CS527-Team 11: Evaluating Testing and Debugging Tools on Real-world Bugs

Rahul Kumar Veerappa Huleppanour
University of Illinois Urbana-Champaign
rkv7@illinois.edu)

Urja Khadilkar
University of Illinois Urbana-Champaign
uvk2@illinois.edu

Sanjit Kumar
University of Illinois Urbana-Champaign
sanjitk3@illinois.edu

Ajitesh Nair
University of Illinois Urbana-Champaign
ajitesh4@illinois.edu

Sakshil Verma
University of Illinois Urbana-Champaign
sakshil2@illinois.edu

ABSTRACT

This project delves into the analysis and evaluation of bug datasets, testing techniques, and bug localization techniques in software engineering. Leveraging diverse datasets like QuixBugs, BugSwarm, Bears-Benchmark, and Defects4J, we explore various aspects such as bug distributions, bug fix patterns, and bug localization strategies. We employ testing tools like Randoop, Evosuite, and Clover for comprehensive test generation and coverage analysis. Through benchmarking metrics and correlation studies, we aim to offer insights into effective bug detection, testing, and fixing methodologies.

KEYWORDS

Software Testing, Software Debugging, Real-world Bugs

ACM Reference Format:
Rahul Kumar Veerappa Huleppanour, Urja Khadilkar, Sanjit Kumar, Ajitesh Nair, and Sakshil Verma. 2024. CS527-Team 11: Evaluating Testing and Debugging Tools on Real-world Bugs. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 TEAM INFORMATION

Link to the team's GitHub Repo:
https://github.com/urjakhadilkar/CS527-team11
Link to the team's overleaf draft:
https://www.overleaf.com/7831151286tvdfckzycfhb25ef2d

2 INTRODUCTION

Software bugs pose significant challenges in software development, often leading to system malfunctions, security vulnerabilities, and performance issues. Understanding and addressing bugs efficiently is crucial for maintaining software quality and reliability. In this project, we focus on analyzing and evaluating bug datasets obtained from diverse sources such as QuixBugs, BugSwarm, Bears-Benchmark, and Defects4J. These datasets encompass a wide range of bugs in Java and Python programs, providing valuable insights into bug distributions and patterns.

Our project encompasses three main areas: bug datasets analysis, testing techniques evaluation, and bug localization strategies assessment. We explore the characteristics of bugs in terms of complexity, code changes required for fixes, and the effectiveness of bug localization techniques. Additionally, we employ advanced testing tools like Randoop, Evosuite, and Clover to generate comprehensive test suites and measure code coverage.

Through benchmarking metrics such as Cyclomatic Complexity Change (CC), Levenshtein edit distance (LD), and CodeBLEU, we quantify the impact of bug fixes on code complexity and semantic similarity. Furthermore, correlation studies between bug localization metrics and ranks provide insights into the effectiveness of different bug localization techniques.

Overall, our project aims to contribute to the advancement of software engineering practices by offering a deeper understanding of bugs, effective testing strategies, and reliable bug localization techniques.

3 EMPIRICAL SETUP

Five datasets: QuixBugs, BugSwarm, Bears-Benchmark, Defects4j, and ManySStubs4j were selected to perform the evaluation. At a later stage, ManySStubs4j was dropped due to difficulties in extracting and evaluating the bugs. We first aim to study and benchmark the datasets by deriving insights on the quality of the bugs. For this, metrics such as Levenshtein Distance, CodeBLEU, etc were used to gauge the complexity of the bugs. For better bug detection, along with the existing tests, we generate tests by using tools like Randoop and Evosuite. We aim to create regression and error revealing tests that pass on the patched version and fail on the buggy version. We then calculate the suspiciousness score of each statement in the relevant file by executing the tests, generating statement coverage reports, and ranking the statements in order of their suspiciousness. We then explore the correlation between the ranks of the buggy statements with their complexity.

Conference'17, July 2017, Washington, DC, USA

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

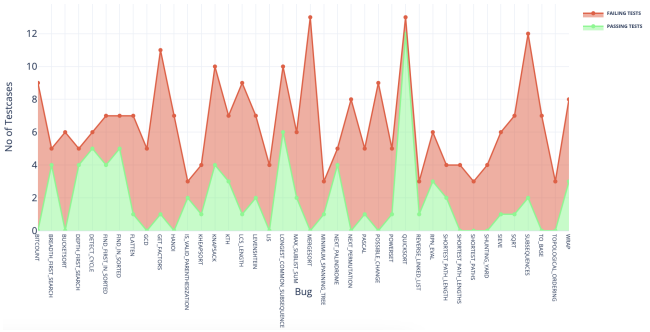


Figure 1: Distribution of Java Test cases- QuixBugs

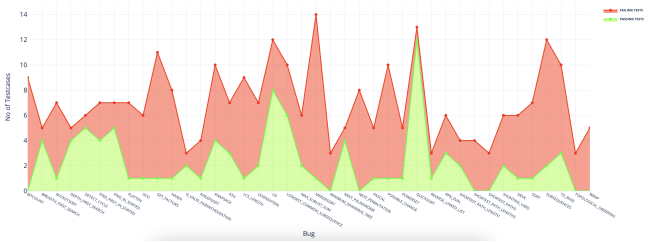


Figure 2: Distribution of Python Test cases- QuixBugs

3.1 Bug Datasets

3.1.1 QuixBugs. Quixbugs [5] is a dataset containing 40 Java and Python programs, each containing a single bug. These buggy programs were built as part of the Quixey Challenge, organized by a startup named Quixey in 2011. These defective programs of classic algorithms were provided to programmers to fix in a minute for a monetary prize. These programs were initially written only in Python. Researchers at MIT and Google translated them into Java to generate a dataset for Multilingual Program Repair. Each bug is accompanied by pass/fail test cases. Figure 1 and 2 shows the distribution of test cases, with the area in green representing the passing test cases on the buggy code and the red area representing the failing tests.

