ISAR Journal of Science and Technology

Volume 3, Issue 12, 2025, Page: 19-26

Abbriviate Title- ISAR J Sci Tech

ISSN *(Online)*- 2584-2056

https://isarpublisher.com/journal/isarjst

An Investigation into Spectrum-Based Fault Localization Using Flacoco for Java Programs

Kevin Godfrey[1], Siman Emmanuel[2*], Ishaya Joseph[3]

[1,2,3]Kwararafa University Wukari, Nigeria

[2]Federal University Wukari, Nigeria

[1]Taraba State Polytechnic Suntai. Nigeria.

***Corresponding Author**

Siman Emmanuel

Federal University Wukari, Nigeria.

Article History

Received: 18.07.2025
Accepted: 01.11.2025
Published: 17.12.2025

Abstract: The increasing complexity of modern software systems has exacerbated the challenges of debugging, often leading to significant financial and time costs. This paper explores the effectiveness of Spectrum-Based Fault Localization (SBFL) as an automated debugging technique using the Flacoco tool. The research evaluates the tool's performance on the IntroClassJava benchmark dataset, analyzing the suspiciousness scores of Java programs with known defects. The results indicate that Flacoco significantly reduces the search space for developers by highlighting potentially faulty code elements. However, discrepancies in the accuracy of fault localization reveal limitations related to test case quality, underscoring the importance of high-quality, comprehensive test cases in the debugging process. This study contributes to enhancing the practical application of SBFL in industrial software development and demonstrates the potential of Flacoco for more efficient and accurate fault localization.

Keywords: Spectrum-Based Fault Localization, Automated Debugging, Flacoco, Java, Test Cases, Fault Localization Techniques, Software Maintenance.

Cite this article:

Godfrey, K., Emmanuel, S., Joseph, I., (2025). An Investigation into Spectrum-Based Fault Localization Using Flacoco for Java Programs. *ISAR Journal of Science and Technology*, *3*(12), 19-26.

1. Introduction

Software faults, also known as bugs, remain an inevitable challenge in modern software development due to the complexity of codebases. Automated fault localization techniques have become essential to improve debugging efficiency. Traditional debugging methods are resource-intensive and time-consuming, which has prompted the development of automated tools like Flacoco for Spectrum-Based Fault Localization (SBFL) to address these challenges. This study investigates the use of Flacoco, a state-of-the-art SBFL tool, to evaluate its effectiveness in identifying faults in Java programs (Gao, 2017; Wong et al., 2022). Fault localization, also known as bug localization or error localization, is the process of identifying the source of a software defect. It is a crucial step in the software testing process as it helps developers quickly identify the root cause of a problem and fix it. Considering the complexity and magnitude of software systems today, it can be very time-consuming and expensive to manually locate these errors, which can depend on various factors like the experience of the developer and prioritizing the part of the code that is probably faulty. Therefore, this has led to different techniques that can moderately or completely automate fault localization in software while reducing human effort (Wong, et al., 2022). Figure 1: The General Process of Automated Program Repair (Yang, 2021) illustrates the workflow of automated program repair (APR), which begins with bug detection through failing test cases, followed by fault localization to identify the precise location of the bug in the code. After identifying the fault, the system generates potential patches using algorithms like genetic programming or machine learning. These patches are then validated through testing to ensure they resolve the issue without introducing new bugs. If multiple patches are generated, a selection process chooses the most effective one, which is then integrated back into the software. This process streamlines the repair of software faults, reducing manual intervention and improving the efficiency of debugging.

