From Input to Failure: Explaining Program Behavior via Cause-Effect Chains

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

***Abstract*—Debugging a fault in a program is an error-prone and resource-intensive process that requires considerable work. My doctoral research aims at supporting developers during this process by integrating test generation as a feedback loop into a novel fault diagnosis to narrow down the causality by validating or disproving suggested hypotheses. I will combine input, output, and state to detect relevant relations for an immersive fault diagnosis. Further, I want to introduce an approach for a targeted test that leverages statistical fault localization to extract oracles based on execution features to identify failing tests.**

***Index Terms*—Software/Software Engineering, Software Engineering, Testing and Debugging, Debugging aids, Diagnostics**

I. Introduction

Debugging is one of the most challenging tasks in today's software engineering. To repair a fault, developers face the problem of reproducing, understanding what causes it, and identifying where it originates. During my doctoral research, I plan to address these issues by developing techniques that provide an adequate and precise fault diagnosis refined with test generation helping developers to repair faulty software efficiently. These diagnoses aim to assist developers in their daily task of debugging complex software systems by providing detailed insides into the fault further improvable by the developer's feedback.

II. Fault Diagnosis

My research focuses on providing adequate diagnoses for faults, revealing the buggy locations, detecting failure-inducing inputs, and constructing an entire cause-effect chain that leads to the fault, including its precise root cause. I will demonstrate the approaches along the Heartbleed vulnerability [1]. Figure 1a shows the input and output specifications of the Heartbeat exchange as a grammar and Figure 1b the example representation of a Heartbeat message in the program code.

$\langle request \rangle$ ::= 0x1 $\langle length \rangle$ $\langle payload \rangle$ $\langle padding \rangle$
$\langle response \rangle$ ::= 0x2 $\langle length \rangle$ $\langle payload \rangle$ $\langle padding \rangle$
$\langle length \rangle$ ::= $\langle int \rangle$
$\langle payload \rangle$ ::= ϵ | $\langle byte \rangle$ $\langle payload \rangle$
$\langle padding \rangle$ ::= ϵ | $\langle byte \rangle$ $\langle padding \rangle$

(a) Syntax of TLS Heartbeat exchanges

```
struct {
    HeartbeatMessageType type;
    uint16 payload_length;
    opaque payload[...];
    opaque padding[padding_length];
} HeartbeatMessage;
```

(b) Struct for TLS Heartbeat messages

Fig. 1: TLS Heartbeat protocol

A. Input and Output

The plan for diagnosing the input is to leverage a grammar that describes its elements as terminal and non-terminal symbols and learn conditions needed for a failure-inducing input. This technique orientates on the existing ISLearn [2]–[4] approach. ISLearn leverages an extendable pattern catalog to automatically learn constraints over the input, resulting in a formal description. The goal is to extend this approach by considering features based on their significance to trigger the fault. This approach could infer for the Heartbleed vulnerability that the fault occurs when $\text{int}(\langle request \rangle.\langle length \rangle) > \text{len}(\langle request \rangle.\langle payload \rangle)$ holds. I also want to apply the developed techniques to the output. Even though the output does not directly influence the fault and cannot show its origins, it can help to identify faulty executions and could contain hints at the fault's cause. By combining input and output specifications, the approach could infer that the fault shows if $\langle request \rangle.\langle payload \rangle \neq \langle response \rangle.\langle payload \rangle$ holds.

B. State

Diagnosing the input and the output is one part but does not reveal the fault itself, only its higher-level causes and results. Hence, I want to develop an approach relying on the one for input and output that learns the constraints that need to hold for the state during the program execution of failing runs. For this approach, I consider the state as the set of local and global variables that exist at a certain point during the execution and the current stack trace, making it feasible to extract from a single log. This approach produces an entire cause-effect chain by inferring which constraints imply the values or conditions of a later point in the execution to pinpoint the exact state and changes for which the fault arises.