3.1.2 ManySStubs4j. ManySStubs4j [4] is a mined corpus of bugs from 100 Java Maven projects and 1000 Java projects. It contains a total of 170,000 bug fixes, some of which that fall under any of the 16 commonly found categories have been classified as Simple Stupid Bugs (SStuBs). These bug-fixing changes were collected using the SZZ heuristic, and filtered to obtain the simple bugs. These bugs are all stored into JSON files *bugs.json* (100 Java Maven Projects - 25000 bugs) and *bugsLarge.json* (1000 Java Projects - 153,000 bugs). The SStuBs idenetified from these files are stored in *sstubs.json*(100 Java Maven Projects - 10000 SStuBs) and *sstubsLarge.json* (1000 Java Maven Projects - 63000 SStuBs). The JSON files contain fields like 'fixCommitSHA1', 'fixCommitParentSHA1', 'bugFilePath' and 'projectName', which we use to clone the project repositories, checkout to the buggy and fixed versions to extract the buggy and fixed files, and create their diff file.

3.1.3 BugSwarm. BugSwarm [8] is a toolset that helps create a continuously growing set of real-world reproducible build failures and fixes. It involves 2 parts, mining for the bugs from GitHub projects that use Travis-CI, and reproducing by generating scripts to build and run regression tests for each build. The BugSwarm toolset has mined a dataset of 3000 bugs and the tests run on them. The JSON file for the dataset of bugs contains fields like 'repo', 'trigger_sha' for failed and passed builds, 'num_tests_run', and 'num_tests_failed', which we use to clone the repositories, get the diff of the buggy and fixed versions, and then find the buggy and fixed files corresponding to the diff. We also got the test statistics of the bugs in the file *test_info.json*

3.1.4 Bears-Benchmark. Bears-Benchmark [6] aims to provide a comprehensive dataset of software bugs and their fixes, collected from various open-source projects using the Bears-Benchmark. The Bears-Benchmark, or just Bears, is a benchmark of bugs for automatic program repair studies in Java 8. Bugs are collected from open-source projects hosted on GitHub through a process that scans pairs of builds from Travis Continuous Integration and reproduces bugs (by test failure) and their patches (passing test suite). The dataset contains detailed information about each bug, including unique identifiers, commit hashes, file paths, and repository URLs. Additionally, it includes metadata such as bug types, timestamps, and project names. Researchers and developers can leverage this dataset to analyze common types of bugs, study bug fix patterns, and develop automated bug detection and fixing tools. We strive to maintain the accuracy and integrity of the dataset, and contributions or suggestions for improvement are always welcome. Explore the dataset, conduct your research, and join us in advancing software quality and reliability.

3.1.5 Defects4J. Defects4J [3] is a collection of reproducible bugs and a supporting infrastructure (defects4j framework) with the goal of advancing software engineering research. It is a collection of real-world bugs from open-source Java projects that contains buggy and fixed versions of source code. The dataset contains 17 projects. It contains a folder called framework/projects which contains important meta information like the number of active bugs and their SHAs, the relevant test class names, the patch files of src code and test code for each bug. All the bugs are in Java. The total number of bugs are 835 Bugs (+29 deprecated and non-reproducible). The information about the number of tests were obtained via processing information in the framework/projects/relevanttests directory. The breakdown of the tests can be seen in Fig3-8.

3.2 Testing Techniques

3.2.1 Defects4J. Defects4j has a dataset framework that comes with multiple features including finding modified classes for each bug fix, creating test suites with evosuite/randoop and testing present tests in both buggy and patched versions before producing the failing tests. This is the approach that was taken. First, gen_tests.pl was used to create the appropriate tests with Randoop (Regression & Error-Revealing) and Evosuite (Regression) with budget. So we created tests for 24 bugs with about 5+ mins of budget for Randoop Regression tests. The rest could not be generated even with 10+ min budget for each bug. Created tests for 59 bugs using

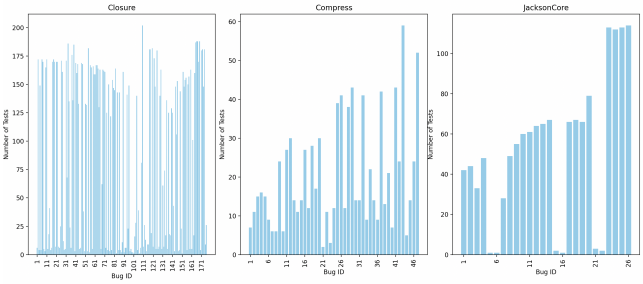


Figure 3: Defects4J Tests Projects 1-3

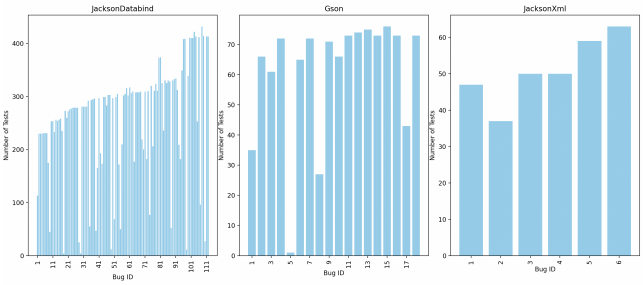


Figure 4: Defects4J Tests Projects 4-6

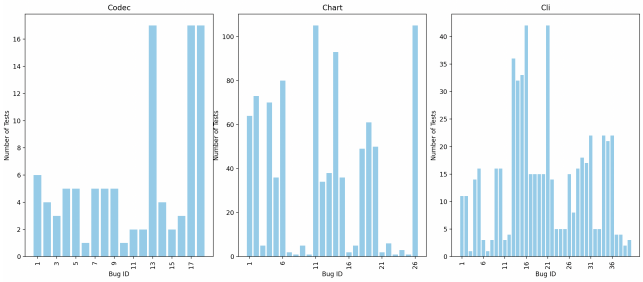


Figure 5: Defects4J Tests Projects 7-9

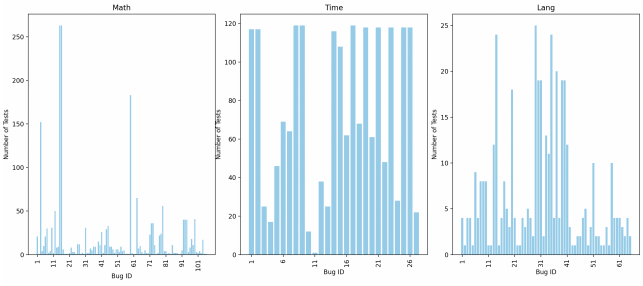


Figure 6: Defects4J Tests Projects 10-12

Randoop for Randoop Regression tests. Created tests for 68 bugs using Evosuite Regression tests with about 3+ min budgets. The bugs that do not have tests have to do with local dependency issues, the Mockito project having specific classes that do not allow the test