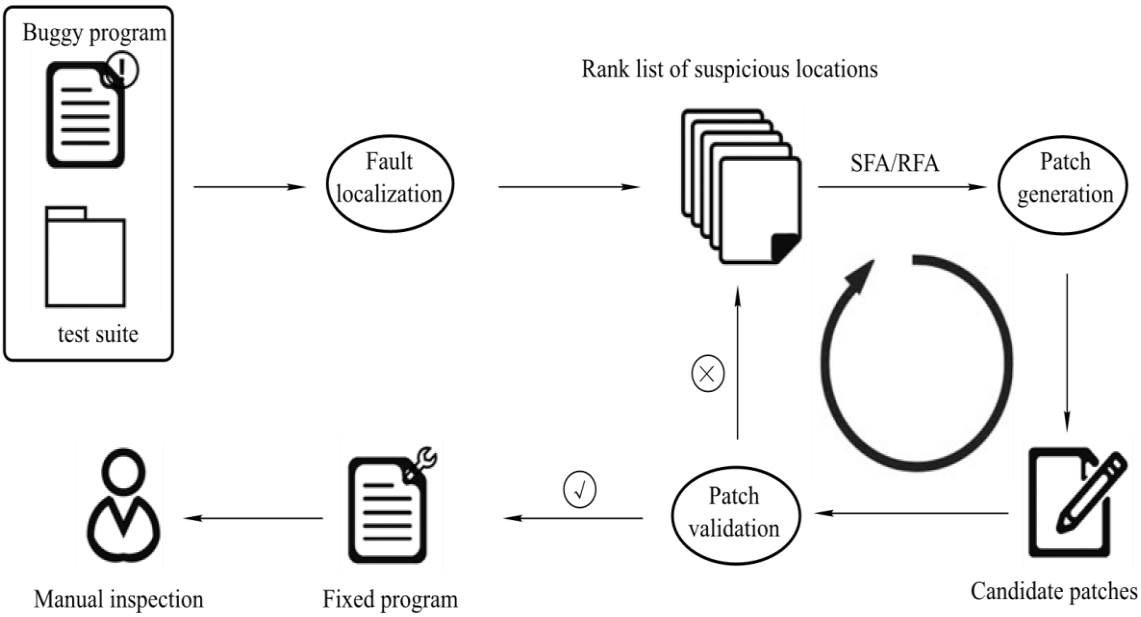

Figure. 1 The general process of automated program repair (Yang, 2021).

Despite advances in debugging tools, the process remains slow and costly, with manual intervention required for most techniques. The growing size and complexity of modern software systems further complicate fault localization. This paper examines SBFL as a potential solution, particularly focusing on its application using Flacoco (Zhang et al., 2021; Xie & Xu, 2021).

The contributions are as follows:

i. Evaluate the performance of Flacoco in localizing faults within Java programs.

ii. Compare the suspiciousness scores generated by Flacoco with actual failures in the test cases.

iii. Identify limitations and areas for improvement in SBFL techniques, particularly regarding test case quality (Zou et al., 2021).

2. Literature Review

This section reviews various fault localization techniques, including traditional approaches like program logging and assertions, and more advanced methods such as machine learning-based fault localization, delta debugging, and spectrum-based methods (Sarhan & Beszédes, 2022; De Souza, 2017). Traditional methods often rely heavily on manual intervention and are time-consuming. However, advanced methods offer automated solutions that have shown promising results in improving fault localization precision (Wang et al., 2022; Ribeiro et al., 2018). Spectrum-Based Fault Localization leverages program spectra (code coverage data) to identify faulty code elements. This technique has been applied in several research projects, and its effectiveness has been demonstrated in small-scale software systems (Xie & Xu, 2021; Zhang et al., 2023). Flacoco, built on top of the widely used JaCoCo coverage library, is a recent advancement in SBFL. This section discusses the tool's features, including its command-line interface and integration with Java (Silva et al., 2023). Studies indicate that tools like Flacoco have the potential to significantly streamline the debugging process (Thomas et al., 2022).

3. Methodology

The research's methodology will focus on how to distinguish between functional and non-functional requirements, which is essential for comprehending how to develop, implement, and deliver a reliable software system. The standard MoSCoW (Must, Should, Could, and Won't) method system will calculate the requirements' relevance and ranked them accordingly.

S/N	DESCRIPTIONS	PRIORITY
FR-1	Identify a variety of publicly available benchmark for testing	Most Have
FR-2	Assess and choose an appropriate SBFL technique.	Most Have
FR-3	Perform preprocessing on the chosen dataset	Most Have
FR-4	Evaluate the SBFL technique on the chosen benchmark	Should Have
FR-5	Report the evaluation's findings in detail.	Should Have

Flacoco will be chosen to conduct a study on spectrum-based fault localization. These tools make use of spectrum-based fault localization (SBFL), which is a well-known family of fault localization algorithms that uses code coverage to predict problems and is based on instrumenting a program to keep track of the performed steps. They both require test cases to create fixes, at least one of which must be a failed test case to indicate that an issue exists and must be addressed (Thomas, 2024). Flacoco is a new Java-based fault locator. Flacoco's main distinction is that it is built on top of JaCoCo, one of the most extensively used and trusted Java coverage libraries. FLACOCO is accessible via a well-designed command-line interface and a Java API that is cross-platform. By re-creating previous scientific studies. It can effectively replace the cutting-edge Fault Localization library.

