# Poster: Pairika—A Failure Diagnosis Benchmark for C++ Programs

Md. Rezaur Rahman
Technical University of Munich
reza.rahman@tum.de

Mojdeh Golagha
Technical University of Munich
mojdeh.golagha@tum.de

Alexander Pretschner
Technical University of Munich
alexander.pretschner@tum.de

## ABSTRACT

Empirical studies in software testing require realistic benchmarks which are able to mimic industry-like environments. For evaluating automated failure diagnosis techniques, one needs real reproducible bugs with at least one associated failing test. Extracting such bugs is challenging and time-consuming. This paper presents Pairika, a failure diagnosis benchmark for C++ programs. Pairika contains 40 bugs extracted from 7 modules of OpenCV project with more than 490 KLoC and 11129 tests. Each bug is accompanied by at least one failing test. We publish Pairika to facilitate and stimulate further research on automated failure diagnosis techniques. Pairika is available at: https://github.com/tum-i22/Pairika

## KEYWORDS

Real Bugs, Failure clustering, Fault localization, C++ Benchmark

## 1 INTRODUCTION

Debugging is tedious and time-consuming even in well-developed software. Related processes can be automated to reduce debugging time. Testing automates failure detection. A failure is an observed deviation between actual and intended behavior [10]. Failure clustering and fault localization techniques automate failure diagnosis and fault detection. A fault or a bug is the reason for the failure [10]. Finally, software repair techniques automate the process of fixing the faults.

In order to assess the relative effectiveness and efficiency of these automated techniques, we need realistic benchmarks that mimic different characteristic of industry-like environments. Wong et. al. [11] and De Souza et. al. [3] listed a summary of popular subject programs used in fault localization and repair studies. Most of the openly available subjects are written in C and Java.

In our paper, we introduce Pairika, a new benchmark for C++ programs. To build Pairika, we chose OpenCV, a library of programming functions aimed at real-time computer vision written in C and C++. We reviewed 247 issues from issue tracking system of OpenCV. We extracted 40 *isolated*, *reproducible* bugs which are related to the *source code*.

Pairika is the first publicly accessible benchmark for C++ programs. To the best of our knowledge, there isn't any other benchmark for C++ programs with real bugs. Using Pairika, researchers will be able to evaluate fault localization and failure clustering techniques.


*ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden*

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195097

## 2 OPENCV REAL BUGS

OpenCV (Open Source Computer Vision Library) is an open-source computer vision and machine learning software library. The library is written in C and C++ and runs under Linux, Windows and Mac OS X. It has 52 modules [9]. For reporting issues and requesting features, OpenCV has an issue tracking system [1]. We reviewed 247 "Closed" issues that were labeled as "Bug" or "Feature". Among the reviewed issues we could extract 40 reproducible bugs that each yields at least one failing test. Table 1 demonstrates the related modules.

**Table 1: OpenCV Modules**

| Module | LoC | #ofFiles | #ofTests | #ofBugs |
|---|---|---|---|---|
| Core functionality (core) | 196550 | 345 | 10528 | 15 |
| Machine learning (ml) | 19398 | 34 | 39 | 5 |
| Calibration and 3D reconstruct (calib3D) | 44647 | 84 | 87 | 2 |
| 2D features (features 2D) | 60557 | 101 | 119 | 1 |
| Deep neural network (dnn) | 147671 | 162 | 122 | 13 |
| Video analysis (video) | 13895 | 48 | 68 | 1 |
| Computational photography (photo) | 10880 | 47 | 166 | 3 |

## 3 HOW IS PAIRIKA USEFUL?

We believe Pairika facilitates and stimulates further research on automated failure clustering and fault localization. Our first proposal for further research in the area of automated fault localization is *exploring the impact of bug complexity on the accuracy of fault localization.*

The complexity of a bug is measured by "how substantial the bug's fix is required to be"[1]. Thus, to measure the complexity of a bug, we measure the complexity of the best *fix* of the bug. We define the following metrics to measure the complexity of a fix:

- Changed Functions of code (CFuoC): The number of executable source functions that were changed in the fix.

[1] https://github.com/opencv/opencv/issues

- Changed Lines of Code (CLoC): The number of executable source code lines that were changed in the fix [1].
- Changed Files of Code (CFioC): The number of executable source files that were changed in the fix.
- Bug Fixing Life Span (BFLS): The number of days between reporting a bug and resolving it [1].
- Number of Participants (NoP): The number of participants in bug fixing discussion.
- Number of Failing Tests (NoFT): The number of failing tests caused by the bug to measure the detectability of a bug [1].

We measured the aforementioned metrics for all the bugs we found in 7 modules. Table 2 demonstrates the measured values for only 7 randomly selected bugs to conserve space. The complete table can be found at the project's repository.