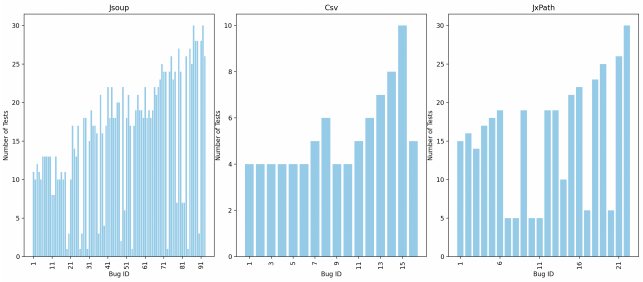


Figure 7: Defects4J Tests Projects 12-15

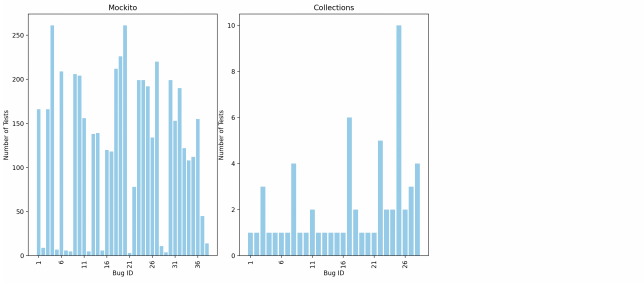


Figure 8: Defects4J Tests Projects 16-17

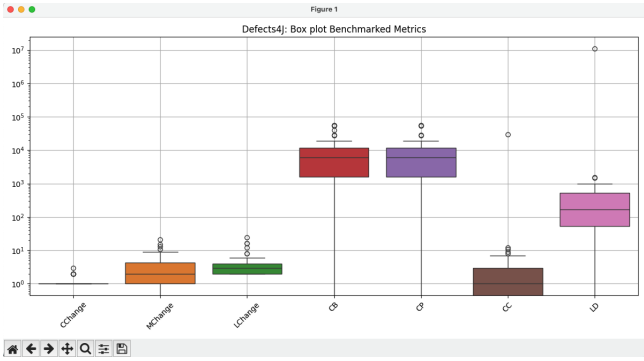


Figure 9: Defects4J Metrics

generation to terminate, Randoop not being able to create Error-Generating tests with 10+ min/bug budgets (10+ hours overall). So we assume that the error revealing behaviour in such cases are hard to spot and encode by the models. After this we run the tests and create .txt files in the Buggy, Patched versions to store the results of "defects4j test" that behind the scenes runs mvn test in both buggy and patched versions. We parse these .txt files to get the tests that pass and fail as required and update failing_tests.txt

3.2.2 Bears. For Bears in this project, we utilized two powerful test generation tools, namely Randoop and Evosuite, to automate the process of generating unit tests for Java projects. Randoop was employed to create both regression tests and error-revealing tests. The former are designed to fail on the buggy version of the code and pass on the patched version, while the latter aim to expose errors specifically in the buggy version. These tests were executed

on targeted classes or methods related to identified bugs, providing a focused approach to test generation.

On the other hand, Evosuite was utilized exclusively for generating regression tests, which were run on both the buggy and patched versions. These tests were crucial in identifying failed tests on the buggy version that passed on the patched version, as well as passing tests on the patched version that failed on the buggy version.

Following the generation of tests using Randoop and Evosuite, a comprehensive coverage report was obtained by executing these tests and collecting test execution results along with coverage information. Clover, integrated as a Maven plugin, facilitated the seamless collection of coverage information during test execution, ensuring a thorough analysis of the test coverage across the software codebase. This meticulous approach to test generation and coverage collection contributes significantly to the overall quality assurance efforts and software reliability enhancement within the project.

3.2.3 QuixBugs. For the QuixBugs dataset, we used two tools: Evosuite and Randoop, to generate tests that passed on the patched version but failed on the faulty code. Tests were first run on both versions of the code. The error-revealing tests were then extracted from these by running them against both versions and verifying that they succeeded on the patched version but failed on the faulty one. Each bug was run for 20 minutes using both Evosuite and Randoop. The tests were carried out by creating new tasks in Gradle.

The test coverage was generated using the Jacoco Gradle plugin. The tools' capacity to generate error-revealing tests for various bugs of different complexities is depicted in box plots of Levenshtein Distance and CodeBLEU distributions.

3.3 Bug Localization Techniques

3.3.1 QuixBugs. Firstly, the lines involved in the bugs was calculated. For this, the lines modified or deleted in the diff were considered and their line numbers were extracted from the buggy file. Then, all tests were run once to extract the names of all available tests. Then each test was run one by one. A code coverage tool, Jacoco was used to generate code coverage for each test execution. The statement coverage information was extracted by parsing the XML coverage report. The Suspiciousness score was then calculated for each statement and ranked in descending order. The first rank and average rank was then calculated for the statements involved in the bug.

3.3.2 Defects4J. First, since the previously used Defects4J framework coverage tool was had insufficient information (only overall hit ratio for each statement), a good amount of time was taken to get some of the projects to generate Clover reports. Once Clover reports were generated, a Selenium script was written to help with extracting the test result vs statement coverage information from the clover sites. The script also simultaneously automatically calculated the suspiciousness score based on the extracted data. We then find out the statements involved in the bug by comparing the buggy files vs patched files. We also rank the statements extracted from clover based on previously calculated suspiciousness score.