I will consider multiple features that may hold during the execution; for instance, the existence of a specific variable in the state or the value of a variable is less than a specific number. The approach infers features that correlate with the fault by calculating the support of each feature for failing runs. Then the approach learns the connections between the

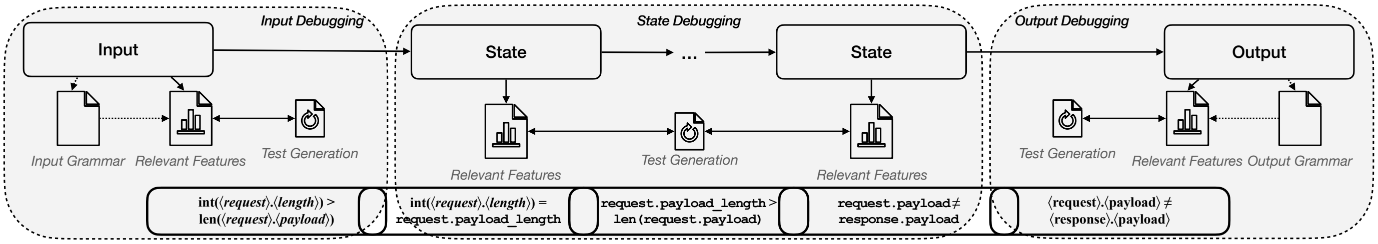

Fig. 2: Extracting cause-effect chains: input (left) $\text{int}(\langle \textit{request}\rangle.\langle \textit{length}\rangle) > \text{len}(\langle \textit{request}\rangle.\langle \textit{payload}\rangle)$ and output (right) $\langle \textit{request}\rangle.\langle \textit{payload}\rangle \neq \langle \textit{response}\rangle.\langle \textit{payload}\rangle$ connect to the origin `request.payload_length` $>$ `len(response.payload)`

features that correlate with failing runs to create a cause-effect chain, e.g., if `request.payload_length` $>$ `len(request.payload)` then `request.payload` $\neq$ `response.payload`. Note that the above approaches consider the grammar, while this example considers the concrete code at specific times. This approach can simultaneously start from inputs and outputs until the derived chains meet.

Further, I will leverage this derived cause-effect chain not only to understand the fault's origin but also to pinpoint the exact location when the execution gets affected by the fault.

C. A Unified Approach

My fault diagnosis approach will consider the input, output, and state diagnosis strategies. Figure 2 demonstrates their interleaving and the derived cause-effect chain for the Heartbleed example. This joint approach will leverage each of the approaches to refine the diagnoses of one another iteratively. Understanding what parts of the input are failure-inducing improves inferring the execution features that correspond to the bugs and vice-versa. The same holds for the output. The result of this approach would then include a detailed description of reproducing the failure, where it originates, and how it propagates.

III. Test Generation

The approaches in Section II consider an immense search space of possible solutions that I plan to reduce by introducing a feedback loop that leverages a guided test generation to refine diagnoses by supporting or disproving inferred hypotheses iteratively. I will correlate execution features from statistical fault localization (SFL) [5]–[8] with input features learned with the approach in Section II-B to generate tests that satisfy certain features. I also require an oracle that distinguishes between passing and failing runs, meaning I can ignore the expected result. My approach here is to collect features with SFL that describe the runs, which describe an N-dimensional space. When classifying a test, the approach measures the distance between its surrounding tests in this space and then calculates how likely a test will fail. This approach is further improvable by integrating a human decider to narrow down the space corresponding to failing test cases. If a generated test is interesting, e.g., if it is not nearby any previous test, the approach could ask a human to classify it. I plan to leverage my SFLKit [9], [10] to extract the features to correlate these with the input or derive oracles.

IV. Planned Evaluation

To evaluate the approaches presented in Section II and Section III, I plan to extend the BugsInPy [11], [12] benchmark with the possibility of generating system and unit tests and oracles to verify tests. I will leverage the generation to evaluate the statistical oracles approach and the specification of the bug to evaluate the other approaches. To the best of my knowledge, there exists no sufficient baseline. Hence, I will consider the precision, recall, and accuracy of correctly generating tests and identifying a fault's root cause. Besides, I want to conduct user studies to evaluate the developer's benefits; even though this is challenging, it could provide further inside into their needs. Yet, my research will be independent of user studies' results.

