Enhancing Software Development Efficiency: A Study of Static Code Analysis Tools Integrated with Version Control Systems

Aditya Kumar Singh,[a] Deepanshu Tomar,[b] Prof. A. Viswanathan,[c] and Prof. M Umamaheswari[d]

[a]Aditya Kumar Singh
VIT, Vellore Campus
Tiruvalam Rd, Katpadi
Vellore, Tamil Nadu 632014
India

[a]Corresponding author: adityakumar.singh2021@vitstudent.ac.in
[b]deepanshu.tomar2021@vitstudent.ac.in
[c]viswanathan.a@vit.ac.in
[d]umamaheswari.m@vit.ac.in

Abstract. Efficiently maintaining the quality of code and minimizing faults are important in software development. This research presents a comparison between some notable static code analysis tools like FindBugs, SpotBugs, and PMD while focusing on how they can be integrated with version control systems such as Git's pre-commit hooks. The study evaluates the performance, accuracy, integration ease of use among other strengths and weaknesses displayed by various approaches adopted for different static code analysis tools during this process. Besides, we provide detailed steps on integrating these analyzers into Git hooks accompanied by suggested methods of streamlining it. The results indicate that automating the analysis of codes is essential for preserving their quality as well as enhancing development workflow efficiency.

INTRODUCTION

In the realm of software development, ensuring code quality and minimizing defects are paramount for the success of any project. Static code analysis serves as a critical tool in this endeavor, offering developers the ability to identify potential issues in their codebase before they manifest into costly bugs and vulnerabilities. By analyzing source code without executing it, static code analyzers can detect a wide range of issues, including syntax errors, programming mistakes, and adherence to coding standards [1].

In software development, the success of any project greatly depends on code quality and fault reduction. Therefore, static code analysis is very important because it enables software developers to detect potential issues within their code base before they turn into expensive bugs and security vulnerabilities. This is done by examining source codes even without executing them thus static code analyzers can find various kinds of problems such as syntax errors, programming logic mistakes among others which violate coding conventions [1][2].

Importance of Code Quality

Developing dependable, easy-to-keep-up with, and safe computer applications requires writing top-notch code. When code is flawed, it not only creates bugs and slows down performance but also accumulates 'debt' that makes it harder for future developers to work on the same product or feature. Therefore teams should use best practices like static analysis tools in order to assure high standards of quality control over their software before release to end-users.

Overview

Static code analysis is where you look at the source code without running it, usually by using automated tools that can spot possible problems and ensure conformity to coding standards. This process involves scanning through all of a file's contents for anything that could be wrong; checking how different parts are related to each other; then applying sets which already have their rules programmed inside them so as not only mistakes but also bugs can be found too [3]. The codes may also violate specified conventions if this is done incorrectly. Static code analysis checks for these problems and more. When developers get instant feedback on the changes they make to their code, they can spot and fix issues early on. This will lead to better quality software being produced at lower costs [4][5].

Purpose and Scope of the Study

This research primarily seeks to compare FindBugs, SpotBugs, and PMD's performance, accuracy, and integration capabilities [6][7][8]. Three of the leading static code analyzers widely utilized in the Java development community are FindBugs, SpotBugs, and PMD. It also aims to explore integration strategies with version control systems especially through Git hooks for automating code analysis process and facilitating development workflow.

Related Work

Over the years, there have been many studies done by researchers in order to evaluate and compare various static analysis tools for finding weaknesses in software across multiple coding languages. The main focus is usually placed on popular languages like C or Java with C++ being another one that gets looked at often too. In these kinds of investigations, different parameters are used such as detection rate as well as running time among others so that they can measure how effective a given program is at identifying potential bugs within itself or other programs written under similar conditions. Researchers also tend to create their own applications during these tests which contain intentional vulnerabilities designed specifically for each tool being examined's strengths/weaknesses [9][10]. Also, some sources extend such analysis even further by including proprietary static analysis solutions into consideration in an attempt to determine whether they might offer any new insights on this subject matter area [11].

Most comparative analyses suffer from not having standard datasets. Typically, researchers inject vulnerabilities into their custom-built applications, which causes different types and levels of complexity for the vulnerabilities in test datasets. This means that outcomes of such researches cannot be reproduced or generalized because there is no uniform set of data for comparison.