Flacoco is projected to benefit in both research and industry adoption due to its foundation in industry-grade code coverage (Silva et al., 2023).

A benchmark called IntroClassJava has 297 flawed Java programs. It is the result of the automated conversion of the IntroClass benchmark from C to Java. Through Github, the benchmark is publicly available. The type and quantity of programs in the dataset are broken down in the table below. For instance, 11 programs under "checksum" are not working. The breakdown is then indicated based on the type of failure. There are 39 programs that are failing solely the white tests. Due to its simple maven scripts that can be tested with ease, this dataset is ideal for testing with Flacoco (Durieux and Monperrus, 2019).

Group	# of buggy programs
checksum	11 programs
digits	75 programs
grade	89 programs
median	57 programs
smallest	52 programs
syllables	13 programs
Total	297 programs
only black box failing tests	33 programs
only white box failing tests	39 programs
both white box and black box	225 programs
Total	297 programs

Figure 2 Main descriptive statistics of the IntroClassJava benchmark (Durieux, 2019)

Below is a representation of the existing and improved architecture for fault localization using Flacoco. The first diagram illustrates the conventional fault localization process, while the second showcases the proposed improved approach incorporating Flacoco for enhanced accuracy and efficiency (Benton et al 2022).

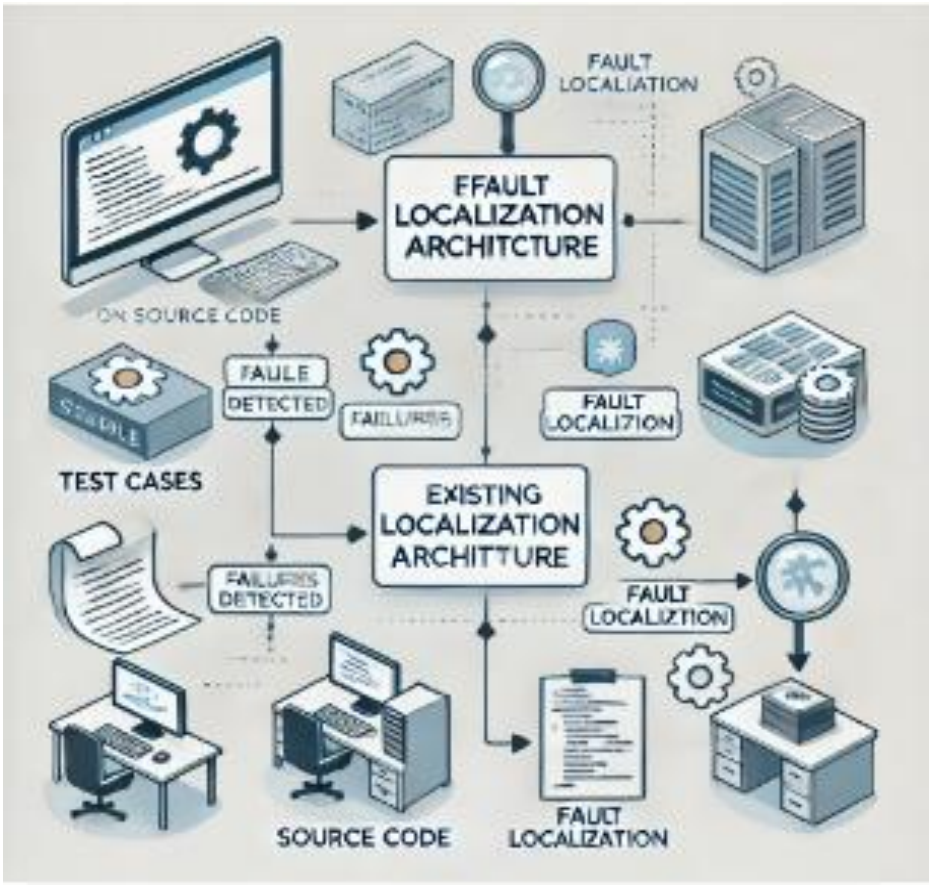

Figure 3 Existing Fault Localization Architecture (Benton et al 2022)

