Explaining and Debugging Pathological Program Behavior

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

ABSTRACT

Programs fail. But which part of the input is responsible for the failure? To resolve the issue, developers must first understand how and why the program behaves as it does, notably when it deviates from the expected outcome. A program's behavior is essentially the set of all its executions. This set is usually diverse, unpredictable, and generally unbounded. A pathological program behavior occurs once the actual outcome does not match the expected behavior. Consequently, developers must fix these issues to ensure the built system is the desired software. In our upcoming research, we want to focus on providing developers with a detailed description of the root causes that resulted in the program's unwanted behavior. Thus, we aim to automatically produce explanations that capture the circumstances of arbitrary program behavior by correlating individual input elements and their corresponding execution outcome.

To this end, we use the scientific method and combine generative and predictive models, allowing us (*i*) to learn the statistical relations between the features of the inputs and the program behavior and (*ii*) to generate new inputs to refine or refute our current explanatory prediction model.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Theory of computation** → *Grammars and context-free languages*; *Oracles and decision trees*; Active learning.

KEYWORDS

program behavior, debugging, behavior explanation, testing

ACM Reference Format:
Martin Eberlein. 2022. Explaining and Debugging Pathological Program Behavior. 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/10.1145/3540250.3558910

1 INTRODUCTION

All software behavior is triggered by some defined program input. But, which part of the input triggers which program behavior? Answering this question is fundamental to understanding how and why your program behaves as it does - especially when the behavior does not meet the expected outcome. To assure that a program fulfills the intented purpose, testing is an inevitable quality assurance technique [2] and several test case generation techniques [3, 5, 21–24, 26, 30, 34], including our own tool EvoGFuzz [12], have been proposed to detect pathological behavior. However, once a pathological behavior has been observed, localizing the underlying root cause of the defects and problems can be highly challenging and time-consuming [19, 29, 36]. Furthermore, fixing the problem may take even longer [15, 36].

When diagnosing why a program shows a specific pathological behavior, the first step is determining the class of program inputs that trigger the unintended behavior. Recently, Kampmann et al. [18] presented an approach to automatically identify the circumstances of program failures. Their method associates the program's defect with the syntactical features of the input data, allowing them to learn and extract the properties that result in the specific failing behavior. Their proposed tool Alhazen generates a diagnosis by forming an explanatory model based on observed, failure-inducing input properties. Then, additional test inputs are generated and executed to refine or refute the initial hypothesis, allowing Alhazen to obtain a prediction model for why the failure in question occurs. Although Alhazen can determine the circumstances of failure-triggering behavior, it cannot explain the root causes of all diverse and unbound program behaviors. Because Alhazen uses a binary oracle for each test case - pass or fail - it cannot classify and learn continuous numerical or categorical behavior, like unusually long run-time or memory consumption. Furthermore, the approach is limited to failure conditions inferred by syntactic input elements. Semantic properties, for instance, string allocations with $\langle string \rangle \to \langle string\text{-}length \rangle$ "`.`" $\langle chars \rangle$, where $\langle string\text{-}length \rangle$ and the number of used chars need to be equal in order to describe a valid string encoding, cannot be explained.

To overcome these limitations, we propose a research plan that extends and reaches beyond the statistical associations between input elements and program failure. We plan to unify and generalize the predictive and generative models in a single modular approach, going way beyond simple statistical relations. This generalization allows us to infer and refine relationships involving arbitrary input features and program behaviors, thus boosting our understanding of how and why software behaves as it does.

In summary, we plan to make the following contributions:

- We propose an approach to explain pathological program behavior that utilizes predictive and generative models.
- We plan to invoke semantic properties and intermediate features which serve as a vocabulary to model, predict, and explain pathological program behavior.
- We will perform an empirical evaluation of the proposed approach on a number of subjects and grammars and compare the results to the state-of-the-art [18].

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3558910

Figure 1: Context-Free grammar G, which allows us to encode simple strings of characters.

$\langle string \rangle \rightarrow \langle string\text{-}length \rangle$ "." $\langle chars \rangle$;
$\langle chars \rangle \rightarrow \langle char \rangle \mid \langle char \rangle\langle chars \rangle$;
$\langle char \rangle \rightarrow$ `/[a-Z]/`;
$\langle string\text{-}length \rangle \rightarrow \langle digit \rangle \mid \langle digit \rangle\langle string\text{-}length \rangle$;
$\langle digit \rangle \rightarrow$ `/[0-9]/`;

2 ASSOCIATING INPUT FEATURES AND PROGRAM BEHAVIOR

In the following, we illustrate how we can utilize input format specifications to extract syntactical features and properties from inputs and utilize them to explain pathological behavior.