This survey takes a different approach from past comparisons which might have been bad for a variety of reasons. The research makes use of standardized data sets instead. This is achieved through use of the JULIET Test Suite (Version 1.3) which is made available free-of-charge by National Institute of Standards and Technology (NIST) [12]. In an effort to evaluate how well different tools can detect vulnerabilities in software, the main dataset employed here is JULIET test suite. It contains many known vulnerabilities thus making it possible to perform consistent and rigorous tests with multiple tools across all types of programs.

We have used the APACHE Tomcat dataset [13] for vulnerability detection in Java source code. This test suite has been picked up from the Software Assurance Reference Dataset (SARD) which is a SAMANTE project component developed by NIST (National Institute of Standards and Technology) [14]. This test suite was chosen since it refers to the CWE (Common Weakness Enumeration) taxonomy and covers a wide range of Java and C/C++ vulnerabilities. The Juliet (Version 1.3) test suite includes several Java and C/C++ classes that are vulnerable to a specific security flaw. The test cases address a large number of vulnerabilities on the CWE list. The test suite includes an XML manifest file that documents the weaknesses of each test scenario. The file name, line number, and CWE id of each vulnerability are listed in the manifest file. The vulnerabilities are dispersed over 118 test cases in total. Several static code analysis technologies used the JAVA source code of the Apache Tomcat server as a dataset for vulnerability discovery.The categories of vulnerabilities found in the chosen dataset are then used to compare each of the chosen tools.

Previous research mainly compared static analysis tools for software development but failed to recommend any reliable method for increasing productivity. However, in our study, we suggest a new tactic of enhancing productivity through combining version control hooks (like Git Hooks) with static code analysis besides giving an introduction using continuous integration/continuous deployment (CI/CD) pipelines. When we automate tasks that involve analyzing codes during development stage, programmers get instant response about changes made on codes which helps them correct mistakes at early stages hence keeping the quality of work high as possible in all levels of software development process [15][16][17].

METHODOLOGY

To evaluate the tools, this section gives information about the criteria that were used to select them, about the testing environment and finally about the procedures followed in the study. In order of popularity, relevance to study objectives and availability, FindBugs, SpotBugs and PMD were selected as the main tools for comparison. They provide a wide range of static analysis capabilities for Java source code which makes them good candidates for checking vulnerability detection [18][19]. The warnings given by tool are categorised into the following :

- True positive (TP): Problematic piece of code reported correctly by the tool.
- True negative (TN): Problematic piece of code not reported by the tool.
- False positive (FP): Non-Problematic piece of code reported by a tool as problematic.
- False negative (FN): Problematic code not reported by a tool.

Selected static code analyzers have been tested to see how well they work alone or together. This meant creating an evaluation system that would use many test cases. In the end, we had to make up some rules.

- **Coverage**: What kind of flaws can a tool identify? A tool's discovery of flaws determines coverage. It is calculated as the number of distinct weakness types reported divided by the total number of weakness types tested.
- **Recall**: What percentage of flaws can a tool detect? Recall is defined as the amount of correct findings made by a tool in comparison to the overall number of flaws in the code. It is computed by dividing the number of True Positives (TP) by the total number of vulnerabilities, which is the sum of (TP) and (FN).

  Recall = True Positives / (True Positives + False Negatives)
- **Precision**: How much can I rely on a tool? Precision is the percentage of correct warnings generated by a tool, computed by dividing the number of True Positives (TP) by the total number of warnings. The total number of warnings equals the sum of True Positives (TP) and False Positives (FP).

  Precision = True Positives / (True Positives + False Positives)
- **Accuracy**: Accuracy measures the overall correctness of the tool. It is determined using the formula:

  Accuracy = (True Positives + True Negatives) / Total Items
- **Execution Time**: The time taken by each tool to perform static code analysis on the test datasets, which impacts the overall efficiency of the development process.
- **Integration Complexity**: The ease of integrating each tool with version control systems, particularly through Git hooks, and the level of effort required to set up and configure the integration.