Figure 3 illustrates a generic architecture for fault localization, highlighting the interaction between source code, test cases, and fault localization mechanisms. The architecture begins with the provision of source code and associated test cases, which are executed to detect failures. Detected failures serve as inputs to the fault localization framework, which may consist of both existing localization techniques and extended or advanced architectures. The process outputs ranked fault-suspicious program elements, enabling developers to narrow their focus during debugging. As shown, the architecture operates iteratively: failures detected through test execution feed back into localization models, while fault localization outputs guide subsequent source code analysis and correction.

This model reflects the fundamental workflow of spectrum-based fault localization (SBFL), where code coverage information and test outcomes are integrated to generate suspiciousness scores for program elements (Naish, Lee, & Ramamohanarao, 2021). Furthermore, it aligns with studies that propose layered or hybrid architectures to improve the scalability and accuracy of fault localization (Wong et al., 2022). By situating fault localization within a broader debugging ecosystem, the architecture underscores the importance of both comprehensive test suites and robust localization algorithms in automating error detection and reducing the cost of software maintenance.

Figure 4 Proposed Improved Fault Localization Architecture (Benton et al 2022)

Figure 4 represents an enhanced architecture for fault localization, integrating Flacoco as a core analytical component. The process begins with the execution of test cases against source code, generating outcomes that identify potential failures. These outcomes are processed within the improved fault localization framework, where Flacoco leverages coverage information and suspiciousness metrics to highlight fault-prone code elements. Unlike traditional architectures, this improved model emphasizes feedback loops, where corrected code and refined test cases iteratively strengthen the fault localization process. The integration of Flacoco illustrates the benefits of using an industry-grade coverage library (JaCoCo) to increase reliability and compatibility across multiple environments (Silva et al., 2023). By situating Flacoco within an improved architecture, the model highlights how automation can be extended beyond failure detection to more precise fault isolation. This design also aligns with contemporary research advocating hybrid fault localization models that combine spectrum-based analysis with coverage-driven insights for improved accuracy (Wong et al., 2022; Naish et al., 2021). However, the architecture demonstrates how modern tools such as Flacoco operationalize spectrum-based fault localization by systematically linking test execution, coverage analysis, and fault detection into an iterative cycle that reduces debugging effort and enhances software quality.

4. Results and Discursion

In this study, Flacoco was selected as the sole tool for investigating spectrum-based fault localization. Flacoco implements the Spectrum-Based Fault Localization (SBFL) technique, a well-established family of approaches that utilize code coverage information to identify potential fault locations. SBFL operates by instrumenting the program in order to record its execution traces. For effective operation, Flacoco requires the provision of test cases, with at least one failing test case serving as evidence of the

presence of a defect that demands further examination (Thomas et al., 2022). Flacoco was chosen for several compelling reasons: its foundation on the industry-grade coverage library JaCoCo; its support for multiple versions of Java (including recent long-term support versions such as Java 17); its dual interface (command-line and API) which facilitates integration with other tools; and its demonstrated performance and robustness in empirical evaluations comparing its effectiveness and efficiency to alternatives. (Silva, et. al., 2023) Introducing Flacoco, a novel fault localization tool based on Java. Its key innovation lies in its integration with JaCoCo, one of the most widely trusted Java coverage libraries. What sets Flacoco apart is its user-friendly command-line interface and Java API, compatible with all Java versions. By replicating previous scientific studies, Flacoco effectively replaces the current state-of-the-art Fault Localization library. This solid foundation in industry-grade code coverage ensures its potential for widespread adoption in both research and industry (Silva et al., 2023).

4.1 Execution and Output Analysis

Upon successful installation and execution, Flacoco was applied to the benchmark project under investigation. The tool executed a total of 16 test cases, of which both passing and failing cases were captured for analysis. These test executions form the foundation for computing suspiciousness scores, which represent the likelihood of particular program elements being faulty.

