

Reproducing and Debugging Field Failures in House

Wei Jin

Georgia Institute of Technology, USA
weijin@gatech.edu

Abstract—As confirmed by a recent survey among developers of the Apache, Eclipse, and Mozilla projects, failures of the software that occur in the field, after deployment, are difficult to reproduce and investigate in house. To address this problem, we propose an approach for in-house reproducing and debugging failures observed in the field. This approach can synthesize several executions similar to an observed field execution to help reproduce the observed field behaviors, and use these executions, in conjunction with several debugging techniques, to identify causes of the field failure. Our initial results are promising and provide evidence that our approach is able to reproduce failures using limited field execution information and help debugging.

I. PROBLEM AND MOTIVATION

Due to the limitations of in-house quality assurance activities, and the increasing complexity of software systems, deployed software is bound to generate *field failures*—failures that occur after deployment, while the software is running on user machines. Field failures are notoriously difficult to reproduce and investigate in house, as shown by a recent survey among developers [32].

To address this problem, both researchers and practitioners have proposed several techniques and systems. Several industrial bug reporting systems are designed to collect information (e.g., stack traces and register dumps) when a program crashes and send it back to the software producer (e.g., [2], [4]). Although useful, the information collected by these systems is often too limited to allow for reproducing and investigating a failure. In fact, subsequent research has shown that the usefulness of the information collected for reproducing and debugging increases when more data are collected [10], [11]. Researchers have therefore defined several capture-and-replay techniques that gather a wide spectrum of richer data, ranging from path profiles to complete execution recordings (e.g., [3], [10], [11], [16]). These techniques can address the issue of reproducibility of field failures. However, richer information can be very expensive to collect and is likely to contain sensitive user information.

The overall goal of our research is to address these limitations of existing techniques by developing a general framework for (1) reproducing field failures in a faithful way, without imposing too much overhead on the users or violating the users' privacy, and (2) identifying likely causes of the field failure. More precisely, we aim to develop a general technique that synthesizes, given a program P , a field execution E of P that results in a failure F , and a set of execution data D for E , a set of in-house executions as follows. The executions

must be actual executions of P , must be “similar” to E , and must result in a failure “similar” to F .¹ Moreover, the approach should generate executions using only P and D , without the need for any additional information. Finally, the approach should be able to identify the causes of F using the generated executions.

II. OUR APPROACH

In this section, we summarize our BugRedux technique for reproducing field failures in-house using limited field data. BugRedux is presented in detail in Reference [17]. BugRedux consists of three main components: the instrumenter, the analyzer, and, potentially, the debugger.

Instrumenter. The first component of BugRedux is the *instrumenter*, which takes as input an *application* provided by *software developers* and generates, by adding probes into the program, an *instrumented application* that can collect execution data and add such execution data to *crash reports* from the field.

Analyzer. The *analyzer* takes as input a *crash report* and tries to generate a *test input* that, when provided to the *application*, results in the same failure that was observed in the field. The generated input can be provided to *software testers* or an automated debugging approach to debug the field failure. This part of the approach can be defined in different ways depending on the kind of execution data collected and on the technique used for synthesizing execution. In particular, we consider execution data consisting of sequences of statements and use a guided symbolic execution algorithm to synthesize executions. Our algorithm implements a customized search strategy that guides the symbolic execution through the sequence of statements considered in the right order.

Fault Localizer (On-going work). One of the benefits of BugRedux is that, in many cases, it can generate a number of similar *actual* failing and passing executions. In our on-going work, we apply our customized statistical based fault localization techniques on these synthesized executions from BugRedux to perform fault localization.

III. EVALUATION AND DISCUSSION

To evaluate our approach, we implemented BugRedux for C programs using the LLVM compiler infrastructure and KLEE, a symbolic execution engine. We then used our implementation to perform an empirical study to answer the following two research questions:

- **RQ1:** Can BugRedux synthesize executions that are able to reproduce field failures?

¹ For more details on the concept of similarity, see Reference [17].

This paper summarizes our recent work published at ICSE2012 [17] and discusses some ongoing future work.

TABLE I
EFFECTIVENESS AND EFFICIENCY OF BUGREDUX IN SYNTHESIZING
EXECUTIONS STARTING FROM COLLECTED EXECUTION DATA.