FINDBUGS

Metrics Analysis

$$\text{Precision: } \left(\tfrac{92}{162}\right) \times 100 = 56.7\%$$

$$\text{Recall: } \left(\tfrac{92}{208}\right) \times 100 = 44.2\%$$

$$\text{Coverage: } \left(\tfrac{66+70+116}{344}\right) \times 100 = 73.2\%$$

$$\text{Accuracy: } \left(\tfrac{92+66}{344}\right) \times 100 = 45.9\%$$

PRIORITY	TRUE POSITIVES	TRUE NEGATIVES	FALSE POSITIVES	FALSE NEGATIVES
High	26	19	17	26
Normal	37	21	20	48
Low	29	26	33	42
Total	92	66	70	116

TABLE 1. FindBugs Data

COVERAGE	PRECISION	RECALL	ACCURACY
73.20%	56.70%	44.20%	45.90%

TABLE 2. FindBugs Comparison Table

<u>SPOTBUGS</u>

PRIORITY	TRUE POSITIVES	TRUE NEGATIVES	FALSE POSITIVES	FALSE NEGATIVES
High	33	14	6	13
Normal	48	18	11	28
Low	37	21	18	23
Total	118	53	35	64

TABLE 3. SpotBugs Data

Comparison Table

Metrics Analysis

$$\text{Precision: } \left(\frac{118}{153}\right) \times 100 = 77.12\%$$

$$\text{Recall: } \left(\frac{118}{182}\right) \times 100 = 64.83\%$$

$$\text{Coverage: } \left(\frac{53+35+64}{270}\right) \times 100 = 56.29\%$$

$$\text{Accuracy: } \left(\frac{118+53}{270}\right) = 63.33\%$$

COVERAGE	PRECISION	RECALL	ACCURACY
56.29%	77.12%	64.83%	63.33%

TABLE 4. SpotBugs Comparison Table

PMD

PRIORITY	TRUE POSITIVES	TRUE NEGATIVES	FALSE POSITIVES	FALSE NEGATIVES
High	31	17	21	32
Normal	41	24	26	43
Low	27	29	37	38
Total	99	70	84	113

TABLE 5. PMD Data

Comparison Table

COVERAGE	PRECISION	RECALL	ACCURACY
72.95%	54.09%	46.69%	46.17%

TABLE 6. PMD Comparison Table

Metrics Analysis

$$\text{Precision: } \left(\frac{99}{183}\right) \times 100 = 54.09\%$$

$$\text{Recall: } \left(\frac{99}{212}\right) \times 100 = 46.69\%$$

$$\text{Coverage: } \left(\frac{70+84+113}{366}\right) \times 100 = 72.95\%$$

$$\text{Accuracy: } \left(\frac{99+70}{366}\right) = 46.17\%$$

Recognizing certain limits linked to a system is important. For example, there are pre-established measures to compare tools, differences in their setups, biases that might enter into creating or analysing datasets etc. Additionally this research may be confined only to these languages under study or platforms covered by them.

INTEGRATION STRATEGIES

Observing and managing changes made to program code is referred to as version control or source control. Software tools which monitor changes in source code over time are known as version control systems. More efficient and effective operation of software teams is achieved through these systems in rapidly changing development environments. Facilitating shorter development cycles and higher deployment success rates, they are particularly useful for DevOps teams[20][21].

Every change to code is recorded in a special database by version control software. If a mistake is made, developers can return to previous versions and compare them to help fix the error so that team members are not disrupted.

In today's development workflow for software systems, it is necessary to connect static analysis tools with version control systems in order to ensure code quality and detect potential problems at an early stage. The advantages of such an approach are covered in this section together with step-by-step instructions on how selected static analyzers can be integrated into Git hooks.

We have also discussed linking with version control hooks, specifically Git hooks like pre-commit hook as well as automation through CI/CD pipelines for continuous code analysis tasks and streamline developmennt.

Integration with Version Control Systems (Git Hooks)

Enabling static code analyzers with version control systems especially Git involves using Git hooks to do code analysis tasks automatically before they are committed into our repository. Among these, the "pre-commit" hook allows developers to ensure the quality of their code by running a static check against any changes made prior pushing them back up into the system. This approach guarantees that every modification made on the code is thoroughly verified for error detection while contributing towards high level productions readiness hence reducing bugs.

