Characteristics Analysis. In addition, ViolationBase analyzes developers' code quality and skill characteristics by analyzing the characteristics of violations that developers introduce and fix.

3 VIOLATIONBASE TOOL

3.1 Analyzing Process Dashboard

Figure 3 shows the dashboard user interface to run the violation history-building process from the Git repository. At the top of the dashboard, users can filter code repositories by ASATs and repository name. The top right section of the *Repository List* allows users to add code repositories and then select the branch to analyze, the ASAT, and the commit time to start the analysis. ViolationBase support adding code repositories to be analyzed through the local addresses. Users need to download the code repository from GitHub or GitLab to the ViolationBase deployed server. Once the code repository download is complete, ViolationBase builds the list of historical revisions to scan. At the same time, the dashboard displays repository-related information (i.e., repository name, branch, and total commits), as well as analysis progress information (i.e., the number of unanalyzed revisions, analysis status, and analysis time).

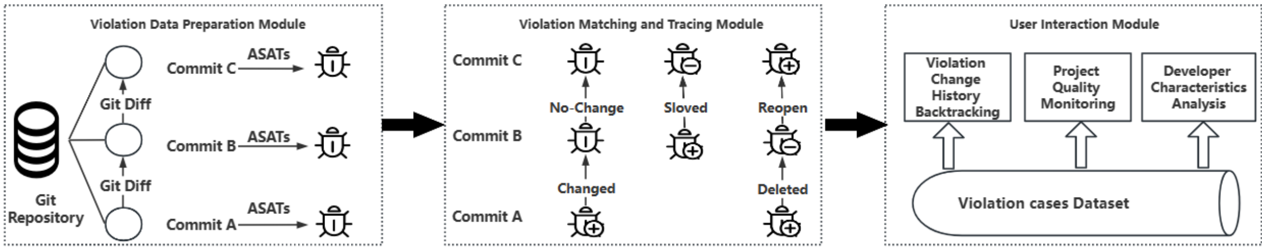


Figure 2: Design of ViolationBase

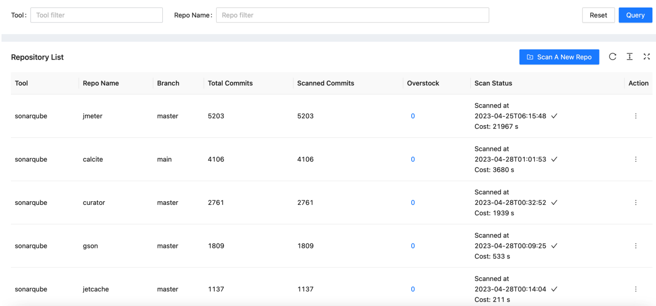
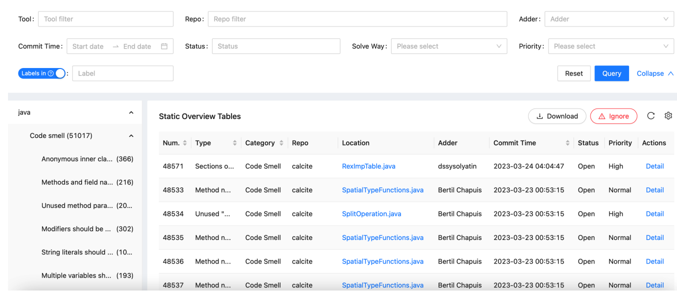
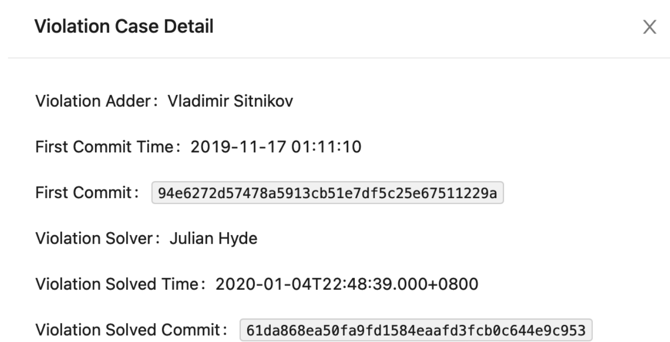


Figure 3: User interface of ViolationBase: We show repository information of building violation history



(a) Violation cases List



(b) A violation case details

(c) The evolution history of a violation case

Figure 4: User interface of violations list and history.

