SFLKit: A Workbench for Statistical Fault Localization

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

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

ABSTRACT

Statistical fault localization aims at detecting execution features that correlate with failures, such as whether individual lines are part of the execution. We introduce SFLKit, an out-of-the-box workbench for statistical fault localization. The framework provides straightforward access to the fundamental concepts of statistical fault localization. It supports five predicate types, four coverage-inspired spectra, like lines, and 38 similarity coefficients, e.g., TARANTULA or OCHIAI, for statistical program analysis.

SFLKit separates the execution of tests from the analysis of the results and is therefore independent of the used testing framework. It leverages program instrumentation to enable the logging of events and derives the predicates and spectra from these logs. This instrumentation allows for introducing multiple programming languages and the extension of new concepts in statistical fault localization. Currently, SFLKit supports the instrumentation of python programs. SFLKit is highly configurable, requiring only the logging of the required events.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; *Software libraries and repositories.*

KEYWORDS

statistical fault localization, statistical debugging, spectrum-based fault localization, similarity coefficient

ACM Reference Format:
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 (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3540250.3558915

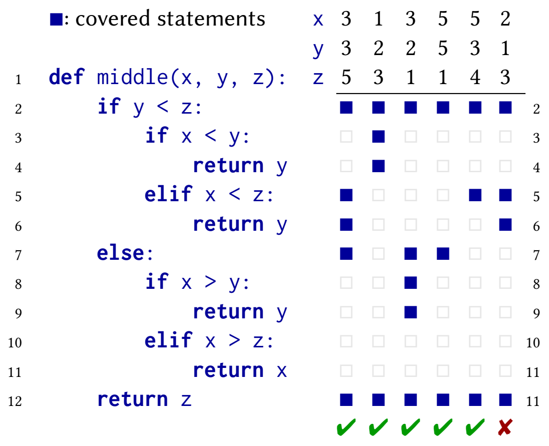
1 INTRODUCTION

The basic concept of *statistical fault localization* is easy to explain. Suppose a particular line in the program is executed primarily on failing runs. In that case, its execution correlates with a failure and gives essential hints on the failure causes, notably a possible location. The seminal TARANTULA paper [7] uses a `middle()` function to illustrate the concept, shown in Figure 1. `middle(x, y, z)` is supposed to return the "middle" of the three values `x`, `y`, `z`—the value that is neither the minimum, nor the maximum of the three: `middle(5, 3, 4)`, for instance, correctly returns 4.

However, the `middle()` code in Figure 1 is faulty, as `middle(2, 1, 3)` returns 1 rather than 2. By feeding `middle()` various values and observing the executed lines, we can determine that the execution of Line 6 has the strongest correlation with failure. Line 6 also happens to be the fault location—it should return `x` rather than `y`.

What works well in a small example does not necessarily scale to large systems. On a large scale, statistical fault localization can suffer from reporting a *multitude of potentially suspicious lines* [14, 15]. Various statistical approaches have been suggested over the past two decades, each improving over the previous benchmark results, to make fault localization more precise [14]. However, each of these approaches introduces their unique infrastructure that with custom debugging loggers, custom predicates and custom spectra [2, 6, 8]. Yet, for research, we'd like to be able to *select* and *combine* these elements in a modular, reusable fashion.

This paper introduces SFLKit, a modular statistical fault localization framework that provides easy-to-use essential concepts of statistical debugging and spectrum-based fault localization. SFLKit



```
        ■: covered statements     x  3  1  3  5  5  2
                                  y  3  2  2  5  3  1
 1  def middle(x, y, z):          z  5  3  1  1  4  3
 2      if y < z:                    ■  ■  ■  ■  ■  ■   2
 3          if x < y:                □  ■  □  □  □  □   3
 4              return y             □  ■  □  □  □  □   4
 5          elif x < z:              ■  □  □  □  ■  ■   5
 6              return y             ■  □  □  □  □  ■   6
 7      else:                        ■  □  ■  ■  □  □   7
 8          if x > y:                □  □  ■  □  □  □   8
 9              return y             □  □  ■  □  □  □   9
10          elif x > z:              □  □  □  □  □  □  10
11              return x             □  □  □  □  □  □  11
12      return z                     ■  ■  ■  ■  ■  ■  11
                                     ✓  ✓  ✓  ✓  ✓  ✗
```

Figure 1: Statistical fault localization [7]. The `middle()` function takes three values and returns the one that is neither the minimum nor the maximum; Line 6 most strongly correlates with failure.