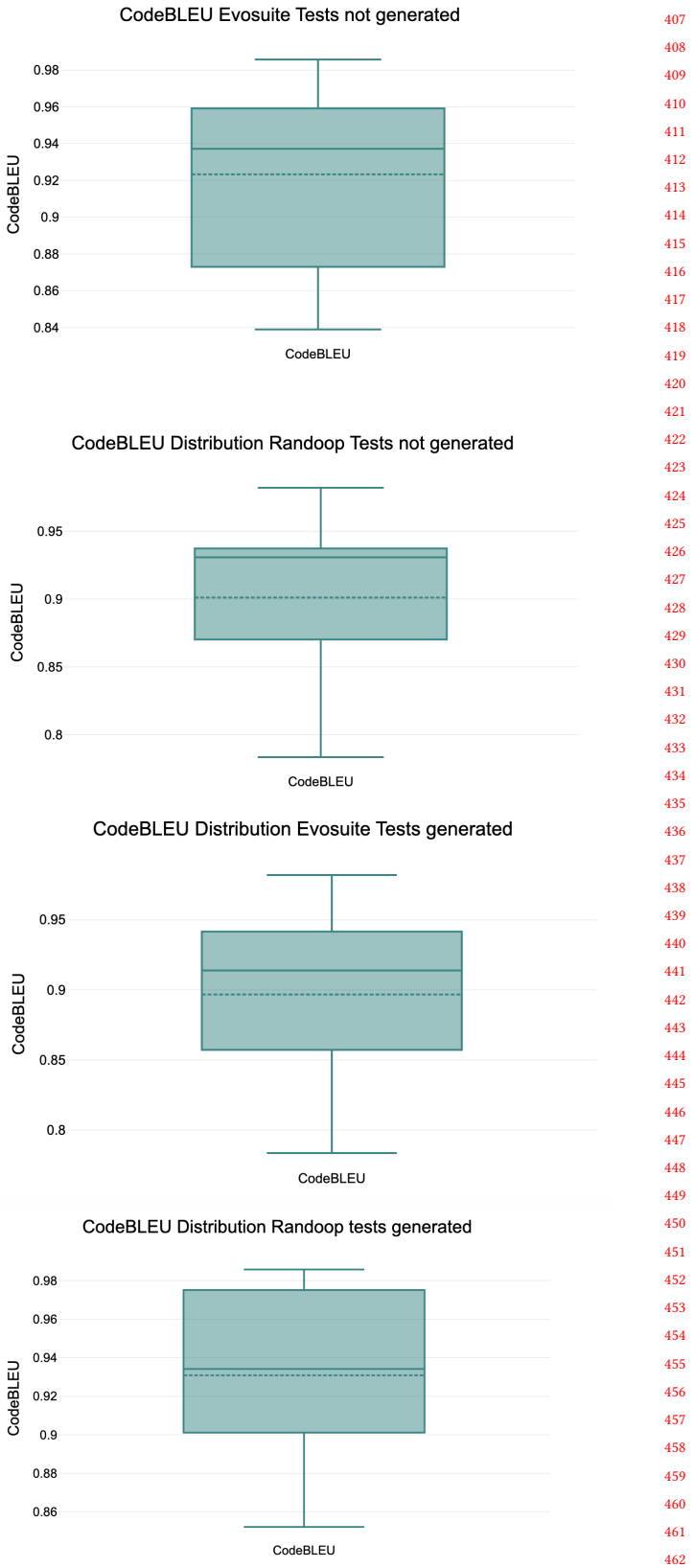


Figure 10: QuickBugs Metrics

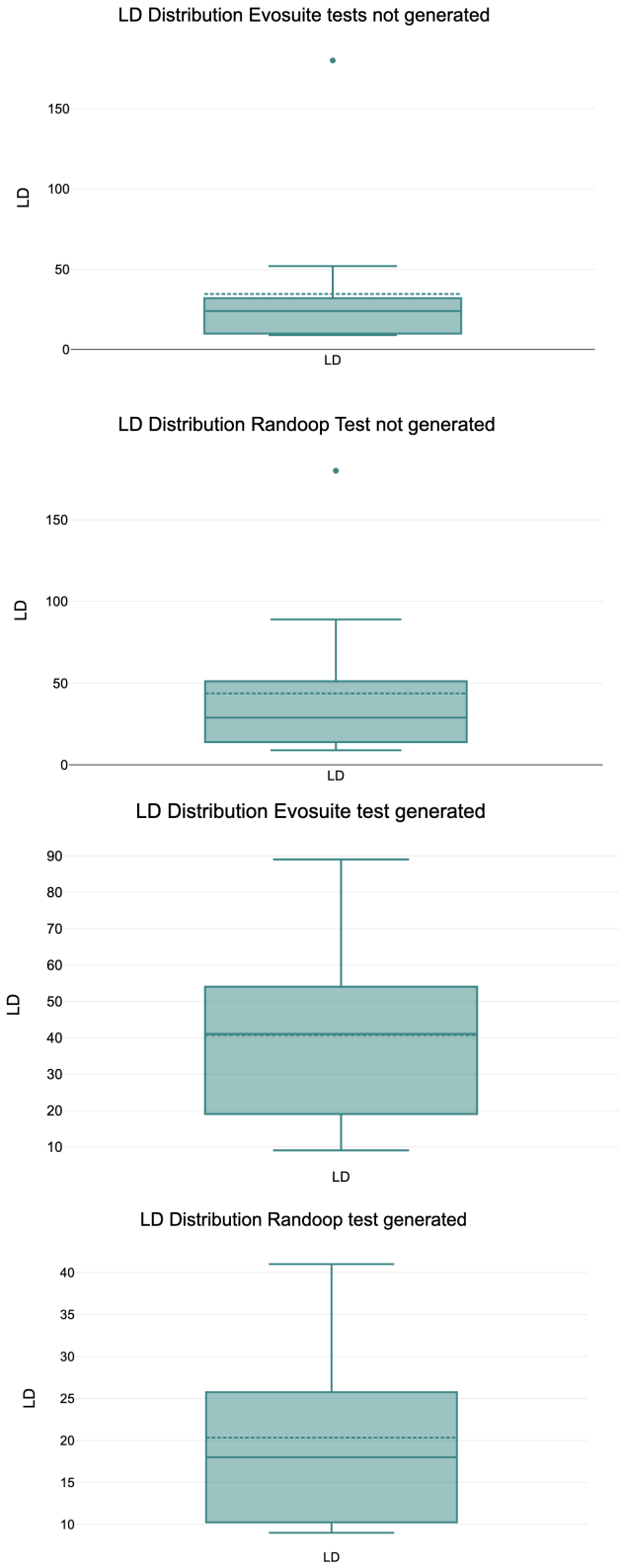


Figure 11: QuickBugs Metrics

The new involved statement line numbers are used to find out the average and first rank of the involved statements in bugs.

3.3.3 Bears. Firstly, the lines involved in the bugs was calculated. For this, the lines modified or deleted in the diff were considered and their line numbers were extracted from the buggy file. Then, all tests were run once to extract the names of all available tests. Then each test was run one by one. A code coverage tool, Jacoco was used to generate code coverage for each test execution. The statement coverage information was extracted by parsing the XML coverage report. The Suspiciousness score was then calculated for each statement and ranked in descending order. The first rank and average rank was then calculated for the statements involved in the bug.

4 EVALUATION

List of research questions:

- **RQ1: Benchmarking Bug Repositories**
  To evaluate the datasets, 8 benchmarking metrics were used: CChange (Number of classes changed/added/deleted to patch the bug), MChange (Number of methods changed/added/deleted to patch the bug), LChange (Number of lines changed/added/deleted to patch the bug), LD (Levenshtein edit distance between buggy and patched files), CB (Cyclomatic complexity of buggy files), CP (Cyclomatic complexity of patched files), CC (Complexity change: |CB-CP|), and CodeBLEU (between buggy and patched files) [7]. Lizard [1] was used for computing the CC and CP, and the Levenshtein Python C extension [2] was used to compute the LD.

  Bears Benchmarking - The analysis of software bug fixes from the Bears-Benchmark dataset reveals several key insights. Across a wide range of bugs, the median CChange, indicating the Cyclomatic Complexity Change, is approximately 1, suggesting minimal alterations to code complexity during bug resolution. Similarly, the median MChange, representing Method Change is 1, and LChange is around 10, denoting Line Changes, are both low, indicating that fixes typically involve modifications to a single method and around 10 lines of code. The median Levenshtein Distance (LD) of approximately 200 highlights the average degree of code modification required for bug resolution. Furthermore, the comparable median values of CB (Cyclomatic Complexity of Buggy Code) and CP (Cyclomatic Complexity of Patched Code) around 35 suggest that bug fixes generally maintain similar complexity levels. Interestingly, the median CC (Cyclomatic Complexity Change) is 0, indicating that complexity tends to remain unchanged post-fix. Lastly, the high median CodeBLEU score of approximately 0.95 signifies significant similarity between patched and buggy code, emphasizing the preservation of code semantics during bug resolution. These findings offer valuable insights for researchers and developers aiming to enhance software quality and reliability through automated bug detection and fixing mechanisms.