On the right side of the *Repository List*, users can delete, update, stop, restart, and analyze the process of the code repository.

3.2 Violation Details and History

During the analyzing process, we further allow the users to investigate any violation cases on how it is introduced and closed or fixed on the commit history, as shown in Figure 4(a). At the top of Figure 4(a), the user can filter the violation list by the ASATs, repository name, the violation introducer, the status of the violation (i.e., open and closed), the way of closing the violation (i.e., file deletion and code modification), and the violation priority. The violation list shows the violation's type, category, and repair priority, the violation introducer, introduction time, and the file and code repository to which it belongs. The right of the violation list can be used to view the violation history and details as shown in Figure 4(b).

Moreover, the user can further investigate the code and its diff information by clicking the file in the violation list. In addition, we can also choose the commits (i.e., violation fixing commit, violation-introducing commit, and the changing commit) and show the historical violation process of evolution on the left. The left-hand side of Figure 4(c) shows simplified commit history with only the changed commits; on the right is a differencing view of source code regarding the violation. More details of violation history manipulation can be found on our website https://github.com/FudanSELab/violationTracker.

4 EXPERIMENT

To evaluate the capabilities of ViolationBase as a whole, we asked three different research questions. We evaluated: (1) the effectiveness of ViolationBase in matching violation instances and violation case tracking, (2) the code modification characteristics of fixed violations and the usefulness of violation histories in identifying actionable violations, and (3) the characteristics analysis of violation fixes. The results here are summarized from the two original papers [28, 29].

4.1 Results (RQ1): Effectiveness

Results (RQ1): Effectiveness of Matching Violation Instances and Tracking Violation Cases. We present the results of matching violation instances, including precision, recall, and F1-score. Our approach, ViolationBase, outperforms the state-of-the-art method [1] with significantly higher precision and F1-score. ViolationBase achieves 0.959 precision and 0.981 recall for NEW violation instances, while for CLOSED violation instances, it achieves 0.901 precision and 0.98 recall in the benchmark dataset. The average F1-score is 0.96, surpassing SOA's average F1-score of 0.58. The improvement is mainly due to our consideration of best-match mechanisms with multiple violation instances and locations for each violation instance. Moreover, we report the number of violation cases and corresponding threads in each project in the benchmark dataset to evaluate the effectiveness of tracking violation cases. Our results show that ViolationBase successfully detected all 30 violation cases and 99% (158/159) violation threads. We also calculated the lifespan (in days) of all violation threads and found that the lifespan of the threads detected by ViolationBase is accurate in the length of days. 99% of the reported threads are precise in terms of the length of lifespan, which shows that ViolationBase effectively constructs the histories of violation cases.

4.2 Results (RQ2): Characteristics and Usefulness

Results (RQ2): The Code Modification Characteristics of Fixed Violations and Usefulness of Violation Histories in Identifying Actionable Violations. We investigate what kinds of code modifications indicate that a closed violation was fixed, and the fixed violation can be treated as an actionable one. The characteristics of code modification can be categorized into two groups. The first group is deletion, which involves removing files, anchors, or code. The other group is the modification, which can be further divided into related and unrelated code modifications. Our investigation discovered that Related Code modification and Unrelated Code Modification could be used to determine if closed violations were fixed. At the same time, Code Deletion can be used for a small number of specific rules. In addition, we compared the collected actionable violations' precision, recall, and F1-score with the *closed-warning heuristic* method. The results indicate that ViolationBase's automatically identified actionable violations achieved 82% precision and 96.5% recall, with an F1-score of 0.89, which outperformed the *closed-warning heuristic*.