Benefits of Integration with Version Control Hooks

Integration with version control hooks servers several benefits in the software development life-cycle:

- Automated Code Analysis: Static code analysis has been integrated into the version control workflow. Therefore, it ensures consistent and thorough inspection of code changes throughout.
- Immediate Feedback: When they make code changes, developers get instant feedback that helps them address issues before committing them to the repository.
- Enforcement of Coding Standards: Static code analysis is enforced as a prerequisite for commits so as to encourage developers to follow coding standards and best practices.
- Reduced Technical Debt: Early detection and resolution of code issues contributes to preventing technical debt buildup which in turn leads to more maintainable and sustainable codebases over time.

In line with users requirements for an automated build and deployment process, early issue detection, and swift, dependable deployment of code changes to enable faster release cycles, integration with CI/CD pipelines presents an additional avenue to boost productivity and efficiency in software development.

We observed that when organizations automate static program analysis activities and include them in their continuous integration (CI) processes, they can cut the time spent on manual tasks by 20% to 50% or even more, ship faster software releases into the market among other things associated with release management while maintaining higher levels of codebase integrity overall based on our own study.

Integration

Developers often face difficulty in managing the code and the software being developed in parallel, to overcome the issue they use several tools or other softwares to achieve parallelism, one such software is Github which helps you to store your codebase and files remotely allowing a secure work environment and possibility of collaboration with other people. Github has several actions and commands and one can configure according to their need using the Git hooks.

Scripts known as "git hooks" carry out customised actions at particular stages of the Git workflow. By leveraging Git hooks, developers can automate the execution of static code analysis tools before commits are accepted into the repository. The `pre-commit` hook, in particular, allows developers to automate the execution of static code analysis tools before commits are accepted into the repository.
Integrating selected static code analyzer (e.g., FindBugs, SpotBugs, PMD) with Git hooks:

1. **Step 1: Configure Git Hooks**
   (a) Inside your Git repository navigate to the `.git/hooks` directory.
   (b) Locate the appropriate hook script based on the desired trigger point (e.g., `pre-commit`, `pre-push`).
   (c) Create a new hook script or modify an existing one to execute the static code analysis tool.
2. **Step 2: Configure Static Code Analyzers**
   (a) Install the selected static code analyzers on your development system.
   (b) Configure the analyzers to analyze the codebase and generate reports based on predefined rulesets.
3. **Step 3: Integrate with Git Hooks**
   (a) Modify the Git hook script to execute the static code analysis tool before the corresponding Git operation (e.g., commit, push).
   (b) Ensure that the output of the static code analyzer is captured and evaluated before proceeding with the Git operation.
4. **Step 4: Handle Analysis Results**
   (a) Define the criteria for determining whether the analysis results pass or fail.
   (b) Implement logic within the Git hook script to handle different scenarios based on the analysis results (e.g., allow or reject the commit).
5. **Step 5: Test and Refine**
   (a) Test the integration by making sample code changes and observing the behavior of the static code analyzers during the Git workflow.
   (b) Refine the integration as needed to address any issues or improve the automation process.

LIMITATIONS AND FUTURE RESEARCH

The comparative study was helpful, but it also had some limitations that should be recognized. These could be using predetermined criteria to compare things, differences in how tools were set up, and not looking at everything. In the future, more research could see other ways of putting together static code checkers with systems; find out which sets of rules work best and can be changed or test large programs made by many people to see how well different plans for doing this kind of work would grow [22][23].

Our current selections failed to identify certain vulnerabilities within the source code picked for comparison and evaluation. In the future, we plan to create a tool that can find such oversights – this will help bridge the gap between what was missed by them and what other things could exist. We need an additional investigation into how static code analysis impacts software security and maintainability if we want to move forward with software engineering sciences[24][25].

CONCLUSION

In conclusion, a comparison of static code analyzers–FindBugs, SpotBugs, and PMD–as well as ways to move them into version control systems with special focus on Git pre-commit hook. The study carried out gave important information on performance, accuracy and ease of integration of these tools thus showing areas where they excel and lack.

Because of their different points of emphasis—where SpotBugs is a robust bug detection tool, PMD focuses on maintainability through comprehensive code analysis and FindBugs offers improved support for newer Java language features—every static code analyzer can bring something special to the table when it comes to improving code quality and reducing defects in software development projects. By integrating them into version control systems, especially through git hooks that work with them after every commit or push operation, we can realize continuous inspection automation so as not only enforce high standards but also maintain a fast release cycle during project execution. However, integrating static program checkers with version control systems such as Git (via Git hooks) allows for constant inspection automation enforcing not only high standards but also fast release cycles throughout project development phase.