2.1 Describing Input Features

As already noted, a program's behavior is the set of all execution outcomes triggered by all possible inputs. As this set is generally unbounded, program input can have a wide range of definitions. For example, users can provide input via the command line, programs can accept inputs by reading files, or programs can receive inputs from interacting with other programs and communicating with their environment. Thus, the set of all possible inputs determines how a program will behave. Inputs accepted by a program are called *valid* and can be described using a language. Note that we commonly separate input languages and programming languages. Input languages define the input space of valid inputs, whereas the programming language refers to the language in which the program was written. Notable exceptions are compilers or interpreters, which take actual programs as input, thus ranking them among the hardest to test pieces of software. Context-free grammars are a very popular formalism to describe the input language of a program and are a well-studied field of theoretical computer science, compiler design, and linguistics [17]. Using a grammar allows us to express a wide range of the properties of an input language. For example, context-free grammars are great for expressing an input's syntactical structure and are the formalism of choice to describe nested or recursive inputs.

Definition 2.1 (Context-Free Grammar). *A* context-free grammar *is a 4-tuple (N,T,P,s), where N is the set of* non-terminals, *T the set of* terminals, *P the set of* productions rules *with $P : N \rightarrow (N \cup T)$, and $\langle s \rangle \in N$ the initial starting symbol. Production rules are used to expand a* non-terminal *$\langle S \rangle \in N$ to one of its n alternatives A_i:*

$$\langle S \rangle \rightarrow A_1 \mid A_2 \mid A_3 \mid \dots \mid A_n \tag{1}$$

A tuple $(u, v) \in P$ can also be described using the binary relation $u \rightarrow v$, which is called a *derivation*. The most important short hand for derivations is $u \rightarrow^* v$ which signifies u derives to v using any amount of derivations. An expression w is called a *word* of the grammar G if it is derivable via $s \rightarrow^* w$.

For most languages, a context-free grammar corresponds to the set of all possible input structures we are interested in: Each word over the language is a valid input to the program. To learn a relation between all valid program inputs and their execution outcomes, we have to define a set of possible properties that we can map to the actual behavior. A property of a word that can be

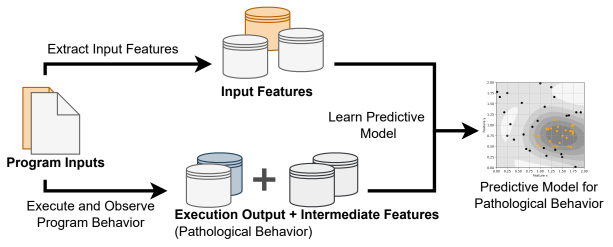
Extract Input Features
Input Features
Learn Predictive Model
Program Inputs
Execute and Observe Program Behavior
Execution Output + Intermediate Features
(Pathological Behavior)
Predictive Model for Pathological Behavior

Figure 2: We parse inputs into their features and learn a predictive model that associates the input features with the execution outcome.

described using only a context-free grammar is called a *syntactical feature*. Kampmann et al. [18] already proposed a set of syntactical features that can be derived from a context-free grammar, including properties like input length or the presence of specific derivation sequences.

For instance, let's pick up the example from Section 1 and consider the grammar G (Figure 1) that allows us to encode simple strings of characters. Now, we can define a syntactical feature that states whether the *terminal* "`A`" was used to derive a word of the language. Therefore, the word "*2.Ai*" has that property because, in the derivation sequence, we must use "`A`" to construct the word.

As the feature set proposed by Kampmann et al. can only capture a limited set of program behaviors, we want to expand their feature catalog to capture the circumstances of pathological behavior more precisely. For example, given a grammar that describes the language of program inputs, extended structural features would be expressed as (*i*) probability distributions of the grammar production rules; (*ii*) predicates over individual grammar elements, including the existence of arbitrary constraints on their semantic interpretation; (*iii*) predicates over multiple grammar elements, expressing relationships over their interpretation ("the number of $\langle chars \rangle$ is equal to the value of $\langle string\text{-}length \rangle$"); and (*iv*) predicates over substructures, but limited to a particular input element ("All $\langle chars \rangle$ in a string have an *ASCII* value between 97 and 122').

