BASHIRI: Learning Failure Oracles from Execution Features

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

Martin Eberlein
Humboldt-Universität zu Berlin
Berlin, Germany
martin.eberlein@hu-berlin.de

Tural Mammadov
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany
tural.mammadov@cispa.de

Lars Grunske
Humboldt-Universität zu Berlin
Berlin, Germany
grunske@hu-berlin.de

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

***Abstract*—Program fixes must preserve passing tests while fixing failing ones. Validating these properties requires *test oracles* that distinguish passing from failing runs.**

We introduce BASHIRI, a tool that learns *failure oracles* from test suites with labeled outcomes using execution features. BASHIRI leverages execution-feature-driven debugging to collect program execution features and trains interpretable models as testing oracles. Our evaluation shows that BASHIRI predicts test outcomes with 95% accuracy, effectively identifying failing runs. BASHIRI is available as an open-source tool at

https://github.com/smythi93/bashiri

A demonstration video is available at

https://youtu.be/D2mJkCtSXtM

***Index Terms*—Testing oracles, automated debugging, execution features, dynamic analysis**

I. Introduction

Many problems in software analysis and testing could be easily solved if we had good specifications. However, in real-world software development, we often lack good specifications and must deal with these imperfections. Software testing and debugging rely heavily on test oracles that determine whether a program execution is correct or faulty. While specifications would provide ideal oracles, they are rarely available in practice, forcing developers to rely on test suites with specific assertions like `assert sqrt(4) == 2`. These test-specific oracles work well for their intended inputs but do not generalize to new test cases. When fixing programs, be it manually or through automated repair, two critical properties must hold: fixes must eliminate all previously failing behaviors while preserving all previously passing behaviors. Validating these properties requires oracles that can reliably distinguish between correct and incorrect executions across a broader range of inputs than just the original test suite.

We introduce BASHIRI, a tool to *automatically derive failure oracles from test suites*.[1] Unlike traditional test-specific oracles, BASHIRI learns general failure patterns as predicates over execution features. It leverages execution-feature-driven debugging (EFDD) [1] to collect program execution features from labeled test runs and trains interpretable decision tree models that can predict outcomes for new, unseen test cases.

BASHIRI addresses three key challenges: (1) learning oracles from limited examples, (2) maintaining oracle effectiveness across program versions during repair, and (3) improving oracle quality when initial training data is insufficient.

In summary, we make the following contributions:

BASHIRI We introduce BASHIRI, a tool to generate failure oracles linking failures to execution features which can *precisely pinpoint failure conditions,* making them suitable oracles for automated or manual software repair.

Continuity We introduce a mapping that allows us to predict test outcomes even if the program has been modified.

Refinement We introduce a refinement loop that allows us to learn from additional test cases, improving the accuracy of the learned oracles.

II. Learning Oracles from Examples

Our tool targets scenarios where developers have a faulty program and tests that expose the fault. The goal is to derive testing oracles from these labeled test runs (BUG or NO-BUG).

Figure 1 outlines our tool: (1) BASHIRI instruments the program to capture execution events. (2) It runs tests and records execution events. (3) It extracts execution features from these events using EFDD [1]. (4) It trains an interpretable decision tree model as an oracle (Section II-A). For new tests, BASHIRI repeats steps 1-3 and classifies runs using the trained model.

A. Training a Machine Learning Model

We train a decision tree classifier using the extracted feature vectors. Decision trees are ideal because: (1) our discrete features suit tree-based classification, (2) they effectively distinguish failing from passing runs through feature combinations, and (3) their interpretable nature enables identifying

[1] *-bashiri* is the Swahili word for *predict*.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00372

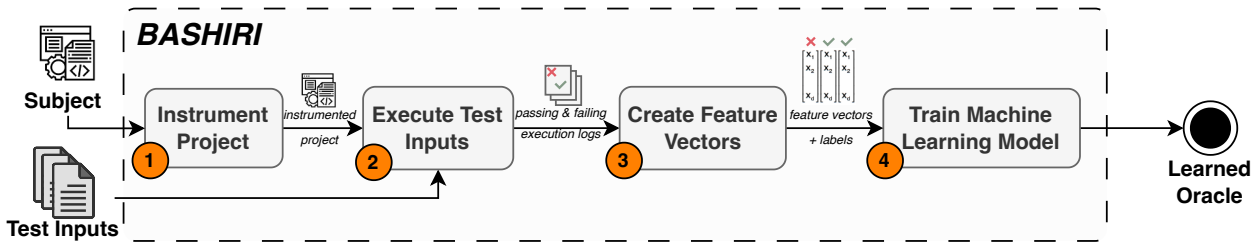

Fig. 1. BASHIRI in a Nutshell. BASHIRI takes a subject and labeled tests as input and leverages execution features to train an oracle.

fault-relevant features. Our experiments with various classifiers (naive, neural networks, large language models) showed that decision trees consistently matched or outperformed alternatives. We maintain classification impartiality, treating passing and failing executions equally. For new test cases, BASHIRI runs them through the instrumented program, extracts feature vectors matching the training structure, and classifies them as BUG or NO-BUG using the decision tree model.