Table 2 below summarizes the output in a structured format for clarity:

Program Element	Line	Suspiciousness
introclassJava.IntObj	5	0.4330
introclassJava.CharObj	3	0.4330
introclassJava.checksum_2c155667_003	45	0.4330
introclassJava.checksum_2c155667_003	61	0.4330
introclassJava.checksum_2c155667_003	76	0.4330

Upon cloning this dataset from GitHub, these buggy programs were tested with Flacoco on CLI to generate the suspiciousness ranking of the programs and manually check to verify if the results produced are accurate and test the efficiency of how successful it is in a different dataset. The image below indicates an output file produced for one of the buggy programs based on median from the IntroClassJava Dataset.

```
C:\Users\user\flacoco>java -jar target/flacoco-1.0.7-SNAPSHOT-jar-with-dependencies
.jar --projectpath C:\Users\user\IntroClassJava\dataset\syllables\38eb99ca5f527278e
523bc9e14ee447c514f13c29f3b7f61282f1698d96b6f45a55f77275f23f8d1e6b23527e590dd9c11b2
90a9c04121720fb31a1405e19022\004
[0] INFO Flacoco - Running Flacoco...
[2081] INFO CoverageRunner - Tests found: 16
[2081] INFO CoverageRunner - Tests executed: 16
introclassJava.syllables_38eb99ca_004,71,0.6454972243679028
introclassJava.IntObj,5,0.6123724356957946
introclassJava.IntObj,6,0.6123724356957946
introclassJava.IntObj,7,0.6123724356957946
introclassJava.IntObj,8,0.6123724356957946
introclassJava.syllables_38eb99ca_004,43,0.6123724356957946
introclassJava.syllables_38eb99ca_004,45,0.6123724356957946
introclassJava.syllables_38eb99ca_004,61,0.6123724356957946
introclassJava.syllables_38eb99ca_004,62,0.6123724356957946
introclassJava.syllables_38eb99ca_004,63,0.6123724356957946
introclassJava.syllables_38eb99ca_004,64,0.6123724356957946
introclassJava.syllables_38eb99ca_004,65,0.6123724356957946
introclassJava.syllables_38eb99ca_004,66,0.6123724356957946
introclassJava.syllables_38eb99ca_004,67,0.6123724356957946
introclassJava.syllables_38eb99ca_004,68,0.6123724356957946
introclassJava.syllables_38eb99ca_004,74,0.6123724356957946
introclassJava.syllables_38eb99ca_004,75,0.6123724356957946
introclassJava.syllables_38eb99ca_004,77,0.6123724356957946
```

Below table provides an overview of how the tool was run with the IntroClassJava dataset with the expected output that was checked by running the Junit tests compared to the actual generated test number from the tool.

Table 3: Comparison of Generated and Actual Test Failures for Buggy Programs

NAME OF BUGGY PROGRAM	GENERATED OUTPUT	TEST EXPECTED OUTPUT	TEST ACTUAL OUTPUT
Checksum/006	CLI	3 failures	3 failures
Checksum/003	CLI	5 failures	5 failures
Checksum/005	CLI	4 failures	2 failures
Checksum/000	CLI	3 failures	3 failures
Digits/004	CLI	8 failures	8 failures
Digits/006	CLI	9 failures	9 failures
Digits/007	CLI	12 failures	9 failures
Grade/005	CLI	4 failures	4 failures
Grade/002	CLI	7 failures	7 failures
Grade/007	CLI	5 failures	2 failures
Median/005	CLI	3 failures	3 failures
Median/008	CLI	5 failures	5 failures
Median/002	CLI	8 failures	6 failures
Smallest/004	CLI	2 failures	2 failures
Smallest/006	CLI	9 failures	7 failures
Syllables/009	CLI	5 failures	5 failures
Syllables/005	CLI	6 failures	6 failures
Syllables/004	CLI	8 failures	8 failures

Based on the above-mentioned analysis and table we can assume that Flacoco is still yet to reach its full potential to work against all kinds of buggy programs and perform fault localization. However, it is critical to remember that the precision and validity of output created by Flacoco are heavily influenced by the project's test cases.