**Table 2: OpenCV Bug Complexity**

| BugID | CFuoC | CLoC | CFioC | BFLS | NoP | NoFT |
|---|---|---|---|---|---|---|
| ml2 | 3 | 26 | 1 | 86 | 5 | 2 |
| ml3 | 1 | 1 | 1 | 4 | 2 | 1 |
| photo1 | 1 | 4 | 1 | 15 | 2 | 2 |
| photo2 | 1 | 16 | 1 | 54 | 4 | 3 |
| calib3d2 | 1 | 2 | 1 | 1 | 4 | 1 |
| core5 | 1 | 6 | 1 | 3 | 2 | 17 |
| core14 | 1 | 1 | 1 | 1 | 2 | 1 |

One of the state-of-the-art techniques in automated fault localization is the spectrum-based technique which ranks program elements based on their suspiciousness to be faulty. We used Aletheia [5] to do method-level spectrum-based fault localization [11] on the selected bugs. The generated hit spectra and results can be found at the project's repository. The ranks of the faulty functions are: 11 in ml2, 4 in ml3, 4 in photo1, 1 in photo2, 1 in calib3D2, 32 in core5, and 304 in core14.

Based on the results, it seems that it is harder to localize faults with higher CFuoC or CLoC values. However, these results are not enough to conclude there is a correlation between the complexity of a bug and the localization accuracy. We employed this small experiment to stimulate further studies on exploring the relationship between bug complexity and accuracy of fault localization.

The bug in core14 is in a class constructor. The result shows that a method-level analysis is not suitable for this kind of bugs. Thus, as the second proposal, we suggest *exploring the relationship between bug types, granularity levels and fault localization accuracy.*

In addition, adding a new C++ benchmark to the existing Java benchmarks provides the opportunity for the community to *explore the relationship between design metrics, domain-specific features, or object-orienting features, and accuracy of fault localization.*

Finally, Some of the bug fixes impose changes in more than one location (e. g. file, function) in the code. However, the state-of-the-art automated fault localization techniques are not able to detect all of these locations. This motivates *exploring for more effective fault localization techniques.*

## 4 RELATED WORK

The Siemens suite [6] is the first benchmark of buggy programs. It consists of 7 C subjects with varying sizes between 141 and 512 LoC. All bugs are manually injected into the code.

Software-artifact infrastructure repository (SIR) [4] is another benchmark introducing reproducible bugs. SIR consists of 85 subjects written in Java, C, C++, and C#. The subject sizes vary between 24 to 503833 LoC. Most of the bugs introduced in SIR are artificial, either hand-seeded or obtained by mutation. It has only one C++ subject with 1034 LoC and artificial bugs. Pairika's subjects have between 10880 and 196550 LoC and all the bugs are real.

CoREBench [1] is a collection of 70 regression bugs extracted from 4 C projects with more than 145 KLoC. ManyBugs [8] consists of 1183 defects extracted from 15 C programs with ca. 5900 KLoC.

iBugs project [2] is a database of real bugs for Java programs. It contains 3 subjects and 390 bugs extracted from version control history. 223 of the bugs are associated with failing tests. Nevertheless, the faulty versions can only be built with an outdated version of the JVM [7]. Defects4J [7] is a database of reproducible bugs for Java programs. It provides 357 bugs out of 5 subjects with 352 KLoC.

In contrast to CoREBench, ManyBugs, Defects4J and iBugs, Pairika is a benchmark for C++ programs. We believe that having realistic benchmarks for both Java and C++ programs provides the opportunity for researchers to explore the relationship between object-orienting features and fault localization accuracy.

## 5 CONCLUSION

This paper presents Pairika, a failure diagnosis benchmark for C++ programs. Pairika consists of 40 real, reproducible bugs that are extracted from the issue tracking system of OpenCV project. Each bug is accompanied by at least one failing test. Pairika contains 490 KLoC and 11129 tests. This makes Pairika a suitable benchmark for failure clustering and fault localization studies. We believe that publishing a new benchmark for C++ facilitates and stimulates further research on automated failure diagnosis techniques.

## REFERENCES


[1] M. Böhme and A. Roychoudhury. 2014. CoREBench: studying complexity of regression errors. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. 105–115.

[2] V. Dallmeier and Th. Zimmermann. 2007. Extraction of bug localization benchmarks from history. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering - ASE '07*. 433.

[3] H. A. De Souza, M. L. Chaim, and F. Kon. 2016. Spectrum-based Software Fault Localization: A Survey of Techniques, Advances, and Challenges. *CoRR* abs/1607.04347 (2016).

[4] H. Do, S. Elbaum, and G. Rothermel. 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. In *Empirical Software Engineering*, Vol. 10. 405–435.

[5] M. Golagha, A. M. Raisuddin, L. Mittag, D. Hellhake, and A. Pretschner. 2018. Aletheia: A Failure Diagnosis Toolchain. In *To appear in: 2018 IEEE/ACM 40th International Conference on Software Engineering Companion*.

[6] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. 1994. Experiments on the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria. *Proceedings of 16th International Conference on Software Engineering* JANUARY 1994 (1994), 191–200.

[7] R. Just, D. Jalali, and M. 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 - ISSTA 2014*. 437–440.

[8] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (Dec 2015), 1236–1256.

[9] OpenCv. 2014. OpenCV Library. (2014). https://doi.org/OpenCV

[10] A. Pretschner. 2015. Defect-Based Testing. In *Dependable Software Systems Engineering*.

[11] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. 2016. A Survey on Software Fault Localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.