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9413-0/22/11…$15.00
https://doi.org/10.1145/3540250.3558915

is highly configurable and effortless to extend with new approaches, making it a great base for future research and education.

2 BACKGROUND

2.1 Spectrum-Based Fault Localization

Spectrum-Based Fault Localization (*SBFL*) [7] is a technique to localize the code location of a fault, founded on so-called program spectra, which describe a program's execution. Typically such an approach considers a faulty program and a test set comprising at least one failing test. This approach then collects coverage information for each test case as the spectrum and correlates each code element, described by the coverage, with the failure by calculating a suspiciousness score. This score ranks how likely a specific code location is faulty. The general idea behind *SBFL* is that elements often executed in failing test cases and not so often in passing ones are more likely to be faulty. In recent years, this technique became more popular as part of *automated program repair* and was the subject of some studies in this context [16].

In practice, *SBFL* techniques consider *executed lines* and use a similarity coefficient to compute the suspiciousness. Different approaches vary only in the choice of this similarity coefficient. This research trend results in an uncountable number of possible coefficients [5, 9, 12]. TARANTULA [7, 8] was one of the first metrics introduced to *SBFL* and is still a popular choice.

SFLKit implements four spectra and 38 similarity coefficients that a user can leverage (see Table 1).

2.2 Statistical Debugging

In contrast to *SBFL*, statistical debugging (*SD*) leverages predicates on the code that can be *true*, *false*, or not executed. Depending on the result of these predicates, *SD* allows for deriving properties that need to hold for producing the fault. While *SBFL* relies on similarity coefficients to measure the suspiciousness of an element, *SD* relies on the standardized metrics *Failure*, *Context*, and *Increase* [11].

Failure is the probability a failure occurs under the condition that a predicate evaluates to *true*.

Context is the probability that a failure occurs when the predicate is observed.

Increase describes how much the predicate being *true* increases the probability of a failure occurring. The *Increase* of a predicate p is calculated as

$$\mathit{Increase}(p) = \mathit{Failure}(p) - \mathit{Context}(p)$$

The most common example of *SD* is to leverage the condition of branches, which was introduced by Liblit et al. [11].

SFLKit supports five types of predicates that cover various program properties, allowing for tracking different program behaviors that could lead to a fault (see Table 1).

3 DESIGN AND IMPLEMENTATION

We designed our framework such that we can separate it into two stages. The first is the collecting of execution features. This stage represents a debugging logger where we collect events during the program's execution to capture its behavior. It is entirely independent of statistical fault localization and allows for using our framework as a debugging logger. We describe this phase in Section 3.1. The second stage is analyzing the collected features with concepts of statistical fault localization. We explain this phase in Section 3.2.

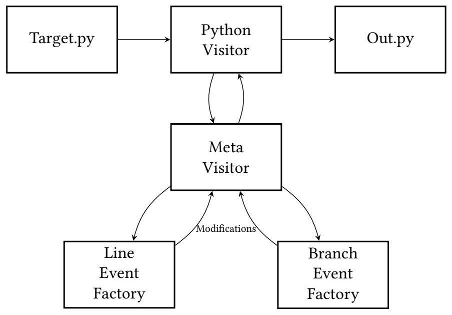


Figure 2: A description of the meta-level instrumentation used in our statistical fault localization framework.

3.1 Collecting Execution Features

The base of our statistical fault localization framework is an instrumentation of the source code that enables the logging of specified events.

This concept leverages the abstract syntax tree (*AST*) of the source code and an interface enabling the visiting of a node of the *AST* on a meta-level. Each event we use comes with a factory to produce it. The factory represents the visitor on the meta-level and only produces the instrumentation for the single event it represents.

For example, we implemented a line event that gets triggered every time a line (or a statement in other programming languages) occurs. The overall visitor, in this case for Python, calls every meta-visitor for each node. The event factory knows the injections and modifications of the code for each node. With this design, we can extend our framework with new programming languages and events while the events are independent.

Our framework accomplishes the instrumentation in three steps on the *AST*.

(1) Parse the source code to an abstract syntax tree (*AST*). Visit each node of the *AST*. We leverage the *ast* module coming with Python for our Python support.
(2) Call the corresponding meta-level visitor on the AST node that knows the statements it needs to inject.
(3) Combine all injections and changes to modify the source code.

The implementation of our instrumentation allows the extension with other programming languages without modifying the core functionalities by implementing the event factories for a new programming language and an AST visitor visiting all nodes and combining the results of these factories.