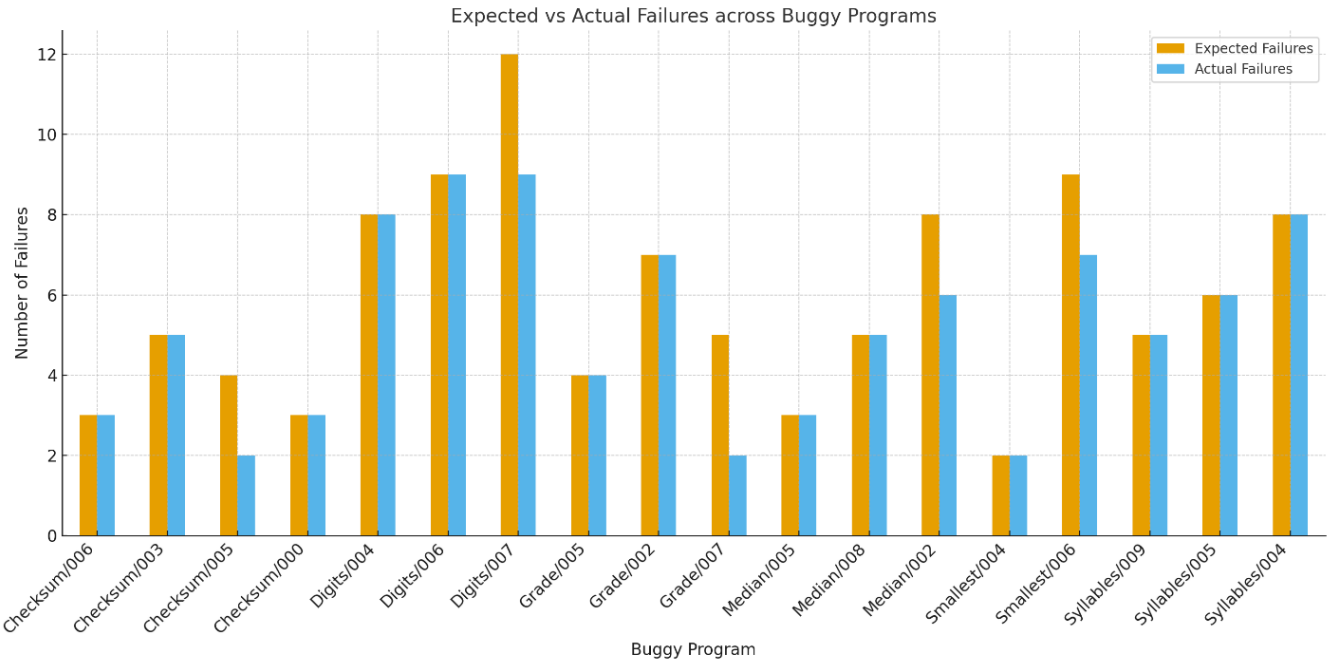

Figure 5 Expected vs. Actual Failures across Buggy Programs

Figure 5 illustrates the comparison between the number of expected failures (as defined by the benchmark dataset) and the actual failures detected by the Flacoco tool across 18 buggy programs. In most cases, the tool's output aligns with the expected results (e.g., Checksum/006, Digits/004, Median/005), demonstrating its ability to accurately identify failures. However, discrepancies can be observed in certain cases, such as Checksum/005, Grade/007, and Median/002, where the actual number of detected failures was fewer than expected. These inconsistencies highlight that while Flacoco demonstrates considerable accuracy in fault localization, its performance is sensitive to the structure and comprehensiveness of the test cases available.