  BugSwarm - The benchmarking metrics for the 20 bugs reveal a diverse set of fixes, from minor line edits to more significant structural changes, without dramatically altering the cyclomatic

complexity of the codebase. Most fixes didn't require class modifications, but varied in the number of methods and lines changed, indicating different levels of effort and complexity in resolving the bugs. The Levenshtein distances and CodeBLEU scores suggest that while some bugs required extensive textual modifications, the fixes generally maintained high syntactic and semantic similarity to the original code, indicating precise and effective integration of patches that preserved the original code's intent and readability.

QuixBugs - The average cyclomatic complexity is approximately 5, and it does not differ significantly between buggy and fixed files. The complexity of the majority of bugs remains constant even in patched code, but a handful differ by one. Quixbugs only contains simple bugs, with only one class and method that need patching. The average Levenshtein Edit Distance is roughly 25, implying that just about 25 characters need to be changed in the buggy code to repair it. The average CodeBLEU metric is about 0.94, indicating high similarity between the buggy and patched files.

Defects4J - Defects4J had significantly large repository sizes for bugs but relatively minor code changes to fix bugs. We see that each fix takes on average modifying 1 class, 3 Methods (in invocation or definition), and about 5 lines of code change (Across the 64 working bugs that were actually sampled). The mean cyclomatic complexity difference after patches was about 430 which is a non-trivial number. Computing CodeBLUE the naive way using a list of file names yields incorrect results (0.5 for all bugs) but using an alternative-approach of concatenating file content and computing it yields the correct result in the range of 0.999. However this has been ignored in the results due to high overhead and taking lots of hours on local machine.

- **RQ2: Evaluating Testing Techniques**

Defects4j - We have updated the coverage statistics using run_coverage. from the defects4j framework.

Bears - The set of box plots in Figure 12 shows LD and CodeBLEU metrics for bugs from Randoop. LD has outliers above the main box, with a median of 200-300, while CodeBLEU is tightly distributed around a median just below 0.95. The set of plots in Figure-13 shows LD and CodeBLEU metrics for bugs from Evosuite. LD has a median of 400, with a couple of outliers indicating higher variability, and CodeBLEU maintains a median just below 0.95, with two outliers showing significant deviations. Compared to Randoop, Evosuite's LD distribution is similar but with more variability. Additionally, CodeBLEU for Evosuite exhibits slightly more variability and outliers.

QuixBugs - Distribution of LD and CodeBLEU for the tests generated and not generated by Evosuite and Randoop are shown in Figure-10 and Figure-11. We observe that the number of bugs for which Randoop couldn't generate tests increases with the Levenshtein distance and in general Evosuite is better at generating error revealing tests for complex bugs.

