PYURIFY: Purifying Python Tests for Precise Fault Localization

Marius Smytzek
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
marius.smytzek@cispa.de

Andreas Zeller
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
zeller@cispa.de

Abstract

Statistical fault localization (SFL) techniques use test coverage to rank suspicious program elements. However, they often suffer from low precision when tests contain multiple assertions or unrelated code that obscures the relationship between test outcomes and faulty code. Test case purification addresses this problem by transforming complex test cases into simplified, single-assertion variants. We present PYURIFY, a tool that implements test case purification for Python through two steps: first, atomization splits each test with multiple assertions into separate single-assertion tests; second, dynamic program slicing removes code irrelevant to each assertion. This purification process produces cleaner test spectra, enabling SFL techniques to pinpoint faulty code more accurately. PYURIFY provides both a command-line interface and a Python API for seamless integration with existing testing and fault localization workflows. Our evaluation of real Python bugs demonstrates that test case purification consistently improves fault localization accuracy.

CCS Concepts

• **Software and its engineering** → **Software testing and debugging**; **Software maintenance tools**; • **General and reference** → **Metrics**; **Measurement**.

Keywords

fault localization, statistical debugging, automated debugging

1 Introduction

When a test fails, developers must locate the fault in their code, a task that can be time-consuming and error-prone. Statistical fault localization (SFL) techniques [1, 8] aim to accelerate debugging by ranking program elements according to their correlation with test failures. However, SFL effectiveness depends on the quality of test spectra: noisy test cases can obscure the relationship between code execution and failures.

A common source of noise is that tests often contain multiple assertions and unrelated code. When such a test fails, SFL treats the entire execution as equally relevant, even though only a subset contributes to the failing assertion. *Test case purification* [29] addresses this by transforming multi-assertion tests into focused, single-assertion variants through atomization (splitting tests) and slicing (removing irrelevant code). While test case purification was initially proposed and evaluated for Java programs, no practical implementation exists for Python, which has become dominant in many software domains, including data science and machine learning. Python's dynamic nature and distinct execution model present unique challenges for implementing test case purification.

We present PYURIFY (Test Case Purification), an open-source tool that brings test case purification to the Python ecosystem. PYURIFY combines test case atomization with dynamic slicing to generate purified test cases that improve the effectiveness of statistical fault localization. Additionally, PYURIFY incorporates a refined ranking strategy that combines traditional suspiciousness scores with coverage ratios from purified tests, as proposed by Xuan et al. [29]. The tool is designed for practical use with both a simple command-line interface and a programmatic API, allowing seamless integration into existing debugging workflows. PYURIFY is intended for both researchers and practitioners who wish to improve fault localization and debugging in Python projects.

Overall, this paper makes the following contributions:

Practical test case purification for Python. A robust implementation that handles PYTEST-based test suites with automatic detection of assertions and test structure.

Efficient dynamic slicing. A dynamic slicing algorithm based on execution tracing and data-flow analysis that removes irrelevant code while preserving test semantics.

Seamless integration. PYURIFY can be easily integrated through a command-line interface, Python API, and compatibility with standard fault localization frameworks.

An empirical evaluation. PYURIFY is evaluated on real bugs from popular Python projects, demonstrating consistent improvements in fault localization metrics.

An open-source tool. PYURIFY is available as open source.

The remainder of this paper is organized as follows. Section 2 provides background on statistical fault localization, and Section 3 explains the test case purification approach. Section 4 describes our Python implementation, and Section 5 demonstrates the tool's usage. Section 6 presents our empirical evaluation. Section 7 discusses threats to validity. Section 8 reviews related work, and Section 9 concludes with future work.

2 Statistical Fault Localization

Statistical fault localization (SFL) techniques rank program elements according to their correlation with test failures. These techniques collect test coverage spectra indicating which program elements (lines, statements, or branches) are executed by each test. For each program element, a suspiciousness score is computed based on how often the element appears in failing versus passing test executions. Common suspiciousness formulas include TARANTULA [8], OCHIAI [1], and JACCARD [3]. The effectiveness of SFL depends critically on the quality of test coverage spectra. When test cases contain multiple assertions and unrelated code, the coverage spectra become noisy. A failing test with multiple assertions provides an ambiguous signal: the entire test execution is marked as failing, even though only a subset of the executed code is relevant to the specific assertion that failed. This noise dilutes the correlation between code coverage and actual faults, reducing the accuracy of suspiciousness rankings. Test case purification addresses this

limitation by refining test cases to produce cleaner, more precise coverage spectra that better isolate fault-relevant code.