4.3 Results (RQ3): Empirical Study

Results (RQ3): Characteristics Analysis of Violation Fixes. We have analyzed the characteristics of violation fixes, focusing on the fix rate and fix time. Our research shows that there are several factors that determine whether a violation will be fixed, including the difficulty of understanding rules, the harm caused by the violation, the context of the violation, the function and effect of the rules, and the specific developers involved. As for factors affecting the time of fixing violations, we found that around 30% of violation instances are fixed within a month, and 85% are fixed within a year. The speed of violation fixes depends on the difficulty of understanding and fixing the violation, the importance of the violation, the hints provided by development tools, and the management of developers and teams. Our findings have contributed to a better understanding of the actions taken by developers when fixing violations. This insight can be utilized by developers, tool builders, and researchers to improve software development practices.

5 RELATED WORK

The current work [1–4, 9, 13, 22] of violation tracking mainly studies the matching between violations, which uses code matching techniques and software change histories to match the code where the violation is located and its context code. Avgustinov et al. [1] proposed a tool named Team Insight, which includes three different ways(i.e., location-based violation matching, snippet-based violation matching, and hash-based violation matching) to match violations when changing files containing violations. The core of this tool is based on a combination of hash-based context matching and diff-based location matching. Based on Team Insight, Li et al. [13] introduce refactoring information and adopt the Hungarian algorithm to improve the accuracy of violation matching. Spacco et al. [22] proposed a fuzzy matcher that contains two location-based violation matching techniques. It can match violations between revisions in different source locations, even if a source code file has been moved by package renaming. Boogerd et al. [2] used diff-based matching of code snippets to track violations forward, while Kim et al. [9] used it to track solved violations backward. Based on software change histories, Heckman [4] and Hanam [3] provide violation matching heuristics but are not exact enough in violation matching. These violation tracking technologies are mainly designed for violation matching, while the ViolationBase not only provides high matching accuracy but also pays attention to the evolution process of violation to reveal the evolutionary history of violation accurately.

6 CONCLUSION

In this work, we propose ViolationBase, which can track the histories of violations by precisely matching the violation instances between adjacent revisions. Our experiments show that compared with the state-of-the-art approach, we have higher accuracy in violation matching. And we can thoroughly uncover the evolution of the violation cases. Our preliminary application show ViolationBase's effectiveness in collecting actionable violations. Based on the empirical study findings, we have identified several areas where developers, tool builders, and researchers related to static analysis violations can benefit the most.

REFERENCES

[1] 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.

[2] Cathal Boogerd and Leon Moonen. 2009. Evaluating the relation between coding standard violations and faultswithin and across software versions. In *MSR*. IEEE Computer Society, 41–50.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] Junjie Li. 2021. A Better Approach to Track the Evolution of Static Code Warnings. In *ICSE (Companion Volume)*. IEEE, 135–137.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. 2018. Lessons from building static analysis tools at Google. *Commun. ACM* 61, 4 (2018), 58–66.

[22] Jaime Spacco, David Hovemeyer, and William W. Pugh. 2006. Tracking defect warnings across versions. In *MSR*. ACM, 133–136.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] Xueqi Yang, Zhe Yu, Junjie Wang, and Tim Menzies. 2021. Understanding static code warnings: An incremental AI approach. *Expert Syst. Appl.* 167 (2021), 114134.