- **RQ3: Evaluating Bug Localization Techniques**

QuixBugs - Distribution of Levenshtein Distance with Rank and CodeBLEU with Rank can be seen in Figure 14 and 15. Since QuixBugs contains, single line bugs, the average rank and first

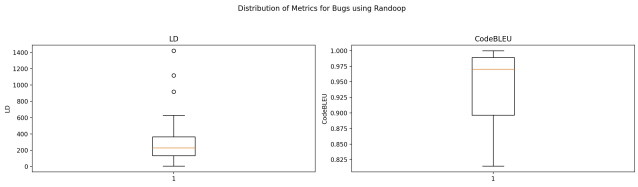


Figure 12: Bears Randoop Metrics

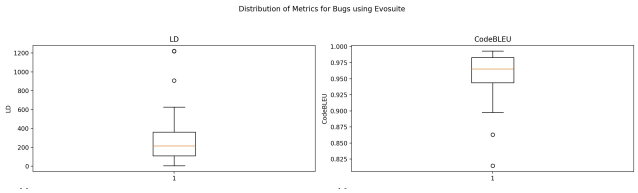
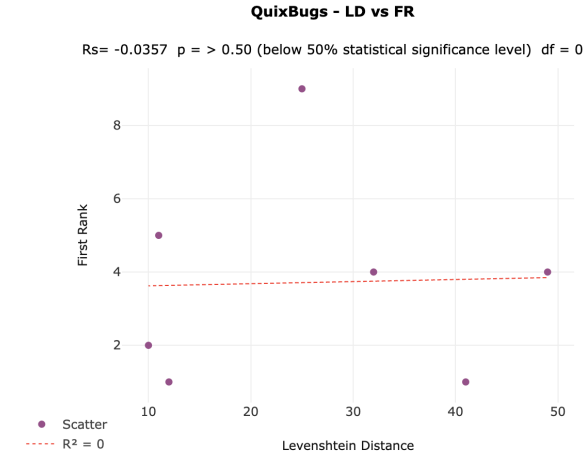