3 Test Case Purification

Test case purification [29] is a technique that refines test cases to improve the quality of coverage spectra used in fault localization. The core insight is that a failing test with multiple assertions provides an ambiguous signal: we know the test failed, but not necessarily which assertion failed or which code is relevant to that specific failure. By purifying tests into focused, single-assertion variants, we can obtain more precise spectra that better correlate with actual faults. This approach not only clarifies which code is responsible for a failure, but also helps reduce noise in the suspiciousness scores produced by fault localization tools. In practice, purification can reveal subtle bugs that would otherwise be masked by unrelated assertions or setup code, making it a valuable preprocessing step for automated debugging.

3.1 Motivating Example

Consider a simple calculator module with a bug in the multiplication function shown in Figure 1. The original test checks multiple operations, but when it fails, traditional SFL sees that all lines were executed before the failure. The suspiciousness score for `add` and `multiply` will be similar because the failing test covered both functions. Moreover, the setup code (`x = 2`, `y = 3`, `sum_result = ...`) appears just as suspicious as the actual bug. After purification, we obtain one single-assertion test case. Figure 1c shows the purified test for the multiplication assertion. The spectrum now clearly shows that only `multiply` is relevant to this failure.

```python
def add(a, b):
    return a + b

def multiply(a, b):
    return a * a  # Bug
```

(a) Program under test

```python
def test_calculator():
    x = 2
    y = 3
    sum_result = add(x, y)
    assert sum_result == 5
    prod_result = multiply(x, y)
    assert prod_result == 6
```

(b) Original test

```python
def test_calculator():
    x = 2
    y = 3
    prod_result = multiply(x, y)
    assert prod_result == 6
```

(c) Purified test

Figure 1: Test case purification example. The program (a) contains a bug in the multiplication function. The original test (b) contains multiple assertions and unrelated code. The purified test (c) focuses only on the failing assertion, making the fault more apparent.

3.2 Framework

The test case purification framework consists of three phases. First, atomization splits each failing test with k assertions into k copies, each preserving one assertion and suppressing the others to avoid premature termination. Only failing atomized tests proceed, so the following analysis focuses on relevant failures.

Second, dynamic slicing removes statements that do not influence the failing assertion. Leveraging execution tracing, we keep only statements that define variables used by the broken statement or control whether it executes. The result is a purified test case that retains only the code necessary to reach and evaluate the failing assertion, often leading to much smaller, more focused test cases that are easier to interpret and debug.

Third, rank refinement recomputes rankings using the purified tests. For each program element s, we combine the normalized suspiciousness score from a base SFL technique with the ratio of purified tests that cover s. The final score is computed as $score(s) = norm(s) \times \frac{1+ratio(s)}{2}$, which boosts elements that appear frequently in purified failing tests while penalizing those that rarely appear.

3.3 When to Purify

Developers often group multiple assertions intentionally to validate different facets of one behavior. Test case purification generates derived tests as debugging artifacts for fault localization. Thus, readability and maintainability of the production test suite remain unchanged. However, atomization increases the number of derived tests by at most the number of assertions in each failing test. Hence, we recommend purification selectively, by running it only for failing tests during debugging sessions, while keeping regression runs on the original suite, keeping the workflow lightweight.

4 Implementation

PYURIFY is implemented in Python and addresses several challenges unique to Python's dynamic nature and its testing ecosystem. The tool is compatible with common Python testing frameworks and can be easily integrated into existing development workflows.

4.1 Test Case Atomization

The atomization phase uses Python's abstract syntax tree (AST) to transform test functions. For each test with k assertions, we generate k AST variants where one assertion remains unchanged (the target assertion) and all others are wrapped in exception handlers. The transformed AST is unparsed and written to separate files with systematic names to track the target assertion.

A key challenge is preserving test semantics. Our implementation keeps PYTEST fixtures, parametrization decorators, class-based organization, imports, and setup/teardown methods to ensure the atomized tests maintain the same context as the original.