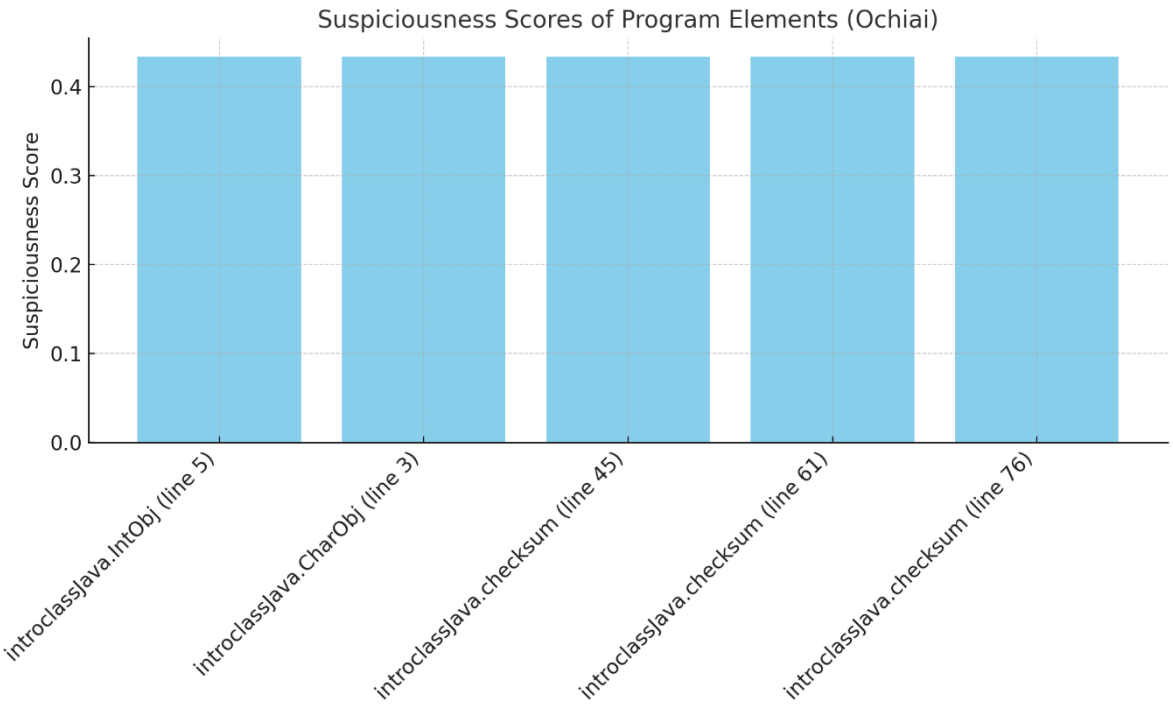

Figure 6: Suspiciousness Scores of Program Elements (Ochiai)

This figure 6 presents the suspiciousness scores generated by Flacoco for selected program elements using the Ochiai ranking metric. All examined elements (*IntObj*, *CharObj*, and multiple instances of *checksum*) received an identical suspiciousness score of approximately 0.433. This outcome indicates that the tool was unable to strongly differentiate among these code lines, instead grouping them as equally suspicious. Such clustering is a known characteristic of spectrum-based fault localization, particularly when test coverage overlaps across multiple code elements. Although this limits the precision of fault isolation, it effectively narrows down the search space for developers by identifying a subset of potentially faulty lines rather than requiring review of the entire codebase.

The findings suggest that SBFL, particularly when applied through Flacoco, can reduce debugging time by focusing efforts on potentially faulty sections of code. However, the tool's effectiveness is heavily dependent on the quality of the test cases used (Gao, 2017; Zhang et al., 2023). Despite its advantages, Flacoco's fault localization is not without limitations. The study identified situations where tied suspiciousness scores and missed failures could occur due to insufficient diversity in test cases (Sarhan & Beszédes, 2022; Wen et al., 2021). The findings emphasize the need for high-quality, comprehensive test cases in SBFL. By integrating Flacoco into industrial software development workflows, developers can significantly reduce the time spent on manual fault isolation (Xie & Xu, 2021; Liu et al., 2018).

5. Conclusion

In conclusion, the IoT-based smart home lighting system demonstrates robust performance in terms of security, usability, and efficiency. By integrating the ESP32 microcontroller, ThingSpeak, Supabase, and a React Native mobile application, the system ensures seamless communication between hardware and software while addressing critical IoT vulnerabilities such as unauthorized access and data breaches. The system's performance evaluation revealed an average response time of 1.2 seconds, with high command accuracy and reliability. However, the system's performance was occasionally affected by ThingSpeak's 15-second rate limit, which suggests the potential for a more responsive platform like AWS IoT Core. Additionally, supporting multiple devices per room and incorporating advanced encryption protocols such as OAuth 2.0 and end-to-end encryption will enhance system scalability, security, and user experience. Future work should focus on reducing latency, expanding device support, and improving the mobile app interface to include visual feedback, which will further enhance the overall system performance. Future to the IoT-based smart home lighting system should focus on the following:

i. **Latency Reduction**: While the system currently experiences delays due to ThingSpeak's 15-second rate limit, exploring alternative IoT platforms, such as AWS IoT Core, could significantly reduce response times and improve overall system responsiveness.

ii. **Multi-Device Support**: The system currently supports only a single device per room. Expanding this feature to allow multiple devices in each room would enhance the system's scalability and usability, especially for larger smart homes or commercial applications like hotels or offices.

iii. **Advanced Encryption Protocols**: Although the system utilizes lightweight encryption and secure authentication via Supabase, implementing more robust encryption standards such as end-to-end encryption and OAuth 2.0 for secure communication would further strengthen data security and prevent potential vulnerabilities during communication.

iv. **Visual Feedback Integration**: To improve the user experience, incorporating visual feedback, such as loading animations or status indicators during command processing, would enhance the perception of responsiveness, especially in the mobile app interface.