B. Mapping Features

To maintain oracle continuity across program versions, we map execution features between different program versions using a two-step algorithm: (i) First, we identify corresponding events in original and altered programs using patch files to translate line numbers. Events are mapped based on type, file, and line number. (ii) Second, we translate execution traces from the altered program back to the original using the mapped events. This approach allows the same oracle to be applied to modified programs, supporting automated repair workflows.

C. Refinement Loop

Initial test sets may lack comprehensive coverage of execution features that correlate with faults. We extend BASHIRI with a refinement loop that generates additional inputs using fuzzing tools like AFL++ [2] and AFLFAST [3], selects interesting cases for labeling, and retrains the classifier. The process works as follows: After building an initial oracle, BASHIRI generates new inputs and extracts their features, retaining those with novel feature combinations. Inputs exceeding a similarity threshold (based on Jaccard similarity of execution feature sets) are selected as interesting cases, labeled either manually or through automated heuristics, and added to the training set. Finally, the model is retrained with this expanded corpus.

D. Implementation

BASHIRI builds upon EFDD [1] for execution-feature-driven debugging and SFLKIT [4] for statistical fault localization. We use `scikit-learn`'s `DecisionTreeClassifier` [5] and `unidiff` [6] for feature mapping. The refinement loop adopts fuzzing approaches similar to AFL++ and AFLFAST.

E. Usage

BASHIRI first needs an event collector which can be either a `SystemtestEventCollector`, a `UnittestEventCollector`, or a custom event collector. The event collector is responsible for capturing execution features during test runs. The event collector gets the set of passing and failing tests as input. Next, the `EventHandler` builds feature vectors from the collected events. Finally, we can train a BASHIRI by creating a `Bashiri` object and calling its `train()` method. We can then use the `Bashiri` object to classify new test cases by calling its `classify()` method.

III. Experimental Evaluation

For the evaluation of BASHIRI, we want to investigate the following research questions:

RQ1 Oracle Quality. How accurately can BASHIRI from Section II distinguish between passing and failing test cases for an unseen labeled set of runs?

RQ2 Oracle Continuity. How well does BASHIRI from Section II-B predict for altered program versions?

RQ3 Oracle Refinement. How does the refinement loop from Section II-C impact the quality of the learned oracles?

A. Experimental Setup

We asses the quality of BASHIRI using eight subjects from TESTS4PY [7], which extends real-world BUGSINPY [8] defects with targeted test generation.

For RQ1, we train 240 models (30 per subject) with various distributions of passing/failing tests, then evaluate on 100 passing and 100 failing inputs per subject. For RQ2, we apply oracles trained on buggy versions to fixed versions using 10 evaluation tests per subject. For RQ3, we train on minimal input sets (one passing, one failing test) and compare refinement loop improvements.

B. RQ1: Oracle Quality

Table I shows that BASHIRI achieves 95% accuracy across 240 trained models evaluated on 48k unseen tests. The tool demonstrates strong performance with macro-averaged precision of 0.95, recall of 0.94, and F1 score of 0.94. Remarkably, BASHIRI generates perfect oracles (100% accuracy) in 70.83% of cases, indicating that execution features often capture complete failure conditions. The high accuracy achieved even with minimal training data (two test cases, one failing and one passing) can be attributed to execution features precisely capturing the behavioral patterns that distinguish failing from passing runs. When the fault manifests through specific control

TABLE I
QUALITY OF BASHIRI ORACLES FOR RQ1. RESULTS SHOW ACCURACY, PRECISION, RECALL, AND F1 SCORE FOR BUG/NO-BUG LABELS, PLUS PERFECT ORACLE COUNTS AND EXECUTION TIME.