As an extension of the purely syntactical features, we also plan to utilize intermediate features observed during program execution to enrich the learning of the predictive model. Examples of such intermediate features are the execution time, the sequence of method calls, or the execution count of specific program loops. In addition, intermediate features allow us to gain insights into advanced cause-effect chains, further guiding the developer toward the root cause of the pathological behavior: "Because the number of $\langle chars \rangle$ was greater than $\langle string\text{-}length \rangle$, the variable *var* became true, which is why *validate()* was not called". Figure 2 illustrates that such input features would then serve as a vocabulary to model, predict, and explain pathological behavior.

2.2 Learning from Input Features to predict Behavior

Once we parsed the inputs into their syntactical features, we can use a prediction model to learn the relation between the individual parts of the input samples and their program behavior. Kampmann et al. [18] use a decision tree classifier to map the responsible elements to the execution outcome. Although decision tree classifiers are

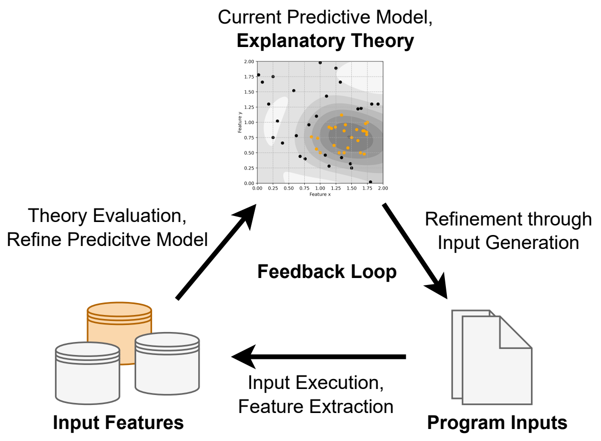

Figure 3: We use a combination of predictive and generative models to repeatedly refine the explanatory theory.

among the most popular machine learning algorithms [33], given their comprehensibility and simplicity, they often are incredibly prone and sensitive to small changes in the input data. Minor variations may already result in a complete change in the constructed tree and, consequently, in the final predictions [4]. Additionally, with their hyperrectangular cuts of the feature space, single decision trees tend to overfit the learning data, severely decreasing its generalizability and adaptability. With many advances in recent machine learning models [7], other classifiers may be much more efficient and improve upon the decision tree's limited prediction performance. Finally, binary decision trees can predict only a single (Boolean) outcome - pass or fail. However, to apply our approach to varying kinds of program behavior, we need to learn and predict multiple behaviors of a program at once. These behaviors would include not only Boolean pass or fail but also numerical features — notably features such as resource consumption or fitness for a particular goal. Thus, we plan to model the problem of learning a mapping between multiple input features and output features as a multi-task learning problem (MTL), typically solved by training a multi-task machine learning model. The general idea of MTL, as initially defined in [6], is to exploit the interdependencies between various related tasks by learning a joint model. In particular, MTL introduces an inductive bias that causes the model to prefer hypotheses that can explain more than one task, resulting in better generalization, i.e. by avoiding overfitting to a particular task, and improved learning efficiency.

3 EXPLAINING PATHOLOGICAL PROGRAM BEHAVIOR

Based on the initial starting conditions, the theory as to why the pathological behavior occurred may be far from being perfect. Thus, to improve the explanatory model, we employ the scientific method: hypothesis testing. We automatically generate new test inputs that cover pathological output features, hypothesize about the associations of program behavior and input features, and derive a general theory of the circumstances of the pathological program behavior by repeatedly refining the hypothesis through test experiments (Figure 3). After several iterations of conducting test experiments and refining the explanatory theory, we obtain a model that explains and predicts the pathological behavior. Finally, the obtained explanation allows us to generate targeted program inputs that fulfill the relevant properties and trigger the unwanted program behavior.

The generation of additional inputs serves the following two purposes: First, by generating inputs that satisfy the relevant features, we can effectively explore the surroundings of the original program inputs; refining and retraining the predictive model will strengthen the specialization— that is, distinguishing features that can be associated with the behavior of interest. Second, we will explore the surroundings of yet unknown program inputs by inverting or negating relevant features; hence, refining and retraining the predictive model will also result in generalization—if both predicates p and $\neg p$ result in the pathological behavior, then p's relevance is diminished.