Name	POF	Call stack	Call sequence	Complete trace
sed.fault1	N/A	N/A	98s	Y N/A
sed.fault2	N/A	N/A	17349s	Y N/A
grep	N/A	16s	N 48s	Y N/A
gzip.fault1	3s	Y 18s	Y 11s	Y N/A
gzip.fault2	20s	N 28s	N 25s	Y N/A
ncompress	155s	Y 158s	Y 158s	Y N/A
polymorph	65s	Y 66s	Y 66s	Y N/A
aeon	1s	Y 1s	Y 1s	Y 1s Y
rysnc	N/A	N/A	88s	Y N/A
glftpd	5s	Y 5s	Y 4s	Y N/A
htget	53s	N 53s	N 9s	Y N/A
socat	N/A	N/A	876s	Y N/A
tipxd	27s	Y 27s	Y 5s	Y N/A
aspell	5s	N 5s	N 12s	Y N/A
xmail	N/A	N/A	154s	Y N/A
exim	N/A	N/A	269s	Y 5624s Y

- **RQ2:** If so, which types of execution data provide the best tradeoffs in terms of cost benefit?

The subjects we used in this study are from three repositories: SIR [1], BugBench [23], and exploit-db [8]. The ten faults from exploit-db are real *field failures* actually discovered by users, that is, exactly the kind of issues that our technique targets. As field data, we used four different kinds of data: POF, the point of failure; call stack, call sites on the crash stack; call sequence, a sequence of call sites traversed in the failing execution; and complete execution traces, a sequence of branches visited in the execution.

We list parts of our study results in Table I. The table shows, for each failing execution fe considered and each type of execution data ed , the time it took BugRedux to generate inputs that mimicked fe using ed (or “N/A” if BugRedux was unable to generate such inputs in the allotted time) and whether the mimicked execution reproduced the observed failure (“Y” or “N”). This table addresses both questions and allows us to make some observations.

For RQ1, the table shows that BugRedux can reproduce the observed failures starting from a set of suitable execution data. For RQ2, the results provide initial but clear evidence that call sequences represent the best choice, among all four types considered, in terms of cost-benefit tradeoffs: BugRedux was able to reproduce all failures using call sequences and collect call sequences with an acceptable time and space overhead. We can also make another observation based on the results: POFs and call stacks do not seem to be particularly helpful for reproducing failures. We manually examined the faults and found that failure sites were often distant from the actual location of the faults. In these cases, the POFs and call stacks are unlikely to provide useful information for reproducing field failures. If confirmed, this finding would suggest that extending current crash reports with additional field information should make them more useful to developers.

IV. BACKGROUND AND RELATED WORK

Failure reproduction and software debugging are extremely prolific research areas, and the related work is consequently vast. In this section, we briefly summarize the techniques more closely related to ours.

A. Failure-Reproduction Techniques

One type of related techniques (*e.g.*, [3], [11], [21]) record execution events and possibly interactions between programs and the running environment to later replay or analyze them in house. These approaches tend to either capture too much information, and thus raise practicality and privacy issues, or too little information, and thus be ineffective. Recently, researchers started investigating approaches to reproduce field failures using more limited information. For example, some researchers used weakest preconditions to find inputs that can trigger certain types of exceptions in Java programs [9], [15], [24]. Some leveraged run-time logs to infer paths near logging statements [26], [27]. Some others proposed using partial object states at method levels to reproduce the same crash [7]. The most related approaches are ESD [28] and another technique by Cramer, Bianchini, and Zwaenepoel [13] (CBZ). ESD uses symbolic execution to try to generate inputs that would reach the point of failure (POF). CBZ improves ESD by using partial branch traces, where the relevant branches are identified by different program analyses. However, our empirical study shows POF and partial traces are unlikely to be successful for some failures.

B. Software Debugging

The first family of related debugging techniques are statistical fault localization techniques, which performs fault localization based on a large number of passing and failing executions using statistical inference [5], [6], [18], [20], [22], [31]. The major limitation of these techniques is that their results highly depend on the quality of test cases [25]. Another family of related techniques are experimental debugging techniques, such as delta debugging (*e.g.*, [12], [29], [30]). These techniques simplify and isolate failure causes in program inputs, instruction, or states from both passing and failing executions. The problem with these approaches is that the manipulation on program executions may introduce unsoundness. A third family of automated debugging techniques are formula-based fault localization techniques (*e.g.*, [14], [19]). These techniques transform programs and oracles into formulas, and reason on these formulas to perform fault localization. Scalability is the major problem of these techniques.