TESTS4PY	BUG	NO-BUG
Precision	0.9219	0.9716
Recall	0.9701	0.9256
F1 Score	0.9454	0.9480
mean *Precision*	0.9219	0.9716
mean *Recall*	0.9603	0.9451
mean *F1 Score*	0.9287	0.9538
Accuracy		94.67%
macro *Precision*		0.9467
macro *Recall*		0.9478
macro *F1 Score*		0.9467
mean macro *Precision*		0.9467
mean macro *Recall*		0.9527
mean macro *F1 Score*		0.9413
# of subjects and configurations		240
# of tests evaluated		48000
# perfect oracles		170
% perfect oracles		70.83%
average execution time per subject		10.59s

TABLE II
CONTINUITY OF BASHIRI ORACLES. THE TABLE SHOWS THE RESULTS FOR RQ2 FOR THE SUBJECTS FROM TESTS4PY. THE TABLE LISTS THE PRECISION, RECALL, AND F1 SCORE FOR THE NO-BUG LABEL.

TESTS4PY	Result
Precision	0.6042
Recall	1.0000
F1 Score	0.7532
Accuracy	60.42%
# of subjects and configurations	240
# of tests evaluated	2400
# perfect oracles	145
% perfect oracles	60.42%

flow patterns or data values, these features become strong predictors even with limited examples. The average execution time remains practical at 11 seconds per learning, making BASHIRI suitable for real-time oracle evaluation.

> BASHIRI *achieves high predictive accuracy (nearly 95%) with 70.83% perfect oracles, but execution features cannot capture all failure conditions.*

C. RQ2: Oracle Continuity

Table II presents the oracle continuity results when applying models trained on buggy versions to their corresponding fixed versions. BASHIRI achieves moderate continuity with an F1 score of 0.75 for the NO-BUG label, which is the expected outcome for all tests on fixed programs. The precision of 0.60 indicates that 40% of predictions incorrectly classify passing tests as failing, while the recall of 1.0 shows that all actual passing tests are correctly identified. This performance variation stems from feature mapping challenges when program

TABLE III
REFINEMENT OF BASHIRI ORACLES. THE TABLE SHOWS THE RESULTS FOR RQ3. IT LISTS THE ACCURACY, PRECISION, RECALL, AND F1 SCORE FOR THE ORIGINAL ORACLES AND REFINED ORACLES.

TESTS4PY	Overall	
	w/o	w/
Accuracy	79%	82%
macro *Precision*	0.79	0.82
macro *Recall*	0.79	0.82
macro *F1 Score*	0.79	0.82
avg time	176s	

modifications alter execution paths significantly. Notably, 60% of models achieve perfect predictions, suggesting that feature mapping works well for minor bug fixes that preserve most execution patterns.

> BASHIRI *demonstrates moderate cross-version oracle continuity with F1 score of 0.75, with 60% of models achieving perfect continuity.*

D. RQ3: Oracle Refinement

Table III demonstrates the effectiveness of the refinement loop in improving oracle quality from limited initial training data. The refinement process increases overall accuracy from 79% to 82%, with corresponding improvements in precision (0.78 to 0.82), recall (0.79 to 0.82), and F1 score (0.78 to 0.82). The fuzzing-based input generation successfully discovers new execution feature combinations that correlate with failure conditions. By iteratively expanding the training corpus with these informative examples, BASHIRI learns more comprehensive failure patterns that generalize better.

> *The refinement loop improves oracle quality, with strong gains for subjects with insufficient initial training data.*

IV. THREATS TO VALIDITY

Internal Validity: Our experiments use TESTS4PY-generated inputs that might be incomplete or easily identifiable by simple features. We mitigated this by ensuring generated inputs produce correct BUG/NO-BUG outcomes and verifying subject compatibility.

External Validity: Our TESTS4PY subjects demonstrate real-world applicability, with SFLKIT previously validated on the entire BUGSINPY benchmark. Currently, BASHIRI works exclusively with PYTHON programs due to its reliance on PYTHON-specific instrumentation capabilities. However, the underlying EFDD execution features are language-agnostic and could be extracted from other programming languages with appropriate instrumentation support. Future work could extend BASHIRI to support additional languages by implementing corresponding instrumentation mechanisms.

Construct Validity: We employed established metrics across all measurements to ensure adequate evaluation of learned oracles' ability to distinguish execution outcomes.

V. Related Work

Learning oracles for failures is an instance of the general *oracle problem* [9] in software testing and debugging. The oracle problem refers to establishing a reliable and accurate criterion to determine the correctness of a system's execution. Good oracles are needed for fault detection and localization [10]. Approaches to derive suitable oracles can be grouped into static analysis of existing code-related artifacts and dynamic information from test case execution.