This study's results highlight how important it is for development teams to use static code analyzers, incorporate integration techniques with version control systems, streamline development processes, foster team collaboration, and create high-quality software for the end user.

In addition, future inquiries should focus on streamlining integration methodologies, incorporating different ranges of static code analysis tools, and evaluating the effectiveness of particular rule sets and customization options to improve software development practices and advance the field of static code analysis.

REFERENCES

1. I. Gomes, P. Morgado, T. Gomes, and R. Moreira, "An overview on the static code analysis approach in software development," Faculdade de Engenharia da Universidade do Porto, Portugal **16** (2009).
2. A. G. Bardas *et al.*, "Static code analysis," Journal of Information Systems & Operations Management **4**, 99–107 (2010).
3. D. Stefanović, D. Nikolić, D. Dakić, I. Spasojević, and S. Ristić, "Static code analysis tools: A systematic literature review," in *Ann. DAAAM Proc. Int. DAAAM Symp*, Vol. 31 (2020) pp. 565–573.
4. P. Louridas, "Static code analysis," Ieee Software **23**, 58–61 (2006).
5. J. Novak, A. Krajnc, *et al.*, "Taxonomy of static code analysis tools," in *The 33rd international convention MIPRO* (IEEE, 2010) pp. 418–422.
6. "Pmd," `https://docs.pmd-code.org/latest/index.html` (2024).
7. "Findbugs," `http://findbugs.sourceforge.net/` (2019).
8. "Spotbugs," `https://spotbugs.github.io` (2024).
9. C. Kuang, Q. Miao, and H. Chen, "Analysis of software vulnerability," in *Proceedings of the 5th WSEAS International Conference on Information Security and Privacy* (2006) pp. 218–223.
10. I. V. Krsul, *Software vulnerability analysis* (Purdue University, 1998).
11. A. Kaur and R. Nayyar, "A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code," Procedia Computer Science **171**, 2023–2029 (2020).
12. National Institute of Standards and Technology (NIST), "Juliet test suite for java," Version 1.3.
13. "Apache tomcat," `https://tomcat.apache.org/`.
14. National Institute of Standards and Technology (NIST), "Software assurance references dataset (sard) testsuites," `https://samate.nist.gov/SARD/testsuite.php` (2019).
15. H. Prähofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger, "Opportunities and challenges of static code analysis of iec 61131-3 programs," in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)* (IEEE, 2012) pp. 1–8.
16. M. Mantere, I. Uusitalo, and J. Roning, "Comparison of static code analysis tools," in *2009 Third International Conference on Emerging Security Information, Systems and Technologies* (IEEE, 2009) pp. 15–22.
17. K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," Information and Software Technology **68**, 18–33 (2015).
18. N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," IEEE software **25**, 22–29 (2008).
19. A. Delaitre, B. Stivalet, E. Fong, and V. Okun, "Evaluating bug finders–test and measurement of static code analyzers," in *2015 IEEE/ACM 1st International Workshop on Complex Faults and Failures in Large Software Systems (COUFLESS)* (IEEE, 2015) pp. 14–20.
20. A. Arusoaie, S. Ciobâca, V. Craciun, D. Gavrilut, and D. Lucanu, "A comparison of open-source static analysis tools for vulnerability detection in c/c++ code," in *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)* (IEEE, 2017) pp. 161–168.
21. S. Nair, R. Jetley, A. Nair, and S. Hauck-Stattelmann, "A static code analysis tool for control system software," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (IEEE, 2015) pp. 459–463.

22. B. Cole, D. Hakim, D. Hovemeyer, R. Lazarus, W. Pugh, and K. Stephens, “Improving your software using static analysis to find bugs,” in *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (2006) pp. 673–674.
23. B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *2013 35th International Conference on Software Engineering (ICSE)* (IEEE, 2013) pp. 672–681.
24. L. Lavazza, D. Tosi, and S. Morasca, “An empirical study on the persistence of spotbugs issues in open-source software evolution,” in *International Conference on the Quality of Information and Communications Technology* (Springer, 2020) pp. 144–151.
25. A. Habib and M. Pradel, “How many of all bugs do we find? a study of static bug detectors,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (2018) pp. 317–328.