Figure 13: Bears Evosuite Metrics



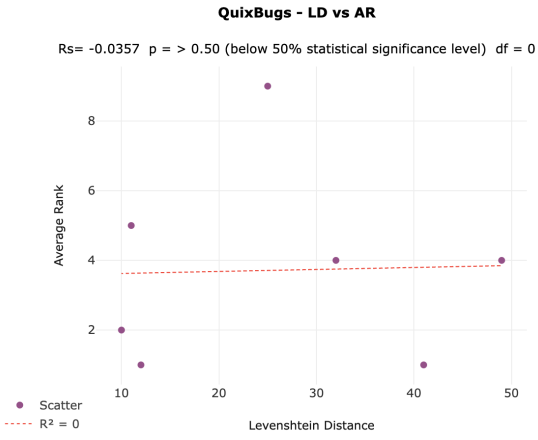


Figure 14: QuixBugs LD vs Rank Metrics

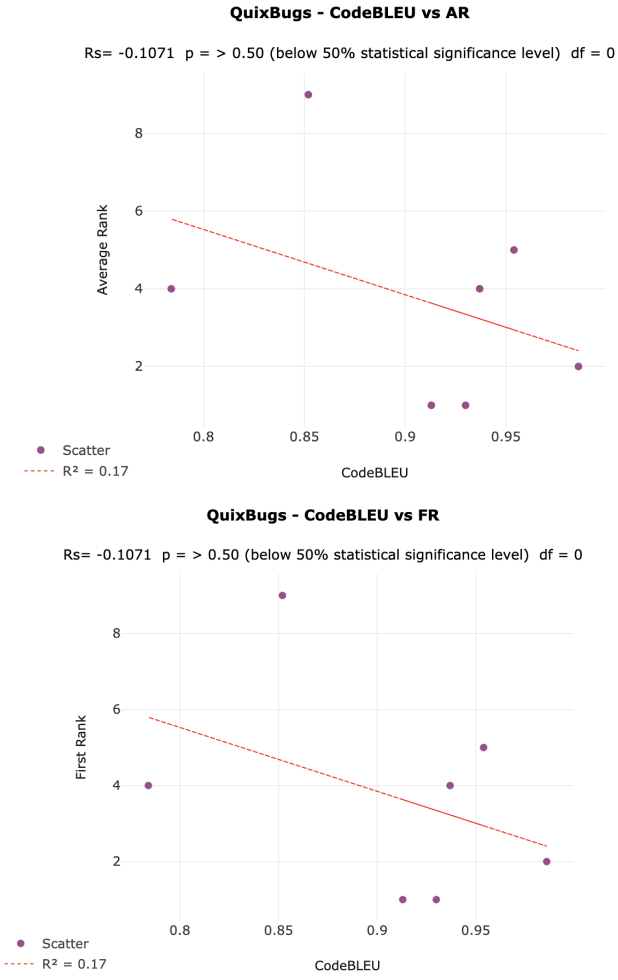


Figure 15: QuixBugs CodeBLEU vs Rank Metrics

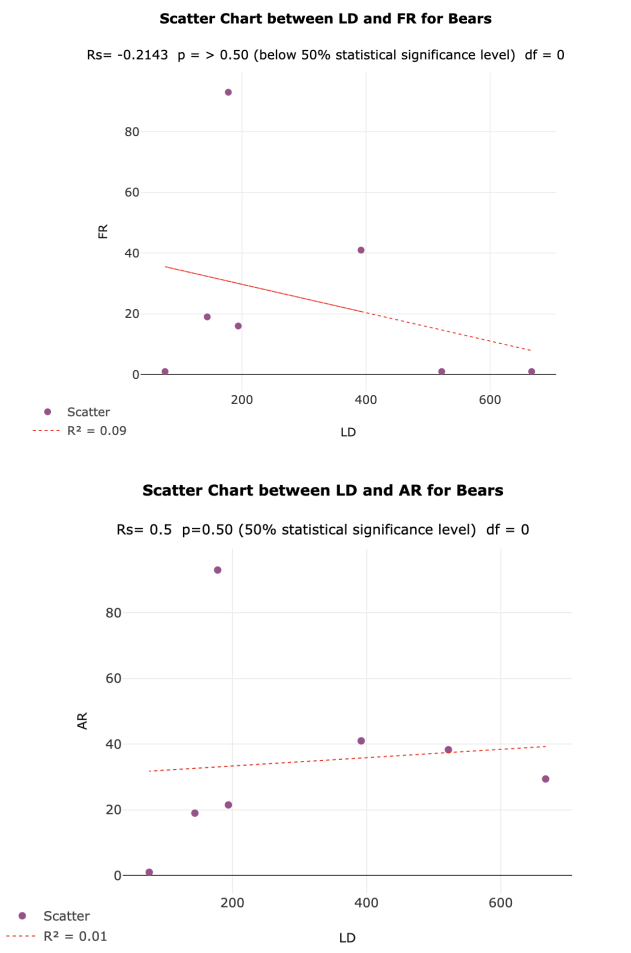


Figure 16: Bears LD vs Rank Metrics

rank for the bugs is the same. We can observe that there is not a very strong correlation between LD and Rank. However, CodeBLEU and Rank is correlated and has a r-value of 0.4. As the CodeBLEU value increases, the Rank decreases. The Spearman rank order correlation indicates a weak negative correlation for both LD and CodeBLEU with Ranks. The ranks of the bugs were 1,1,2,4,4,5 and 9. Almost all the bugs were localized within the top 5 ranks.

Bears - Distribution of Levenshtein Distance with Rank and CodeBLEU with Rank can be seen in Figure 16 and 17 for the Bears Dataset. Since bears contains, single line bugs for Bears-25, Bears-107 and Bears-114, the average rank and first rank for the bugs is the same, but for Bears-3, Bears-17, and Bears-101 multiple line bugs thus the average rank and first rank for the bugs was different. We can observe that there is a very weak correlation between LD and Rank. However, CodeBLEU and FR is correlated and has a r-value of 0.58 and CodeBLEU and AR also has a weak correlation and a r-value of 0.22. As the CodeBLEU value increases, the Rank increases. The Spearman rank order correlation indicates a weak negative correlation for both LD and CodeBLEU with Ranks. The ranks of the bugs were 41,19,93,1,21.5,38.3 and 29.4.

Defects4J - Fig. 18 and Fig. 19 show the distribution of LD and CodeBlue against the rank metrics calculated in Defects4J. From Fig. 17, we notice that in Defects4J the rank metrics do not show any strong correlation against Leveshtein Distance. There is no real trend, the R value is 0.13 and 0.06 and Spearman Cooefficient is 0.5. From Fig. 18, we notice that similarly Rs values are 0.43 and -0.54 which indicates a moderate correlation between the CodeBleu metric values and the Rank metrics.

5 CHALLENGES

We faced quite a few challenges with some datasets - especially in terms of building/running projects, generating tests and building them.

Defects4J - 1. Building and running bugs from JFreeCharts project was extremely challenging due to the use of SVN and compatibility mismatch restrictions in the local machine used to build

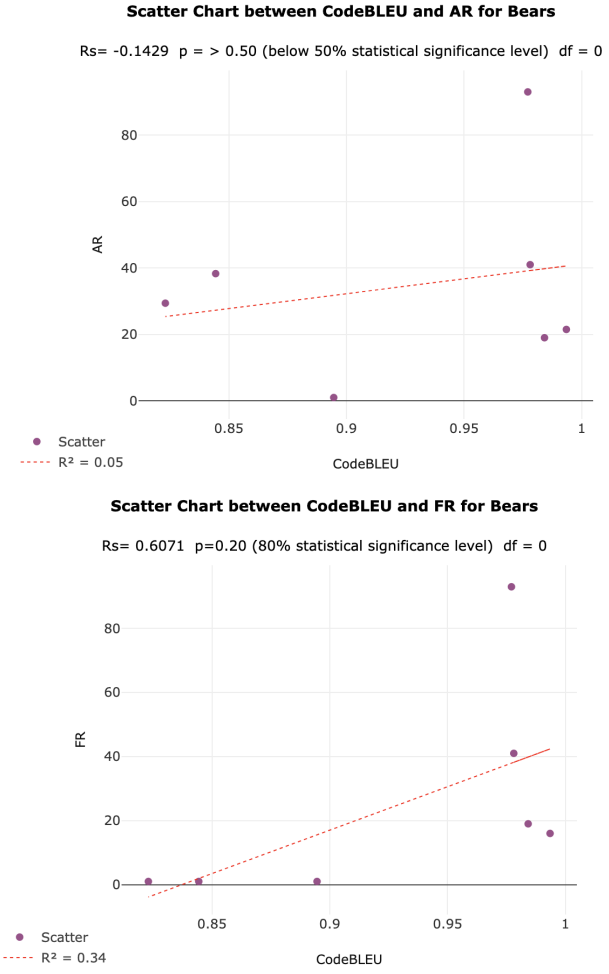


Figure 17: Bears CodeBLEU vs Rank Metrics

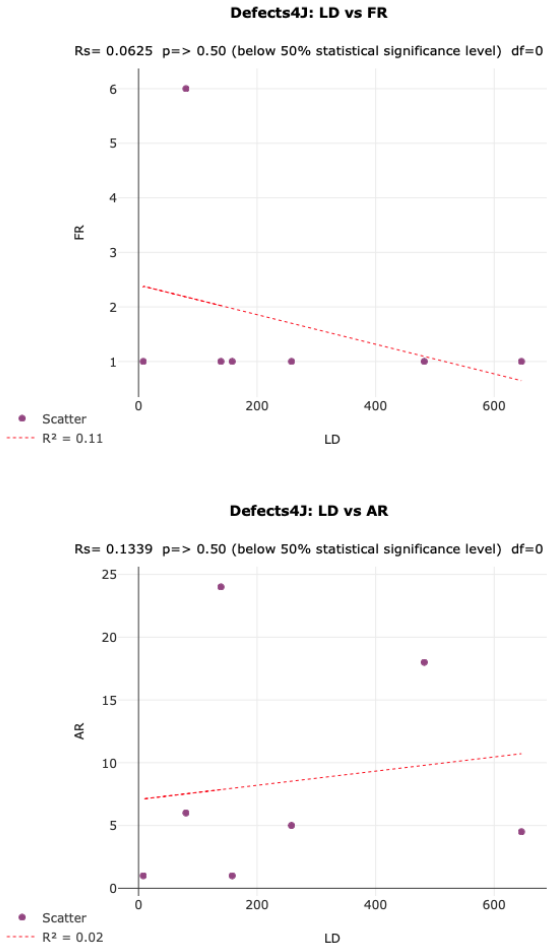


Figure 18: Defects4J LD vs Rank Metrics

them. This could not be resolved by installing other versions because the defects4j framework refused to resolve the version mismatch based on path changes.