Static analysis: Static analysis involves using existing documentation or specifications outlining the expected behavior of the tested method. Examples include TORADOCU [11]. TOGA [12] distinguishes between exception and assertion oracles using a pre-trained CodeBERT model, which has been critically evaluated by the community [13], [14].

Dynamic analysis: Dynamic analysis infers likely invariants based on program executions, as seen in Daikon [15]. AVICENNA [16] infers failure-inducing invariants by analyzing the input of failing executions. LEARN2FIX [17] uses iterative test generation for failure inference. BASHIRI use dynamic execution data to classify test executions.

VI. Conclusion and Future Work

We present BASHIRI, a novel tool to derive *failure oracles* from execution features of labeled test cases. These interpretable oracles refer to executed lines and variable values, achieving 95% accuracy in our evaluation. BASHIRI addresses oracle continuity across program versions through feature mapping and improves quality via refinement loops.

Future work includes applying BASHIRI to test generation, fault localization, and automated repair, while improving feature mapping for better cross-version continuity.

Acknowledgments

This work is partially funded by the European Union (ERC S3, 101093186) and by the Deutsche Forschungsgemeinschaft (German Research Foundation, DFG) — ZE 509/7–2 and GR 3634/4–2 Emperor (261444241). Views and opinions expressed are 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] M. Smytzek, M. Eberlein, L. Grunske, and A. Zeller, "How execution features relate to failures: An empirical study and diagnosis approach," 2025. [Online]. Available: https://arxiv.org/abs/2502.18664

[2] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++: Combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, ser. WOOT'20. USA: USENIX Association, 2020.

[3] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 10321043. [Online]. Available: https://doi.org/10.1145/2976749.2978428

[4] M. Smytzek and A. Zeller, "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*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 17011705. [Online]. Available: https://doi.org/10.1145/3540250.3558915

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[6] M. Bordese, "Unidiff," 2012. [Online]. Available: https://github.com/matiasb/python-unidiff

[7] M. Smytzek, M. Eberlein, B. Serçe, L. Grunske, and A. Zeller, "Tests4py: A benchmark for system testing," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 557561. [Online]. Available: https://doi.org/10.1145/3663529.3663798

[8] R. Widyasari, S. Q. Sim, C. Lok, H. Qi, J. Phan, Q. Tay, C. Tan, F. Wee, J. E. Tan, Y. Yieh, B. Goh, F. Thung, H. J. Kang, T. Hoang, D. Lo, and E. L. Ouh, "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*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 15561560. [Online]. Available: https://doi.org/10.1145/3368089.3417943

[9] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The oracle problem in software testing: A survey," *IEEE Trans. Software Eng.*, vol. 41, no. 5, pp. 507–525, 2015. [Online]. Available: https://doi.org/10.1109/TSE.2014.2372785

[10] G. Jahangirova, D. Clark, M. Harman, and P. Tonella, "Test oracle assessment and improvement," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 247–258. [Online]. Available: https://doi.org/10.1145/2931037.2931062

[11] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, A. Zeller and A. Roychoudhury, Eds. ACM, 2016, pp. 213–224. [Online]. Available: https://doi.org/10.1145/2931037.2931061

[12] E. Dinella, G. Ryan, T. Mytkowicz, and S. K. Lahiri, "TOGA: A neural method for test oracle generation," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022*. ACM, 2022, pp. 2130–2141. [Online]. Available: https://doi.org/10.1145/3510003.3510141

[13] S. B. Hossain, A. Filieri, M. B. Dwyer, S. G. Elbaum, and W. Visser, "Neural-based test oracle generation: A large-scale evaluation and lessons learned," *CoRR*, vol. abs/2307.16023, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2307.16023

[14] Z. Liu, K. Liu, X. Xia, and X. Yang, "Towards more realistic evaluation for neural test oracle generation," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2023*, R. Just and G. Fraser, Eds. ACM, 2023, pp. 589–600. [Online]. Available: https://doi.org/10.1145/3597926.3598080

[15] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The Daikon system for dynamic detection of likely invariants," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 35–45, 2007. [Online]. Available: https://doi.org/10.1016/j.scico.2007.01.015

[16] M. Eberlein, M. Smytzek, D. Steinhöfel, L. Grunske, and A. Zeller, "Semantic debugging," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, S. Chandra, K. Blincoe, and P. Tonella, Eds. ACM, 2023, pp. 438–449. [Online]. Available: https://doi.org/10.1145/3611643.3616296

[17] M. Böhme, C. Geethal, and V.-T. Pham, "Human-in-the-loop automatic program repair," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, 2020, pp. 274–285.