4 CHALLENGES

Generating new Inputs. Arguably one of the biggest challenges will be to generate new program inputs that trigger a particular behavior systematically. If the predictive model captures the circumstances of the pathological behavior, derived inputs would need to fulfill these circumstances to efficiently test the surroundings of the pathological program behavior. Thus, we first need to identify the learned features responsible for the behavior and then instantiate new inputs to exercise the targeted behavior. A solution could be to use *ISLa* [32] which utilizes the smt constraint solver *Z3* [9].

Debugging Diagnosis. To produce a diagnosis for the user, we need to identify a combination of those input features that best explain the observed behavior. To this end, we will make use of techniques like *SHAP* [27], a game theoretic approach to explain the output of any classifier by identifying those features that contribute most to the classification — or in our case, the features that are most associated with the pathological behavior. These features would then be reported to the user as a diagnosis.

Test Oracle Problem. As previously mentioned, our approach requires the execution of the program under test to obtain the actual program behavior and output. This property is directly tied to the *test oracle problem.* Creating oracles manually is nontrivial and extremely time-consuming. A promising solution might be TOGA [11] that creates test oracles based on the context of the focal method.

5 OPPORTUNITIES

Predicting Behavior of yet unseen Inputs. Once we have trained a probabilistic model that captures the circumstances of program behavior, we can use them as predictors. Since we have learned the relation and associations between the input features and the execution outcome, we can provide a prediction of wherever a yet unseen input will result in the pathological program behavior. The trained models can serve as an extremely fast first line of defense without demanding many resources or invoking the actual program. Inputs deemed to trigger pathological behavior can be separated and isolated from the current running system and executed in a detached execution environment. This concept may, for instance, play a major role in providing a continuous active online service. Let's say a developer encountered a security vulnerability in their code and is about to fix it. Instead of taking the system offline for an unspecific amount of time, developers could use our generalized approach to detect and reject pathological input data. This will give developers enough time to locate and fix the vulnerability while minimizing the system's downtime.

Predicting Resource Consumption. With the introduction of intermediate features, we can relate structural input features to intermediate program behavior, such as covering specific methods or measuring the execution time and resource consumption (e.g., memory usage). Then, we would identify relevant input features associated with the pathological behavior using the predictive and generative models. Consequently, by reverting the learning process, we can identify the input features that require a high computational overhead or additional computational resources. And by far most intriguing: By using the trained predictive model we can make assumptions of the expected run-time or memory consumption without invoking the actual program: "the $\langle \textit{string} \rangle$ will require at least $\langle \textit{memory} \rangle$ because $\langle \textit{string-length} \rangle$ is greater than x"

Explainable Machine Learning. The beauty of our proposed approach is that we can treat and evaluate the program under test as a black-box. Thus, it is a highly intriguing opportunity to use our approach to analyze the behavior (i.e., predictions) of other machine learning models, allowing us to explain the decisions made by hard-to-interpret classifiers or regression models. Furthermore, the iterative nature of our approach enables us to create an explainable surrogate model that assesses the circumstances of the black-box model's predictions. Compared to other techniques, the main advantage is the potential benefit of refining the explanatory prediction model. Again, using the general idea of MTL allows us to not only explain binary classifiers but opens a path to explain numerical or categorical output.

6 EVALUATION

To evaluate our learned explanatory theories, we will use an experimental evaluation based on the grammars and test subjects used by Kampmann et al. [18]. First, we will derive a set of test suites with varying sizes by using the grammar as a generator. Then, based on these test suites, we will be able to assess the predictive models' accuracy, precision, recall, and f1 scores. In particular, our experimental setup will randomly separate the test suites into a training set to train the predictive models and a test set to evaluate them. The following research questions will guide our evaluation:

- RQ1 **Predictor**: Can we predict pathological behavior of a program precisely?
- RQ2 **Producer**: Can we produce pathological behavior triggering inputs efficiently?
- RQ3 **Debugging Aid**: Can we reduce the input space to help developers focus on the relevant aspects of the pathological program behavior?

By assessing the models' quality as predictor (**RQ1**) and producer (**RQ2**), we ensure that they neither overspecialize nor overgeneralize. Additionally, by producing more pathological program inputs, we want to investigate whether we can improve fault localization and program repair, further contributing to the practical relevance of our approach.