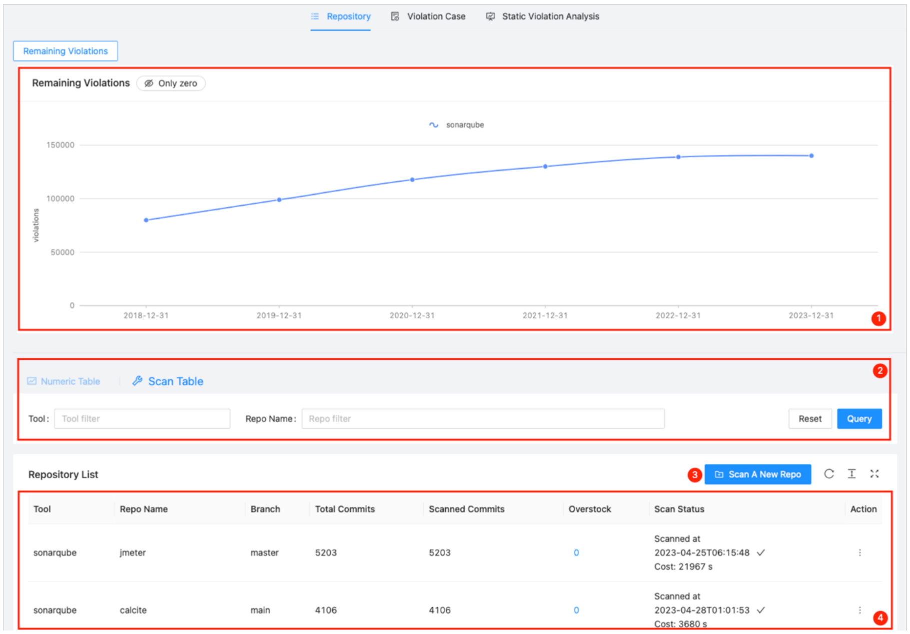
[28] Ping Yu. 2022. Towards Understanding Fixes of SonarQube Static Analysis Violations: A Large-Scale Empirical Study. In *SANER*. ACM, 698–709.

[29] Ping Yu. 2023. ViolationTracker: Building Precise Histories for Static Analysis Violations. In *ICSE*. ACM, 698–709.

[30] Ulas Yuksel and Hasan Sözer. 2013. Automated Classification of Static Code Analysis Alerts: A Case Study. In *ICSM*. IEEE Computer Society, 532–535.

[31] 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.

Appendix
ViolationBase Tool User Manual



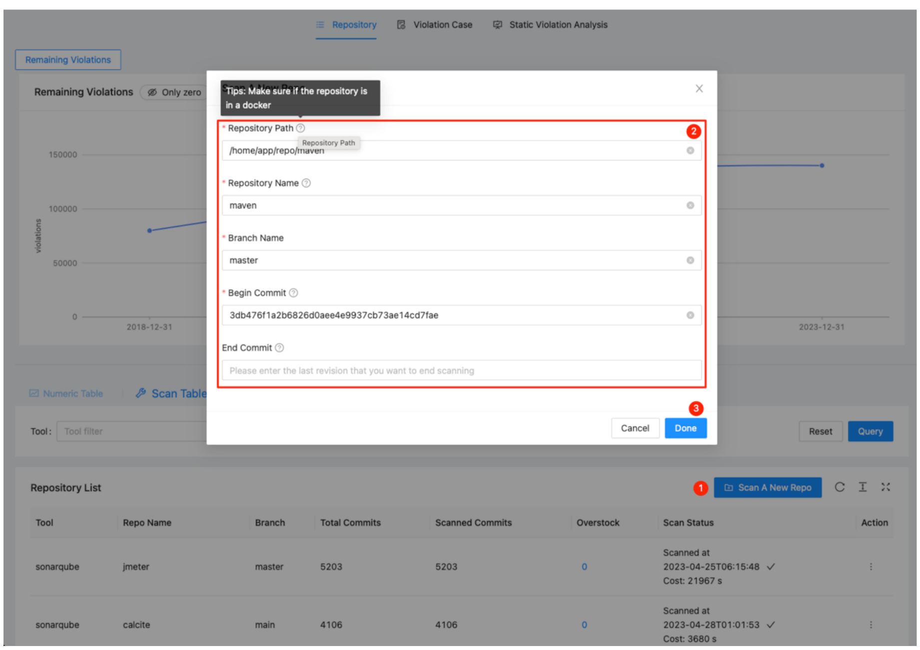
Repository Overview Page

Part 1: <u>**Project Quality Monitoring**</u>
Displays the trend chart of retained violations for all scanned projects, with the default view being an annual count.

Part 2: Search Box
Provides different search options based on various dimensions, such as the static scanning tool (e.g., SonarQube) and repository name.

Part 3: Scan A New Repository
Allows the addition of a new code repository for scanning.

Part 4: Repository Overview Table
Displays information about code repository scans, including scan status and the number of scanned revisions.