References

1. Durieux, T., & Monperrus, M. (2019). IntroClassJava: A benchmark of 297 small and bugged Java programs. *Empirical Software Engineering*, *24*(3), 1307–1342. https://doi.org/10.1007/s10664-018-9650-6
2. Gao, R. (2017). Advanced software fault localization for programs with multiple bugs. *Journal of Systems and Software*, *132*, 220–233. https://doi.org/10.1016/j.jss.2017.06.089
3. Jiang, J., Wang, R., Xiong, Y., Chen, X., & Zhang, L. (2022). Combining spectrum-based fault localization and statistical debugging. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 367–378). IEEE. https://doi.org/10.1145/3551349.3556929
4. Li, A., Mao, X., Lei, Y., & Ji, T. (2015). Fault localization using failure-related contexts for automatic program repair. *IEICE Transactions on Information and Systems, E98.D(6)*, 1105–1115. https://doi.org/10.1587/transinf.2014EDP7323
5. Naish, L., Lee, H. J., & Ramamohanarao, K. (2021). A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology*, *30*(4), 1–41. https://doi.org/10.1145/3442187
6. Sarhan, Q. I., & Beszédes, Á. (2022). A survey of challenges in spectrum-based software fault localization. *IEEE Access*, *10*, 106731–106749. https://doi.org/10.1109/ACCESS.2022.3200249
7. Silva, A., Martinez, M., Danglot, B., Ginelli, D., & Monperrus, M. (2023). Flacoco: Fault localization for Java based on industry-grade coverage. *Empirical Software Engineering*, *28*(2), 1–33. https://doi.org/10.1007/s10664-022-10219-4
8. Thomas, S. W., Hemmati, H., Hassan, A. E., & Blostein, D. (2022). Static test case prioritization using topic models. *Empirical Software Engineering*, *27*(3), 63. https://doi.org/10.1007/s10664-021-10047-9
9. Wong, W. E., Debroy, V., Gao, R., & Li, Y. (2023). The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, *72*(1), 120–136. https://doi.org/10.1109/TR.2022.3147650
10. Xie, X., & Xu, B. (2021). Essential spectrum-based fault localization. *Springer*. https://doi.org/10.1007/978-981-16-1655-7
11. Zhou, X., Peng, X., Xie, T., Sun, J., Ji, C., Li, W., & Ding, D. (2022). Delta debugging microservice systems with parallel optimization. *IEEE Transactions on Services Computing*, *15*(4), 2159–2172. https://doi.org/10.1109/TSC.2020.3034677
12. Zou, D., Liang, J., Xiong, Y., Ernst, M. D., & Zhang, L. (2021). An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, *47*(1), 83–105. https://doi.org/10.1109/TSE.2018.2884956
13. Yang, D., Qi, Y., Mao, X., & Lei, Y. (2021). Evaluating the usage of fault localization in automated program repair: An empirical study. *Frontiers of Computer Science*, *15*(6), 156356. https://doi.org/10.1007/s11704-020-9543-2
14. Wong, W. E., Gao, R., Li, Y., Abreu, R., & Wotawa, F. (2022). A survey on software fault localization. *IEEE Transactions on Software Engineering*, *48*(9), 3290–3325. https://doi.org/10.1109/TSE.2021.3137401
15. Li, X., Li, W., Zhang, Y., & Zhang, L. (2019). DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)* (pp. 169–180). ACM. https://doi.org/10.1145/3293882.3330563
16. Zhang, Q., Fang, C., Ma, Y., Sun, W., & Chen, Z. (2023). A survey of learning-based automated program repair. *Journal of Systems and Software*, *203*, 111711. https://doi.org/10.1016/j.jss.2023.111711