4.2 Dynamic Slicing

We leverage Python's `settrace` to collect execution traces. During execution, we record data dependencies (which statement defines each variable), control dependencies (which statements control execution flow), and the execution order. This approach allows us to accurately identify the minimal set of statements required to reproduce the failure, even in the presence of complex control flow or data dependencies. Tracing and dependency tracking add runtime overhead proportional to executed statements; therefore, PYURIFY is designed to run on failing tests as a diagnostic step rather than on every test in every pipeline stage. We implement a backward slicing algorithm that starts with the failing statement and its variables, then recursively includes all statements that define variable usages or control the path to the target.

4.3 Purified Test Generation

After slicing, we reconstruct purified tests from the atomized test AST. We ensure that the purified test remains syntactically valid and semantically equivalent for the failing assertion. The purified tests are written to files, named to indicate their origin and the specific assertion they target. A generated file contains only a single purified test (without other interfering test functions) while maintaining all necessary contexts.

4.4 Integration with SFL Tools

PYURIFY integrates with existing fault-localization tools by producing purified test files that PYTEST can execute. The tool outputs a mapping from original test identifiers to purified test files, which can be fed into SFL frameworks like SFLKIT [25] to recompute suspiciousness scores based on the refined spectra.

5 Usage

PYURIFY is designed for ease of use and provides both a command-line interface for interactive debugging and a Python API for programmatic integration.

5.1 Installation

The tool is distributed via PyPI, and installation is a single command:

```
pip install pyurify
```

5.2 Command-Line Interface

The primary command-line tool is `pyurify`, which accepts several parameters. Leveraging it, we can purify tests in a specified source directory and output them to a destination directory, for example, as follows:

```
pyurify -s tests/ -d purified/ \
    -f "test_math.py::test_calculator"
```

5.3 Python API

For integration into automated workflows or fault localization pipelines, PYURIFY provides a Python API:

```
from pyurify import purify_tests

result = purify_tests(
    src_dir=Path("tests"),
    dst_dir=Path("purified"),
    failing_tests=["test_math.py::test_calculator"],
    enable_slicing=True,
)
```

Table 1: SFL effectiveness of test case purification. For top-n, higher is better; for EXAM and Wasted Effort, lower is better.

Formula	Metric	Avg	TCP Avg	Δ	Impr.
TARANTULA	top-1	6.28%	7.03%	+0.75%	+11.9%
	top-5	15.24%	15.98%	+0.74%	+4.9%
	top-10	17.74%	17.15%	-0.59%	-3.3%
	EXAM	0.214	0.204	-0.01	+4.6%
	Wasted	912.7	892.6	-20.1	+2.2%
OCHIAI	top-1	6.85%	7.08%	+0.23%	+3.4%
	top-5	15.16%	14.20%	-0.96%	-6.3%
	top-10	17.06%	16.31%	-0.75%	-4.4%
	EXAM	0.213	0.206	-0.007	+3.1%
	Wasted	909.3	894.3	-15.0	+1.6%

The API returns a dictionary mapping original test identifiers to lists of `(purified_file, param_suffix)` tuples. Each tuple contains the path to a purified test file and an optional parameter-suffix for parameterized tests, enabling downstream processing and integration with fault-localization tools.

6 Evaluation

We evaluate PYURIFY on real bugs from popular open-source Python projects to answer the following research question:

- **RQ1:** Does test case purification improve the effectiveness of statistical fault localization on Python programs?
- **RQ2:** What is the practical relevance of test case purification?

6.1 Experimental Setup

We evaluate PYURIFY on bugs from three widely-used projects: FASTAPI (a web framework for building APIs, 16 bugs), HTTPie (a command-line HTTP client, 5 bugs), and TQDM (a progress bar library, 9 bugs). These projects span diverse application domains and contain real bugs that are exposed by existing test suites. The bugs are obtained from TESTS4PY [24], a benchmark for Python program analysis research.

We evaluate two widely-used SFL formulas, TARANTULA [8] and OCHIAI [1], implemented using SFLKIT [25], a framework for spectrum-based fault localization. We measure fault localization effectiveness using three metrics: top-n (percentage of bugs where the fault is in the top-n ranked elements), EXAM Score (percentage of code that must be examined before finding the fault), and Wasted Effort (number of statements examined before finding the fault).

6.2 Results

Table 1 presents the results comparing baseline SFL (*Avg*) with test case purification (*TCP Avg*) for RQ1. We focus on average-case results as they best represent realistic debugging scenarios.