Figure 2 provides an overview of the instrumentation process in our statistical fault localization framework.

Our current implementation supports solely *Python* programs because we did our experiments and evaluation in this programming language.

Table 1: The implemented concepts of statistical fault localization. The first column describes the type of the concept, the second the number of implemented concepts of this type, and the last column lists the concepts.

Type	Number	Concepts
Events	11	LineEvent, BranchEvent, DefEvent, UseEvent, FunctionEnterEvent, FunctionExitEvent, FunctionErrorEvent, LoopBeginEvent, LoopHitEvent, LoopEndEvent, ConditionEvent
Spectra	4	Lines[7], Functions, DefUse Pairs[17], Loops[13]
Similarity Coefficients [5, 9, 12]	38	AMPLE[4], AMPLE2[4], Anderberg, ArithmeticMean, Binary, CBIInc, Cohen, Crosstab[21], Dice, DStar[20], Euclid, Fleiss, GP02[23], GP03[23], GP13[23], GP19[23], Goodman, Hamann, HammingEtc, HarmonicMean, Jaccard[3], Kulczynski1, Kulczynski2, M1, M2, Naish1[10], Naish2[10], Ochiai[1], Ochiai2[1], PairScoring, qe, RogersAndTanimoto, Rogot1, Rogot2, RusselAndRao, Scott, SimpleMatching, Sokal, SorensenDice, Tarantula[7], Wong1[22], Wong2[22], Wong3[22], Zoltar
Predicates	5	Branch[11], Scalar Pairs[11], Variable Predicates, Return Predicates[11], Condition

The core of our statistical fault localization framework are events collected during the execution of the program under test. In detail, a unit test is suitable for extracting the execution events with our framework. These events inject their required statements into the target program as part of the instrumentation.

Our instrumentation can collect the execution events during a test run and put them aside for later analysis. The critical component here is a shared library that is part of the instrumentation. When the program logs an event, it tells the shared library to write the determined event into the log. The analysis can then read the log and construct the needed data from this log. This separation allows for using our framework as an execution event logger.

Table 1 provides an overview of the implemented events.

3.2 Analyzing Execution Features

To analyze the execution features, we leverage the logged events to rebuild a model that we can modify while replaying the events.

This model is a practical way of keeping track of the program's behavior during the execution and enables our framework to derive the spectra and predicates used for future analysis.

For each run, for which a log exists, SFLKit builds a model and checks what spectra and predicates are observed and, in addition, how they evaluate, i.e., for predicates to *true* or *false*. It leverages these observations to calculate a variety of suspiciousness scores defined by the user. The suspiciousness scores provide an overview of how likely an analysis object correlates with the fault to investigate.

We implemented many previously introduced analysis objects like lines, branches, or def-use pairs. As a novel aspect, we introduce a new predicate we are not aware already exists to the best of our knowledge. We based this novel analysis subject on existing coverage criteria.

Conditions We introduce condition predicates derived from condition coverage. For this predicate, we log sub-condition events that occur whenever a sub-condition occurs, e.g., all terms of the test of an if-statement. The predicate checks whether a sub-condition being true (or false) correlates with the program's failure.

After extracting all analysis objects, we evaluate them with the defined similarity coefficients or, in the case of predicates, with the *Increase* metric as explained in Section 2.2.

Table 1 provides an overview of the supported program spectra, predicates, and similarity coefficients.

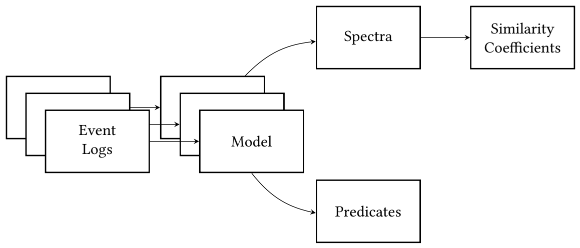


Figure 3: An overview of the analysis stage of our statistical fault localization framework.

4 TOOL USAGE

We provide our workbench as a Python package that is installable by running `pip install .` inside the root directory of the framework. After installing the package, the various concepts are accessible by importing `sflkit` in a Python script. The easy-to-use access points to SFLKit are `sflkit.instrument(config)` and `sflkit.analyze(config)` that take the path to a config file and execute the instrumentation and the analysis of the execution events.