V. CONCLUSION

Reproducing and debugging field failures are notoriously arduous tasks for developers. To address this problem, we presented BugRedux, a general approach for reproducing field failures in house. We performed an empirical evaluation of BugRedux’s performance using four different types of execution data. Our results provide evidence that, for the cases considered, BugRedux is practical and can reproduce real field failures. Currently, we are investigating ways to extend BugRedux with fault localization capabilities. Our preliminary results are promising: by (1) generating multiple passing and failing executions that mimic a filed failure and (2) performing statistical debugging on such executions, BugRedux was able to successfully rank the actual fault within the top 15 entities for all of the 10 complex bugs we used in our study.

REFERENCES

- [1] Software-artifact Infrastructure Repository. <http://sir.unl.edu/>, Apr. 2012.
- [2] Technical Note TN2123: CrashReporter. <http://developer.apple.com/technotes/tn2004/tn2123.html>, Apr. 2012.
- [3] The Amazing VM Record/Replay Feature in VMware Workstation⁶. <http://communities.vmware.com/community/vmtn/cto/steve/blog/2007/04/18/the-amazing-vm-recordreplay-feature-in-vmware-workstation-6>, Apr. 2012.
- [4] Windows Error Reporting: Getting Started. <http://www.microsoft.com/whdc/maintain/StartWER.mspix>, 2012.
- [5] R. Abreu, P. Zoetewij, and A. J. C. van Gemund. An observation-based model for fault localization. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, WODA '08, pages 64–70, New York, NY, USA, 2008. ACM.
- [6] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Directed test generation for effective fault localization. In *Proceedings of the 19th international symposium on Software testing and analysis, ISSTA '10*, pages 49–60, New York, NY, USA, 2010. ACM.
- [7] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, pages 542–565, 2008.
- [8] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley. AEG: Automatic Exploit Generation. In *Proceedings of the 18th Network and Distributed System Security Symposium*, Feb. 2011.
- [9] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: A Powerful Approach to Weakest Preconditions. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 363–374, 2009.
- [10] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani. HOLMES: Effective Statistical Debugging via Efficient Path Profiling. In *ICSE 2009*, pages 34–44, 2009.
- [11] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *ICSE 2007*, pages 261–270, 2007.
- [12] H. Cleve and A. Zeller. Locating causes of program failures. *Proceedings of the 27th international conference on Software engineering - ICSE '05*, page 342, 2005.
- [13] O. Cramer, R. Bianchini, and W. Zwaenepoel. Striking a New Balance Between Program Instrumentation and Debugging Time. In *Proceedings of the 6th European Conference on Computer Systems*, pages 199–214, 2011.
- [14] E. Ermis, M. Schäfer, and T. Wies. Error invariants. In *Proceedings of the 18th International Symposium on Formal Methods, FM '12*, 2012.
- [15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 2002.
- [16] L. Jiang and Z. Su. Context-aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 184–193, 2007.
- [17] W. Jin and A. Orso. BugRedux: Reproducing Field Failures for In-house Debugging. In *ICSE 2012*, pages 474–484, 2012.
- [18] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *ICSE 2002*, pages 467–477, 2002.
- [19] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI 2011*, pages 437–446, New York, NY, USA, 2011.
- [20] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. *SIGPLAN Not.*, 38(5):141–154, May 2003.
- [21] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *PLDI 2005*, pages 15–26, 2005.
- [22] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: statistical model-based bug localization. In *ESEC/FSE-13*, pages 286–295, 2005.
- [23] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: Benchmarks for Evaluating Bug Detection Tools. In *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.
- [24] M. G. Nanda and S. Sinha. Accurate Interprocedural Null-Dereference Analysis for Java. In *Proceedings of the 31st International Conference on Software Engineering*, pages 133–143, 2009.
- [25] C. Parnin and A. Orso. Are Automated Debugging Techniques Actually Helping Programmers? In *ISSTA 2011*, pages 199–209, Toronto, Canada, July 2011.
- [26] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. SherLog: Error Diagnosis by Connecting Clues from Run-time Logs. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 143–154, 2010.
- [27] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving Software Diagnosability via Log Enhancement. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–14, 2011.
- [28] C. Zamfir and G. Candea. Execution Synthesis: A Technique for Automated Software Debugging. In *Proceedings of the 5th European Conference on Computer Systems*, pages 321–334, 2010.
- [29] 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, ESEC/FSE-7*, pages 253–267, London, UK, UK, 1999. Springer-Verlag.
- [30] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.
- [31] Z. Zhang, W. K. Chan, T. H. Tse, B. Jiang, and X. Wang. Capturing propagation of infected program states. In *ESEC/FSE '09*, pages 43–52, 2009.
- [32] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering*, 36(5):618–643, Sept. 2010.