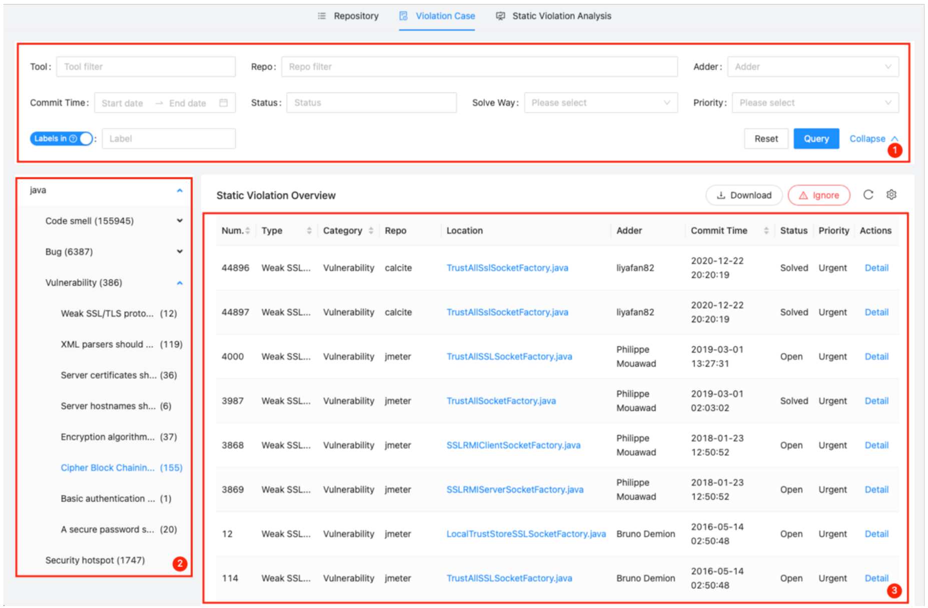


Scan A New Repository

Step 1: Click the "Add A New Repo" button.

Step 2: Enter the necessary information for the project repository to be scanned, including the repository's absolute path, repository name, branch name, and the first revision to be scanned. If using the ViolationBase tool inside a Docker container, ensure that the repository's absolute path corresponds to the path within the Docker container.

Step 3: Click the "Done" button to save the details.The ViolationBase tool will start scanning the violation history for the repository in the background.

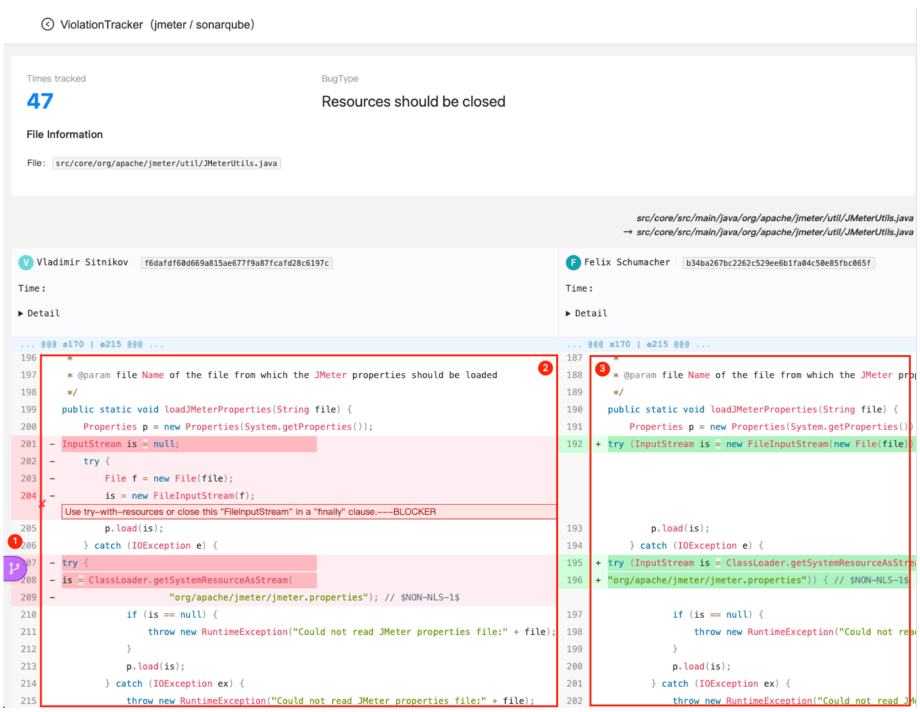


Violation Overview Page

Part 1: Search Box
Provides different search options based on various dimensions, such as searching by "Repository", "Status", and "Priority" to query information.