Consider the example in Figure 1. Figure 4 shows an example of a config file for this scenario. `sflkit.instrument(config)` leverages the predicates to investigate, extract the needed events, and then instruments the target path as described in Section 3.1 to the instrumentation path. `sflkit.analyze(config)` parses the events from passing and failing parameters, builds a model for each run, and then analysis the executed lines and computes the in metrics provided OCHIAI [1, 2], JACCARD [3], and TARANTULA [7, 8] on these lines.

All these steps allow for an individual execution and can be interfered with at any time to leverage intermediate results.

In addition, our tool provides a command-line interface that leverages these configuration files. To execute the command-line interface, a user needs to run `python sfl.py instrument -c config` and `python sfl.py analyze -c config` where `config` is the path to the config file.

5 EXPERIMENTAL EVALUATION

We investigated bugs from the BugsInPy [18, 19] benchmark to test SFLKit under authentic conditions. This collection of real-world faulty Python programs comprises 501 subjects with passing and

```
[target]
path = middle.py
language = Python

[events]
predicates = Line
metrics = Ochiai,Tarantula,Jaccard
passing = events/passing
failing = events/failing

[instrumentation]
path = tmp.py
```

Figure 4: An example of a config file for SFLKit

failing tests that we can leverage. In addition, the benchmark provides a patch file for each bug that we can use to extract the code locations that contribute to the fau

BugsInPy provides the test file for each bug that contains the tests that trigger it. We used only the test cases in the provided file because these are more likely to relate to the fault. In addition, using only these tests instead of all provided test files significantly speeds up our evaluation.

We selected three bugs for each project included in BugsInPy to investigate, if possible. Several programs have no failing tests. Even the tests that should trigger the bug did not fail, so we skipped them for our selection. Moreover, some applications have no passing tests, which we excluded too because there would be no logical explanation for a particular code location to be faulty. In addition, we did not investigate matplotlib/matplotlib and pandas-dev/pandas because both come with a tremendous number of test cases that take too long to run in our cross-validation setup for which we needed to execute each subject multiple times. Moreover, we needed to exclude explosion/spaCy, since there were no suitable subjects as specified by our criteria. However, we want to investigate the subjects we skipped in further evaluating our statistical fault localization framework.

Besides this subject selection, we could not leverage each provided test in our evaluation. We excluded such test cases that changed the outcome because of our instrumentation. We noticed two reasons for this. The first is that some of the tests verified the integrity of the source code, i.e., it was not altered, which we did. The other reason is that a few tests ran into performance issues when instrumented. These performance issues are something we want to investigate further.

Table 2 provides an overview of all the bugs and tests we leveraged to test our framework. We investigated 41 subjects from 15 of BugsInPy's 17 projects with a combined number of 1,823 test cases. Moreover, the chosen subjects demonstrate the scalability of SFLKit, with subjects comprising up to 68,801 lines of code.

We investigated lines as an analysis target for these real-world experimentations. We calculated the precision, recall, precision@k, and recall@k on the suggestions the analysis returns concerning the actual faulty lines. We leveraged the OCHIAI [1, 2], the JACCARD [3], and the TARANTULA [7, 8]similarity coefficients for the experimental evaluation. Table 3 demonstrates the results of this evaluation.

Table 2: The subjects of BugsInPy we investigated. Bugs, #Bugs, #Tests, #lines donate to the number of bugs per subject, the bugs we investigated, the number of tests we leveraged, and the average lines of code analyzed per bug, respectively.

Project	Bugs	#Bugs	#Tests	#Lines
cool-RR/PySnooper	3	1	5	216
ansible/ansible	18	3	245	53,651
psf/black	23	3	341	145
cookiecutter/cookiecutter	4	3	26	997
tiangolo/fastapi	16	3	20	4,268
jakubroztocil/httpie	5	3	37	2,128
keras-team/keras	45	3	199	21,747
spotify/luigi	33	3	211	14,894
huge-success/sanic	5	3	158	4,023
scrapy/scrapy	40	3	53	10,962
nvbn/thefuck	32	3	65	3,953
tornadoweb/tornado	16	3	146	23,868
tqdm/tqdm	9	3	153	3,744
ytdl-org/youtube-dl	43	3	164	68,801
Total	501	41	1,823	639,772

Table 3: The results of the experimental evaluation. p@k donates to precision@k and r@k to recall@k, respectively. p@1 and r@1 are equivalent to precision and recall.