Test case purification improves top-1 accuracy for both formulas. For TARANTULA, top-1 increases from 6.28% to 7.03% (11.9% relative improvement), and for OCHIAI, it increases from 6.85% to 7.08%

(3.4% relative improvement), meaning developers are more likely to find the fault by examining the highest-ranked element first. Both formulas also show reduced EXAM scores with purification: TARANTULA's EXAM score decreases from 0.214 to 0.204 (4.6% less code to examine on average), and OCHIAI's decreases from 0.2129 to 0.2063 (3.1% reduction). Similarly, purification reduces wasted effort, with TARANTULA reducing from 912.7 to 892.6 statements (20.1 fewer) and OCHIAI reducing from 909.3 to 894.3 statements (15.0 fewer). While top-5 and top-10 metrics show mixed results, the improvements in top-1 and EXAM are more valuable in practice, as developers typically start by examining the highest-ranked elements. The improvements demonstrate that purified tests create cleaner coverage spectra that better isolate fault-related code from incidental test execution, enabling SFL formulas to rank faulty elements more accurately.

6.3 Relevance and AI-Assisted Debugging

To answer RQ2, we consider the practical relevance of test case purification in the current LLM-assisted debugging landscape. PYURIFY is intended to complement, not replace, modern LLM-assisted debugging. LLMs can suggest likely fixes from code and test output. At the same time, purification provides a deterministic preprocessing step that reduces irrelevant execution context before ranking suspicious elements, particularly useful in settings where reproducibility and auditability are required (e.g., CI pipelines), or where large repositories and long test traces challenge LLM context limits. Our results indicate modest but consistent gains in top-1 and effort-based metrics, suggesting that purification is most valuable when developers want a reliable first-ranked location to inspect quickly.

7 Threats to Validity

Our implementation may contain bugs; we mitigate this through extensive unit testing that covers atomization and slicing, as well as manual inspection of purified tests. Our evaluation is limited to 30 bugs across 3 Python projects and 2 widely used SFL formulas (TARANTULA and OCHIAI), which may not generalize to all codebases or techniques. In particular, broader benchmarks such as HaPy-Bug [20] are not yet included and may reveal additional strengths or limitations. Dynamic tracing and data-flow analysis introduce computational overhead; while we position PYURIFY as an on-demand debugging step, a larger-scale runtime study is still needed. The standard metrics (top-n, EXAM, Wasted Effort) enable comparison with prior work, building confidence in our findings.

8 Related Work

Statistical fault localization techniques rank program elements by their correlation with test failures and play a crucial role in automated repair [17, 21]. Early approaches like TARANTULA [8] and OCHIAI [1] use coverage-based suspiciousness formulas. Numerous variants and improvements have been proposed [5, 10, 19], with different similarity coefficients and spectrum types (lines, blocks, branches, paths [11, 34]).

Beyond coverage, richer program features for localization have been explored. Data-flow features like definition-use pairs capture state propagation [18, 22, 33]. Predicate-based approaches [15] track branch outcomes and variable values. Call-graph features and execution patterns have also been investigated [26]. A recent study on execution-feature-driven debugging [23] provides a comprehensive comparison of these various types.

Several techniques improve fault localization by modifying or augmenting test suites. Test prioritization and reduction select subsets of tests that maximize diagnostic information while reducing execution cost [4, 6, 7, 32]. Test generation techniques create new tests that increase coverage diversity or resemble failing tests [2].

Test case purification [29], which our tool implements, transforms tests to make them more focused. The original work by Xuan and Monperrus demonstrated the effectiveness of purification on Java programs using the DEFECTS4J benchmark. Our work extends this concept to Python, addressing unique challenges of Python's dynamic typing, PYTEST framework, and test transformation.

Recent work applies machine learning, including large language models, to fault localization [9, 12–14, 16, 27, 28, 30, 31]. These approaches complement test purification, as purified tests can improve the training data for learning-based techniques.

9 Conclusion and Future Work

We have presented PYURIFY, a practical tool for test case purification in Python. PYURIFY implements test case atomization to split multi-assertion tests into single-assertion variants, and dynamic slicing to remove irrelevant code. This results in purified tests that yield cleaner coverage spectra, enabling more accurate fault localization. Our evaluation of 30 real bugs demonstrates that purification improves key fault localization metrics.