Part 2: Violation Category Overview
Displays the count of different violations based on their categories and types.

Part 3: Violation Overview Table
Shows all relevant information related to dimensions, such as violation types, locations, introducers, and more. Additional necessary information can be added for display in the settings based on requirements.



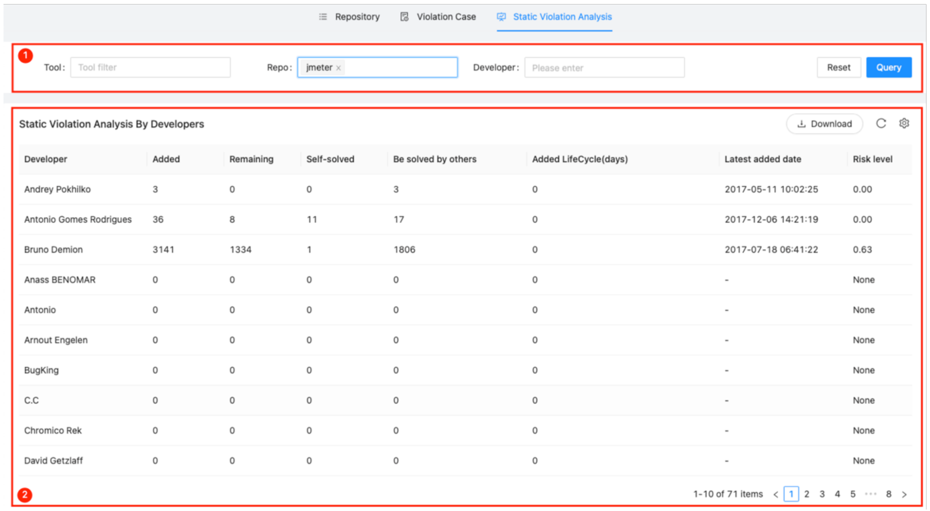
Violation Change History Backtracking

The top section displays basic information about the violation, including the violation type, cumulative trace count, and the file.

Part 1: Violation Tracker-Graph
Graphically presents the relationship between all the historical change nodes of the violation. Selecting any two nodes allows for a comparison of the violation's change data.

Part 2: Old Revision Violation Data
Displays the violation code information from the old revision.

Part 3: New Revision Violation Data
Displays the violation code information from the new revision.



Developer	Added	Remaining	Self-solved	Be solved by others	Added LifeCycle(days)	Latest added date	Risk level
Andrey Pokhilko	3	0	0	3	0	2017-05-11 10:02:25	0.00
Antonio Gomes Rodrigues	36	8	11	17	0	2017-12-06 14:21:19	0.00
Bruno Demion	3141	1334	1	1806	0	2017-07-18 06:41:22	0.63
Anass BENOMAR	0	0	0	0	0	-	None
Antonio	0	0	0	0	0	-	None
Arnout Engelen	0	0	0	0	0	-	None
BugKing	0	0	0	0	0	-	None
C.C	0	0	0	0	0	-	None
Chromico Rek	0	0	0	0	0	-	None
David Getzlaff	0	0	0	0	0	-	None

Developer Characteristics and Risk Analysis Page

Part 1: Search Box
Provides different search options based on various dimensions, such as searching by "Repository", "Developer" and "Tool" to query information.

Part 2: Developer Characteristics and Risk Analysis Table
Displays the historical data of violation introduction and fixes for each relevant repository developer and provides a risk rating. The higher the risk rating, the more likely it indicates potential issues with the code quality of that developer.