The Journal of Open Source Software

deflake.rs: Detect Flaky Tests in Rust Projects using Execution Data

Benjamin Magill[1] and **Phil McMinn**[1]

1 University of Sheffield, United Kingdom

DOI: 10.21105/joss.08757

Software

- Review
- Repository
- Archive

Editor: Richard Littauer

Reviewers:

- @jabhalla
- @djmitche

Submitted: 04 August 2025
Published: 08 September 2025

Summary

In automated software testing, flaky tests are test-cases which fail non-deterministically, without any modifications to the code under test (Parry et al., 2021). They can occur in any programming language, and pose a painful issue for developers due to their randomness and uncontrollability.

To deal with flaky tests, many tools have been created to detect them so that developers can fix, ignore or remove them. Such tools may detect flaky tests by simply running test-cases multiple times and checking for diverging results, or use more advanced methods which take into account static (data collectable without running any tests) and/or execution data (data collected by running the test suite) to determine or predict if a test is flaky without multiple runs.

`deflake.rs` is a tool for detecting flaky tests in Rust projects without re-running failed tests, and due to this provides an efficient method for quickly detecting flaky tests that can assist developers and save them time. It uses per-test code coverage and the changes to the source code since the last good version to determine if the code run by a failing test includes anything that was modified, and if not it will report the test as flaky. This method was originally proposed by DeFlaker (Bell et al., 2018), a tool for Java programs which showed it could accurately detect flaky tests with high accuracy across a diverse range of projects. `deflake.rs` builds upon this work by showing that it can be adapted to other languages, while also showing how the necessary execution data can be collected in Rust for this and any detection methods. The tool is also easy to use within any Rust project as it requires no modifications to the codebase to work, making it simple for developers to adopt.

Statement of need

While some languages have been the focus of lots of research on flaky test detection (Tahir et al., 2023) (e.g. Java (Bell et al., 2018; Lam et al., 2019) and Python (Cordeiro et al., 2021; Parry et al., 2020)), others such as the Rust programming language have not.

The most advanced tool for detecting flaky tests in Rust is Nextest (Rain & others, 2025), which is an alternative test runner that supports rerunning failed tests multiple times to detect flaky tests. The default test runner "`cargo test`", which is used my most Rust developers, does not include any functionality for handling or detecting flaky tests. Compared to the state of the art, this is very far behind what has been shown to be possible and is widely available in other languages, and the limited diversity of detection tools available means developers can't use the best tool for their specific needs. Reruns can also be very time consuming, making them very undesirable in some circumstances.

`deflake.rs` therefore bridges this gap by developing the first flaky test detection method for Rust that does not rely on rerunning tests, showing that Rust can support advanced methods of flaky test detection.

Technical details

In order to collect the data used for classification, `deflake.rs` uses “`cargo test`” internally to build and collect all of the test-cases, but executes each individually to retrieve the necessary per-test coverage, which is not possible otherwise. This is because the code coverage for a program is only available program upon exit, and as “`cargo test`” executes all tests in each file at once it would be impossible to separate the coverage for each test after it exits, or collect it after each test finishes while the program is still running. By using “`cargo test`” under the hood for compiling tests, all projects that work with it will also work with `deflake.rs`, lowering the barrier to entry and making it very accessible for developers to use.

To determine the code affected by modifications, “`git diff`” is used to collect a list of all files which have been modified, and their changes since the previous or specified commit. For each modified file, the Abstract Syntax Tree (tree structure representing the source code) is generated for both the new and old version, and these are compared to find how the structure of both versions varies. This does not compare the changes to statements in functions, only the functions, modules and other structures that can include executable statements, as shown in Figure 1.

```
mod module {
  fn function_a(a: bool) -> bool {
    ...
  }

  struct A;
  impl A {
    fn function_c(a: u32) {
      ...
    }
  }
}

fn function_c(
  a: u32,
  b: u32
) {
  ...
}
```

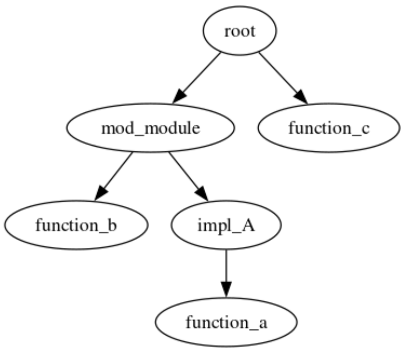

Figure 1: An example tree structure and its associated code

By comparing the change of structure, the functions which have been added or moved can be found and tracked for their overall execution instead of the statements within them. For all other changes, the statements modified are determined from the diff and used to directly determine if any modified code was executed per test. By combining the use of both AST's and "`git diff`", `deflake.rs` is able to more effectively determine how each file has changed, and simplify tracking if these were executed.

To classify a test as flaky the program will simply check if any line or function that was modified was executed, and if not then it is predicted to be flaky.

Limitations

While the described approach will work for detecting flaky tests under most changes to a project, there are some limitations. For one, any test that was modified and fails will be classified as non-flaky no matter what. The tool also requires that the commit being compared against is one where all of the test cases succeed or are flaky. If not, the failures from the previous commit will be classified as flaky (if they are not modified). Lastly, the tool can only track changes to Rust files, so any changes to other files which have impacted a tests result will not be detected and the test case will be marked as flaky.

While it can't be perfectly accurate due to these limitations, as it does not use rerunning tests to detect flakiness it can greatly speed up the process of detecting most flaky tests, especially for larger and slower test suites.

Acknowledgements

We would like to thank Owain Parry and Zalán Lévai for their contributions to the direction of the project. Phil McMinn is funded, in part, by EPSRC grant "Test FLARE", EP/X024539/1.

References

Bell, J., Legunsen, O., Hilton, M., Eloussi, L., Yung, T., & Marinov, D. (2018). DeFlaker: Automatically detecting flaky tests. *Proceedings of the 40th International Conference on Software Engineering*, 433–444. https://doi.org/10.1145/3180155.3180164

Cordeiro, M., Silva, D., Teixeira, L., Miranda, B., & d'Amorim, M. (2021). Shaker: A Tool for Detecting More Flaky Tests Faster. *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 1281–1285. https://doi.org/10.1109/ASE51524.2021.9678918

Lam, W., Oei, R., Shi, A., Marinov, D., & Xie, T. (2019). iDFlakies: A Framework for Detecting and Partially Classifying Flaky Tests. *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 312–322. https://doi.org/10.1109/ICST.2019.00038

Parry, O., Hilton, M., Kapfhammer, G. M., & McMinn, P. (2021). A survey of flaky tests. *ACM Transactions on Software Engineering and Methodology*. https://doi.org/10.1145/3476105

Parry, O., Kapfhammer, G. M., Hilton, M., & McMinn, P. (2020). Flake It 'Till You Make It: Using Automated Repair to Induce and Fix Latent Test Flakiness. *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, 11–12. https://doi.org/10.1145/3387940.3392177

Rain, & others. (2025). *Nextest*. https://github.com/nextest-rs/nextest

Tahir, A., Rasheed, S., Dietrich, J., Hashemi, N., & Zhang, L. (2023). Test flakiness' causes, detection, impact and responses: A multivocal review. *Journal of Systems and Software*, *206*, 111837. https://doi.org/10.1016/j.jss.2023.111837