Future work includes larger-scale evaluation on additional benchmarks, systematic measurement of runtime overhead, and tighter integration with LLM-assisted debugging workflows.

Data and Tool Availability

PYURIFY, including source code and evaluation scripts, is open-source under the Apache License 2.0 and available at:

https://github.com/smythi93/pyurify

Acknowledgments

This research was partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) ZE 509/7–2 (261444241) and by the European Union (ERC S3, 101093186). Views and opinions expressed are, however, those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC '06)*. IEEE Computer Society, USA, 39–46. https://doi.org/10.1109/PRDC.2006.18

[2] Shay Artzi, Julian Dolby, Frank Tip, and Marco Pistoia. 2010. Directed test generation for effective fault localization. In *Proceedings of the 19th International Symposium on Software Testing and Analysis* (Trento, Italy) *(ISSTA '10)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/1831708.1831715

[3] Mike Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: problem determination in large, dynamic Internet services.

In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. 595–604. https://doi.org/10.1109/DSN.2002.1029005

[4] Gong Dandan, Wang Tiantian, Su Xiaohong, and Ma Peijun. 2013. A Test-suite Reduction Approach to Improving Fault-localization Effectiveness. *Comput. Lang. Syst. Struct.* 39, 3 (Oct. 2013), 95–108. https://doi.org/10.1016/j.cl.2013.04.001

[5] Patrick Daniel and Kwan Yong Sim. 2013. Spectrum-based Fault Localization: A Pair Scoring Approach. *Journal of Industrial and Intelligent Information* 1 (2013), 185–190. https://doi.org/10.12720/jiii.1.4.185-190

[6] Alberto González-Sanchez, Rui Abreu, Hans-Gerhard Gross, and Arjan J. C. van Gemund. 2011. Prioritizing tests for fault localization through ambiguity group reduction. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE Computer Society, 83–92. https://doi.org/10.1109/ASE.2011.6100153

[7] Bo Jiang, Zhenyu Zhang, W. K. Chan, T. H. Tse, and Tsong Yueh Chen. 2012. How Well Does Test Case Prioritization Integrate with Statistical Fault Localization? *Inf. Softw. Technol.* 54, 7 (July 2012), 739–758. https://doi.org/10.1016/j.infsof.2012.01.006

[8] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering* (Orlando, Florida). New York, NY, USA, 467–477. https://doi.org/10.1145/581339.581397

[9] Sungmin Kang, Gabin An, and Shin Yoo. 2024. A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization. *Proc. ACM Softw. Eng.* 1, FSE, Article 64 (jul 2024), 23 pages. https://doi.org/10.1145/3660771

[10] David Landsberg, Hana Chockler, Daniel Kroening, and Matt Lewis. 2015. Evaluation of Measures for Statistical Fault Localisation and an Optimising Scheme. In *Fundamental Approaches to Software Engineering*, Alexander Egyed and Ina Schaefer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 115–129. https://doi.org/10.1007/978-3-662-46675-9_8

[11] Wei Le and Mary Lou Soffa. 2010. Path-based fault correlations. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Santa Fe, New Mexico, USA) *(FSE '10)*. Association for Computing Machinery, New York, NY, USA, 307–316. https://doi.org/10.1145/1882291.1882336

[12] Yan Lei, Huan Xie, Tao Zhang, Meng Yan, Zhou Xu, and Chengnian Sun. 2022. Feature-FL: Feature-Based Fault Localization. *IEEE Transactions on Reliability* 71, 1 (2022), 264–283. https://doi.org/10.1109/TR.2022.3140453

[13] Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 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* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 169–180. https://doi.org/10.1145/3293882.3330574

[14] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 661–673. https://doi.org/10.1109/ICSE43902.2021.00067

[15] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable Statistical Bug Isolation. *SIGPLAN Not.* 40, 6 (jun 2005), 15–26. https://doi.org/10.1145/1064978.1065014

[16] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 664–676. https://doi.org/10.1145/3468264.3468580

[17] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 101–114. https://doi.org/10.1145/3395363.3397369

[18] Wes Masri. 2010. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability* 20, 2 (2010), 121–147. https://doi.org/10.1002/stvr.409 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.409

[19] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-Based Software Diagnosis. *ACM Trans. Softw. Eng. Methodol.* 20, 3, Article 11 (aug 2011), 32 pages. https://doi.org/10.1145/2000791.2000795