2. Coverage report generation via the recommended tool - Clover. Generating coverage reports for Defects4J dataset bugs with Clover proved very troublesome because of 2 reasons - Firstly, the generated bugs via Randoop, Evosuite did not build and run with the projects natively (had compilation errors). Secondly, Clover would not natively work with the bug projects without significant modification to the project pom.xml (where each project required vastly different changes) based on their dependency structure. The primary reason that we identified for these were, defects4j authors did not seem to have intended the use building clover or building the tests directly along with the other maven tests. The original intention we believe was to use the native framework that comes with the dataset that generated randoop/evosuite tests in archive files and directly took the archive files to generate coverage reports. Resloving technical issues around these were time consuming.

3. Calculating suspiciousness score "with" the generated tests also was really difficult for the vast majority of the bugs for the previously stated reason of compilation errors. We were able to get correct reports for the 'Cli' java project and thus achieved the expected deliverables.

BugSwarm - 1. For the BugSwarm dataset, we encountered challenges in generating tests using the tools, primarily due to dependency issues, file path discrepancies, and problems related to Java classes. Given the unique structuring and the presence of multiple nested modules in each project within the BugSwarm dataset, devising scripts to automate the test generation process proved difficult.

2. We encountered significant challenges with JaCoCo coverage reports due to the multi-module nature of the BugSwarm dataset. Each module generated its own coverage report, which prevented us from obtaining a unified project-wide view. Additionally, the lack of information about failing tests in the reports hindered our ability to calculate suspiciousness scores effectively. This limitation also impacted our plans to correlate average rank (AR) and first

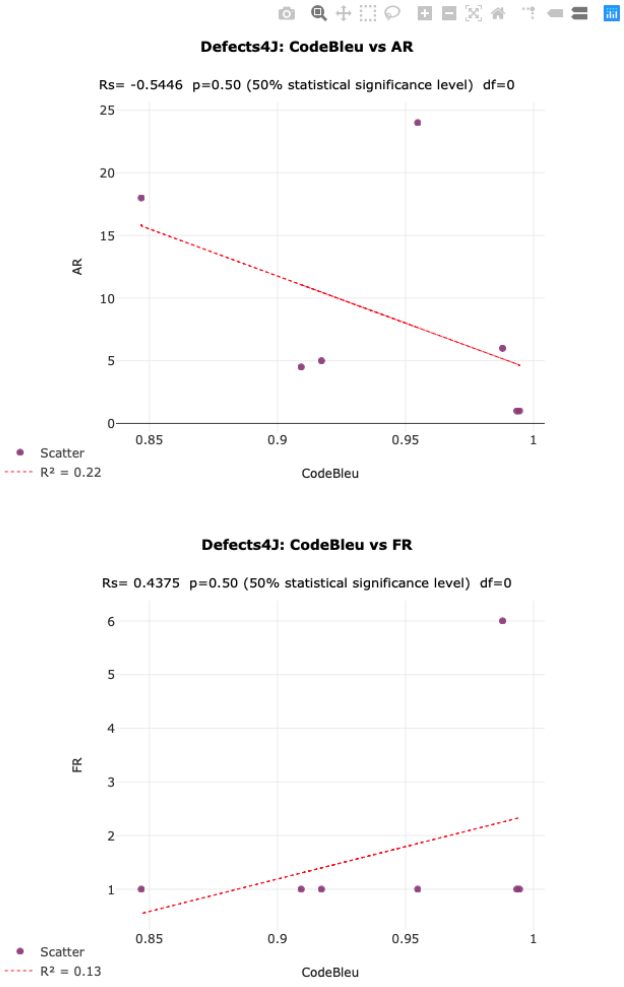


Figure 19: Defects4J CodeBleu vs Rank Metrics

rank (FR) metrics with benchmarking metrics, ultimately restricting the scope of our analysis.

6 CONCLUSION

In conclusion, 4 popular datasets (Defects4J, Bears, BugSwarm, QuixBugs) were analyzed for their real world bugs .This was done by sampling bugs (both buggy and fixed versions) from the datasets from different open-source projects to find applied patches and failing tests before they were benchmarked for their complexity. Then automated test generation was done via Randoop and Evosuite for sampled bugs and coverage tests generated with appropriate tools (for each dataset). Then average and first ranks based on suspiciouness scores were used to localize buggy statements. We visualize findings and detail challenges faced during the above tasks.

REFERENCES

[1] [n. d.]. Lizard. https://github.com/terryyin/lizard
[2] [n. d.]. Python-Levenshtein. pypi.org/project/python-Levenshtein
[3] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. Association for Computing Machinery, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055
[4] Rafael-Michael Karampatsis and Charles Sutton. 2020. ManySStuBs4J Dataset.
[5] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Vancouver, BC, Canada) *(SPLASH Companion 2017)*. Association for Computing Machinery, New York, NY, USA, 55–56. https://doi.org/10.1145/3135932.3135941
[6] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. https://arxiv.org/abs/1901.06024
[7] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: a Method for Automatic Evaluation of Code Synthesis. arXiv:2009.10297 [cs.SE]
[8] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes. In *ICSE*. IEEE / ACM, 339–349.