Coefficient	k=1		k=3		k=5		k=10	
	p@1	r@1	p@3	r@3	p@5	r@5	p@10	r@10
OCHIAI	0.22	0.13	0.21	0.14	0.18	0.16	0.12	0.17
JACCARD	0.22	0.13	0.22	0.14	0.18	0.16	0.12	0.17
TARANTULA	0.27	0.15	0.25	0.16	0.22	0.18	0.14	0.19
Average	0.237	0.139	0.228	0.149	0.194	0.163	0.130	0.178

SFLKit produced meaningful results for the investigated subjects, with an average precision of 0.237 and a recall of 0.139, aligning with similar experiments, e.g., Jiang et al. [6].

6 CONCLUSION

In this paper, we presented SFLKit, a statistical fault localization framework that is accessible and out-of-the-box usable for real-world applications. The framework provides all common key components of statistical debugging and spectrum-based fault localization.

SFLKit leverages extendable program instrumentation to log execution features as occurring events and derives a model from these features. Leveraging this model, our framework provides various analyses for statistically correlating these execution features with failures.

SFLKit opens the door for uniform and comparable future research in the area of statistical fault localization. Our framework is available as open source at

https://github.com/uds-se/sflkit

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] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 89–98. https://doi.org/10.1109/TAIC.PART.2007.13

[3] Mike Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. 2002. Pinpoint: problem determination in large, dynamic Internet services. *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 595–604. https://doi.org/10.1109/DSN.2002.1029005

[4] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. 2005. Lightweight Bug Localization with AMPLE. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging* (Monterey, California, USA) *(AADEBUG'05)*. Association for Computing Machinery, New York, NY, USA, 99–104. https://doi.org/10.1145/1085130.1085143

[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] Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. 2019. Combining Spectrum-Based Fault Localization and Statistical Debugging: An Empirical Study. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 502–514. https://doi.org/10.1109/ASE.2019.00054

[7] J.A. Jones, M.J. Harrold, and J. Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. 467–477. https://doi.org/10.1145/581396.581397

[8] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the TARANTULA Automatic Fault-Localization Technique. 273–282. https://doi.org/10.1145/1101908.1101949

[9] 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

[10] Hua Lee, Lee Naish, and Kotagiri Ramamohanarao. 2009. The Effectiveness of Using Non redundant Test Cases with Program Spectra for Bug Localization. *Computer Science and Information Technology, International Conference on* 0 (08 2009), 127–134. https://doi.org/10.1109/ICCSIT.2009.5234587

[11] 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

[12] 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

[13] Pruthviraj P. 2021. Fine Grained Statistical Debugging for the Identification of Multiple Bugs. *International Journal of Engineering Research* 10, 05 (2021), 7.

[14] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 199–209. https://doi.org/10.1145/2001420.2001445

[15] Chris Parnin and Alessandro Orso. 2021. *Automated Debugging: Past, Present, and Future (ISSTA Impact Paper Award)*. Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/3460319.3472397

[16] 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

[17] Raul Santelices, James A. Jones, Yanbing Yu, and Mary Jean Harrold. 2009. Lightweight Fault-Localization Using Multiple Coverage Types. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, USA, 56–66. https://doi.org/10.1109/ICSE.2009.5070508

[18] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy. https://github.com/soarsmu/BugsInPy/tree/master/projects.

[19] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, Brian Goh, Ferdian Thung, Hong Jin Kang, Thong Hoang, David Lo, and Eng Lieh Ouh. 2020. BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 1556–1560. https://doi.org/10.1145/3368089.3417943

[20] W. Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. 2012. Software Fault Localization Using DStar (D*). In *2012 IEEE Sixth International Conference on Software Security and Reliability*. 21–30. https://doi.org/10.1109/SERE.2012.12

[21] W. Eric Wong, Vidroha Debroy, and Dianxiang Xu. 2012. Towards Better Fault Localization: A Crosstab-Based Statistical Approach. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 3 (2012), 378–396. https://doi.org/10.1109/TSMCC.2011.2118751

[22] W. Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. 2007. Effective Fault Localization using Code Coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, Vol. 1. 449–456. https://doi.org/10.1109/COMPSAC.2007.109

[23] Xiaoyuan Xie, Fei-Ching Kuo, Tsong Yueh Chen, Shin Yoo, and Mark Harman. 2013. Provably Optimal and Human-Competitive Results in SBSE for Spectrum Based Fault Localisation. In *Proceedings of the 5th International Symposium on Search Based Software Engineering - Volume 8084* (St. Petersburg, Russia) *(SSBSE 2013)*. Springer-Verlag, Berlin, Heidelberg, 224–238. https://doi.org/10.1007/978-3-642-39742-4_17