[20] Piotr Przymus, Mikołaj Fejzer, Jakub Narębski, Radosław Woźniak, Łukasz Halada, Aleksander Kazecki, Mykhailo Molchanov, and Krzysztof Stencel. 2025. HaPy-Bug – Human Annotated Python Bug Resolution Dataset. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*. 81–85. https://doi.org/10.1109/MSR66628.2025.00024

[21] Yuhua Qi, Xiaoguang Mao, Yan Lei, and Chengsong Wang. 2013. Using Automated Program Repair for Evaluating the Effectiveness of Fault Localization Techniques. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis* (Lugano, Switzerland) *(ISSTA 2013)*. Association for Computing Machinery, New York, NY, USA, 191–201. https://doi.org/10.1145/2483760.2483785

[22] Henrique L. Ribeiro, P. A. Roberto de Araujo, Marcos L. Chaim, Higor A. de Souza, and Fabio Kon. 2019. Evaluating data-flow coverage in spectrum-based fault localization. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–11. https://doi.org/10.1109/ESEM.2019.8870182

[23] Marius Smytzek, Martin Eberlein, Lars Grunske, and Andreas Zeller. 2025. How Execution Features Relate to Failures: An Empirical Study and Diagnosis Approach. *ACM Trans. Softw. Eng. Methodol.* (Dec. 2025). https://doi.org/10.1145/3783989 Just Accepted.

[24] Marius Smytzek, Martin Eberlein, Batuhan Serçe, Lars Grunske, and Andreas Zeller. 2024. Tests4Py: A Benchmark for System Testing. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 557–561. https://doi.org/10.1145/3663529.3663798

[25] Marius Smytzek and Andreas Zeller. 2022. SFLKit: A workbench for statistical fault localization. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1701–1705. https://doi.org/10.1145/3540250.3558915

[26] Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Árpád Beszédes. 2021. Call Frequency-Based Fault Localization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 365–376. https://doi.org/10.1109/SANER50967.2021.00041

[27] Xu Wang, Hongwei Yu, Xiangxin Meng, Hongliang Cao, Hongyu Zhang, Hailong Sun, Xudong Liu, and Chunming Hu. 2024. MTL-TRANSFER: Leveraging Multi-task Learning and Transferred Knowledge for Improving Fault Localization and Program Repair. *ACM Trans. Softw. Eng. Methodol.* 33, 6, Article 148 (jun 2024), 31 pages. https://doi.org/10.1145/3654441

[28] Ratnadira Widyasari, Gede Artha Azriadi Prana, Stefanus A. Haryono, Yuan Tian, Hafil Noer Zachiary, and David Lo. 2022. XAI4FL: enhancing spectrum-based fault localization with explainable artificial intelligence. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 499–510. https://doi.org/10.1145/3524610.3527902

[29] Jifeng Xuan and Martin Monperrus. 2014. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey (Eds.). ACM, 52–63. https://doi.org/10.1145/2635868.2635906

[30] Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large Language Models for Test-Free Fault Localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/3597503.3623342

[31] Haoran Yang, Yu Nong, Tao Zhang, Xiapu Luo, and Haipeng Cai. 2024. Learning to Detect and Localize Multilingual Bugs. *Proc. ACM Softw. Eng.* 1, FSE, Article 97 (jul 2024), 24 pages. https://doi.org/10.1145/3660804

[32] Shin Yoo, Mark Harman, and David Clark. 2013. Fault Localization Prioritization: Comparing Information-Theoretic and Coverage-Based Approaches. *ACM Trans. Softw. Eng. Methodol.* 22, 3, Article 19 (jul 2013), 29 pages. https://doi.org/10.1145/2491509.2491513

[33] Kai Yu, Mengxiang Lin, Qing Gao, Hui Zhang, and Xiangyu Zhang. 2011. Locating faults using multiple spectra-specific models. In *Proceedings of the 2011 ACM Symposium on Applied Computing* (TaiChung, Taiwan) *(SAC '11)*. Association for Computing Machinery, New York, NY, USA, 1404–1410. https://doi.org/10.1145/1982185.1982490

[34] Zhenyu Zhang, W. K. Chan, T. H. Tse, Bo Jiang, and Xinming Wang. 2009. Capturing propagation of infected program states. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Amsterdam, The Netherlands) *(ESEC/FSE '09)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/1595696.1595705