7 RELATED WORK

Detecting Bugs. A key component of our proposed approach is the generation of additional inputs to refine a working hypothesis. Common approaches that generate inputs to detect pathological behavior automatically are based on the idea of fuzzing [14, 23, 28, 30, 35] and symbolic execution. To reach the deep layers of a program, grammar-based fuzzing uses grammars to generate syntactically valid inputs [1, 12]. Havrikov and Zeller [16] introduced a grammar-based method to systematically cover elements of the grammar that allows revealing more profound defects cost-effectively. In the context of grammar-based fuzzing, the generation of new inputs can also be guided by probabilities attached to competing rules in the grammar. Using a set of initial input files as seeds to obtain a probabilistic grammar, Sorumekun et al. [31] generate similar inputs to these seed files. Using these probabilistic grammars, we showed that our tool EvoGFuzz [12] is able to further refine the probabilities towards specific goals. Symbolic execution [8, 20] traditionally explores program behavior with symbolic input values in contrast to concrete ones, as done in testing and fuzzing. However, all the above approaches focus mostly on detecting security vulnerabilities and failing program behavior. In contrast, our approach focuses on explaining previously detected pathological behavior. Therefore, developers should apply our approach after unwanted behavior has been detected to gain insights into the circumstances under which the behavior occurs.

Program Behavior Classification. One goal of our approach is to predict and explain program behavior. Recently, Tizpaz-Niari et al. [33] proposed an approach to determine and explain differential performance bugs. Their tool *DPFuzz* uses an evolutionary fuzzing approach to generate interesting inputs and then clusters them according to their execution time. Finally, *DPFuzz* uses a decision tree to explain the performance differences in terms of program inputs and internals. Machine learning techniques are also often used to learn models that classify programs into benign and malicious software for vulnerability detection [10]. Elish et al. [13] use SVM's to predict defect-prone software modules and Lo et al. [25] proposed a technique that first extracts iterative patterns from program traces of known ordinary and failing executions. Then, they perform feature selection to select the most promising features for classification. However, in contrast to our approach, the mentioned techniques do not refine their predictive models.

8 CONCLUSION

Our approach focuses on repeatedly refining predictive and generative models to explain the circumstances of pathological behavior, allowing us to infer and refine relationships involving arbitrary input features and thus boost our understanding of how and why software behaves as it does. We expect that our proposed approach will help developers to understand, detect, and debug the program's pathological behavior.

ACKNOWLEDGMENTS

This research work is supervised by Prof. Dr. Lars Grunske from the Humboldt-Universität zu Berlin and was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – GR 3634/4-2 Emperor (392561203).

REFERENCES

[1] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/nautilus-fishing-for-deep-bugs-with-grammars/

[2] Antonia Bertolino. 2007. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE'07)*. IEEE, 85–103.

[3] William Blair, Andrea Mambretti, Sajjad Arshad, Michael Weissbacher, William Robertson, Engin Kirda, and Manuel Egele. 2020. HotFuzz: Discovering Algorithmic Denial-of-Service Vulnerabilities Through Guided Micro-Fuzzing. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/hotfuzz-discovering-algorithmic-denial-of-service-vulnerabilities-through-guided-micro-fuzzing/

[4] L Breiman, JH Friedman, R Olshen, and CJ Stone. 1984. Classification and Regression Trees. (1984).

[5] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. 2009. WISE: Automated test generation for worst-case complexity. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. IEEE, 463–473. https://doi.org/10.1109/ICSE.2009.5070545

[6] Rich Caruana. 1997. Multitask Learning. *Mach. Learn.* 28, 1 (1997), 41–75. https://doi.org/10.1023/A:1007379606734

[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 785–794. https://doi.org/10.1145/2939672.2939785

[8] Lori A. Clarke. 1976. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Transactions on Software Engineering* SE-2, 3 (Sept 1976), 215–222. https://doi.org/10.1109/TSE.1976.233817

[9] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[10] S. Delphine Immaculate, M. Farida Begam, and M. Floramary. 2019. Software Bug Prediction Using Supervised Machine Learning Algorithms. In *2019 International Conference on Data Science and Communication (IconDSC)*. 1–7. https://doi.org/10.1109/IconDSC.2019.8816965

[11] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. IEEE, 2130–2141. https://doi.org/10.1145/3510003.3510141

[12] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. 2020. Evolutionary Grammar-Based Fuzzing. In *Proceedings of the 12th Symposium on Search-Based Software Engineering (SSBSE 2020)*.

[13] Karim O Elish and Mahmoud O Elish. 2008. Predicting defect-prone software modules using support vector machines. *Journal of Systems and Software* 81, 5 (2008), 649–660.

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining incremental steps of fuzzing research. In *14th {USENIX} Workshop on Offensive Technologies ({WOOT} 20)*.