V. Related Work

a) Learning Contraints: DAIKON [13] and other recent research like Yao et al. [14] are conceptualized for analyzing a program rather than inferring diagnoses or generating tests. The work by Malik et al. [15] and Garg et al. [16] present approaches that learn invariants of complex data and linear data structures that theoretically apply to test generation but do not include a needed specification for the generation. The same holds for Usman et al. [17], who studied models for extracting invariants over selected data structure types.

b) Oracle Problem: Ernst et al. [18] presented an approach based on Daikon [13] that verifies test cases against learned invariants of the program that does not consider the correct output. An approach by Böhme et al. [19] learns a model from generated tests and lets humans label tests that could refine the model. However, this approach works only on a small scale for uncomplicated oracles.

c) Fault Diagnosis: ALHAZEN [20] is an approach that learns the causes of faults in the input and iteratively refines inferred hypotheses by generating new tests. DDSET [21] extracts a pattern that matches a failure-inducing input. Both approaches only consider the input and can help identify the fault at a higher level but keep the underlying cause unknown. DeltaDebugging [22] is designed for the input but can apply to the program, which would fail when considering highly dependent faults. Earlier work by Zeller [23] derives cause-effect chains from a program by modifying the state during execution which could lead to false diagnoses.

VI. Acknowledgements

Prof. Dr. Andreas Zeller advises the author of this paper.

REFERENCES

[1] Synopsys, Inc., "The heartbleed bug," https://heartbleed.com/, 2020.

[2] D. Steinhöfel and A. Zeller, "Input invariants," 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. 583–594. [Online]. Available: https://doi.org/10.1145/3540250.3549139

[3] D. Steinhöfel, "Isla: Input specification language," https://github.com/rindPHI/isla, 2022.

[4] ——, "Islearn," https://github.com/rindPHI/islearn, 2022.

[5] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, 2002, pp. 467–477.

[6] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 273–282. [Online]. Available: https://doi.org/10.1145/1101908.1101949

[7] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 15–26. [Online]. Available: https://doi.org/10.1145/1065010.1065014

[8] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: Association for Computing Machinery, 2013, pp. 191–201. [Online]. Available: https://doi.org/10.1145/2483760.2483785

[9] 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. 1701–1705. [Online]. Available: https://doi.org/10.1145/3540250.3558915

[10] M. Smytzek, "Sflkit: A workbench for statistical fault localization," https://github.com/uds-se/sflkit, 2022.

[11] 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*. ACM, Nov 2020, pp. 1556–1560. [Online]. Available: https://dl.acm.org/doi/10.1145/3368089.3417943

[12] ——, "Bugsinpy," https://github.com/soarsmu/BugsInPy, 2020.

[13] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st International Conference on Software Engineering*, ser. ICSE '99. New York, NY, USA: Association for Computing Machinery, 1999, p. 213–224. [Online]. Available: https://doi.org/10.1145/302405.302467

[14] J. Yao, G. Ryan, J. Wong, S. Jana, and R. Gu, "Learning nonlinear loop invariants with gated continuous logic networks," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 106–120. [Online]. Available: https://doi.org/10.1145/3385412.3385986

[15] M. Z. Malik, A. Pervaiz, E. Uzuncaova, and S. Khurshid, "Deryaft: A tool for generating representation invariants of structurally complex data," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 859–862. [Online]. Available: https://doi.org/10.1145/1368088.1368223

[16] P. Garg, C. Löding, P. Madhusudan, and D. Neider, "Learning universally quantified invariants of linear data structures," *CoRR*, vol. abs/1302.2273, 2013. [Online]. Available: http://arxiv.org/abs/1302.2273

[17] M. Usman, W. Wang, K. Wang, C. Yelen, N. Dini, and S. Khurshid, "A study of learning data structure invariants using off-the-shelf tools," in *Model Checking Software: 26th International Symposium, SPIN 2019, Beijing, China, July 15–16, 2019, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 226–243. [Online]. Available: https://doi.org/10.1007/978-3-030-30923-7_13

[18] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[19] 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.

[20] A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller, "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*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1228–1239. [Online]. Available: https://doi.org/10.1145/3368089.3409687

[21] R. Gopinath, A. Kampmann, N. Havrikov, E. O. Soremekun, and A. Zeller, "Abstracting failure-inducing inputs," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 237–248. [Online]. Available: https://doi.org/10.1145/3395363.3397349

[22] A. Zeller, "Yesterday, my program worked. today, it does not. why?" in *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE-7. Berlin, Heidelberg: Springer-Verlag, 1999, p. 253–267.

[23] ——, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*, ser. SIGSOFT '02/FSE-10. New York, NY, USA: Association for Computing Machinery, 2002, p. 1–10. [Online]. Available: https://doi.org/10.1145/587051.587053