[15] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. 2020. Memory and resource leak defects and their repairs in Java projects. *Empir. Softw. Eng.* 25, 1 (2020), 678–718. https://doi.org/10.1007/s10664-019-09731-8

[16] Nikolas Havrikov and Andreas Zeller. 2019. Systematically Covering Input Structure. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering* (San Diego, California) *(ASE '19)*. IEEE Press, 189–199. https://doi.org/10.1109/ASE.2019.00027

[17] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. 2001. Introduction to automata theory, languages, and computation. *Acm Sigact News* 32, 1 (2001), 60–65.

[18] Alexander Kampmann, Nikolas Havrikov, Ezekiel O Soremekun, and Andreas Zeller. 2020. When does my program do this? learning circumstances of software behavior. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 1228–1239.

[19] Charles Edwin Killian, Karthik Nagaraj, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. 2010. Finding latent performance bugs in systems implementations. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, Gruia-Catalin Roman and André van der Hoek (Eds.). ACM, 17–26. https://doi.org/10.1145/1882291.1882297

[20] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[21] Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi. 2019. PySE: Automatic Worst-Case Test Generation by Reinforcement Learning. In *12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019*. IEEE, 136–147. https://doi.org/10.1109/ICST.2019.00023

[22] Xuan-Bach Dinh Le, Corina S. Pasareanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. 2019. Saffron: Adaptive Grammar-based Fuzzing for Worst-Case Analysis. *ACM SIGSOFT Softw. Eng. Notes* 44, 4 (2019), 14. https://doi.org/10.1145/3364452.3364455

[23] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. 2018. PerfFuzz: automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, Frank Tip and Eric Bodden (Eds.). ACM, 254–265. https://doi.org/10.1145/3213846.3213874

[24] Penghui Li, Yinxi Liu, and Wei Meng. 2021. Understanding and Detecting Performance Bugs in Markdown Compilers. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 892–904. https://doi.org/10.1109/ASE51524.2021.9678611

[25] David Lo, Hong Cheng, Jiawei Han, Siau-Cheng Khoo, and Chengnian Sun. 2009. Classification of software behaviors for failure detection: a discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 557–566.

[26] Kasper Søe Luckow, Rody Kersten, and Corina S. Pasareanu. 2017. Symbolic Complexity Analysis Using Context-Preserving Histories. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*. IEEE Computer Society, 58–68. https://doi.org/10.1109/ICST.2017.13

[27] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 4765–4774. https://proceedings.neurips.cc/paper/2017/hash/8a20a8621978632d76c43dfd28b67767-Abstract.html

[28] Hoang Lam Nguyen and Lars Grunske. 2022. BEDIVFUZZ: Integrating Behavioral Diversity into Generator-based Fuzzing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. IEEE, 249–261. https://doi.org/10.1145/3510003.3510182

[29] Adrian Nistor, Tian Jiang, and Lin Tan. 2013. Discovering, reporting, and fixing performance bugs. In *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, Thomas Zimmermann, Massimiliano Di Penta, and Sunghun Kim (Eds.). IEEE Computer Society, 237–246. https://doi.org/10.1109/MSR.2013.6624035

[30] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 2155–2168. https://doi.org/10.1145/3133956.3134073

[31] Ezekiel Soremekun, Esteban Pavese, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. 2020. Inputs from Hell: Learning Input Distributions for Grammar-Based Test Generation. *IEEE Transactions on Software Engineering* (2020).

[32] Dominic Steinhöfel and Andreas Zeller. 2022. Input Invariants, In Technical Track. *ESEC/FSE 2022*. https://publications.cispa.saarland/3596/

[33] Saeid Tizpaz-Niari, Pavol Černỳ, and Ashutosh Trivedi. 2020. Detecting and understanding real-world differential performance bugs in machine learning libraries. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 189–199.

[34] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. 2018. Singularity: pattern fuzzing for worst case complexity. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 213–223. https://doi.org/10.1145/3236024.3236039

[35] Michal Zalewski. 2021. American Fuzzy Lop (AFL) - a security-oriented fuzzer. http://lcamtuf.coredump.cx/afl/. Accessed: August 26, 2021.

[36] Shahed Zaman, Bram Adams, and Ahmed E. Hassan. 2012. A qualitative study on performance bugs. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, Michele Lanza, Massimiliano Di Penta, and Tao Xie (Eds.). IEEE Computer Society, 199–208. https://doi.org/10.1109/MSR.2012.6224281