Diagnose Crashing Faults on Production Software

Rongxin Wu
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China
wurongxin@cse.ust.hk

ABSTRACT

Software crashes are severe manifestations of software faults. Especially, software crashes in production software usually result in bad user experiences. Therefore, crashing faults mostly are required to be fixed with a high priority. Diagnosing crashing faults on production software is non-trivial, due to the characteristics of production environment. In general, it is required to address two major challenges. First, crash reports in production software are usually numerous, since production software is used by a large number of end users in various environments and configurations. Especially, a single fault may manifest as different crash reports, which makes the prioritizing debugging and understanding faults difficult. Second, deployed software is required to run with minimal overhead and cannot afford a heavyweight instrumentation approach to collect program execution information. Furthermore, end users require that the logged information should not reveal sensitive production data. This thesis contributes for developing crashing fault diagnosis tools that can be used in production environment.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging Aids.*

General Terms

Measurement, Reliability

Keywords

Crash stack, software crash, statistical debugging, software analytics

1. PROBLEM STATEMENT

Software crash is common in various kinds of software. It is one of most severe manifestations of software faults. When software crashes occur in production software, they usually result in bad user experiences and sometimes even involve serious consequences in the critical systems. Recently, crash reporting systems such as Windows Error Reporting [8], Apple Crash Reporter [2], and Mozilla Crash Reports [15] have been developed and deployed to collect crash reports from end users. When a crash occurs at a deployment site, with the permission upon end users, these systems will collect crash-related information such as the product name and version, the operating system, the crash method signature, and a crash stack. Due to the large number of users, numerous crash reports can be collected over a short period of time. For example, Mozilla [15] receives 2.5 million crash reports every day. If such crashes are not diagnosed promptly, it adversely affects the experience of many users.

Diagnosing crashing faults, especially for production software, is difficult. Especially, the characteristics of production environment make it even more challenging. First, numerous crash reports generated from a large number of end users make it impossible for developers to manually inspect each of them. Besides, due to the various environments and configurations, a single crashing fault can manifest in different ways, e.g. different crash stacks. The various manifestations of crashing faults reduce the effectiveness of effort prioritization and fault diagnosis in current crash reporting systems [8]. Due to the large volume of crash reports, developers are used to sample only some to inspect. As such, they are not able to ensure that all relevant crashes can be fixed by the patches subsequently submitted. This is a reason why many bug reports with a "fixed" status need to be re-opened afterwards when it turns out that the bugs have not been completely patched. Second, deployed software is required to run with minimal overhead, so the collected information about crashing faults should not result in an obvious runtime overhead in production software. Besides, the collected information should not expose end users' sensitive data. Current crash reporting systems mainly record crash stack in the crash reports, so that there are no additional run-time overhead and no privacy issues in crash reports. The crash stack is a snapshot of call stack at the time of crashing and captures very limited information about the failed executions. Some existing techniques [10] [11] are not efficient in diagnosing the crashing faults based on the limited information in crash report.

Considering the characteristics of production software, we aim to propose several techniques to assist the diagnosis of crashing faults based on the large number of crash reports in existing crash reporting systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2666601

2. RELATED WORK

Bucket crash: Bucketing crash reports is the process of organizing similar crash reports caused by the same bug. In production software, a large number of crash reports may be generated daily. Many of them are actually caused by the same bug and therefore are duplicate reports. Bucketing crash reports [8] is useful for developers to prioritize debugging efforts as well as understanding the crashing faults. The existing crash reporting systems [2] [8] [15] leverage some heuristics to bucket the crash reports. Due to different environments and configurations, crashes caused by same bugs (duplicate crashes) may manifest differently. As such, it is still not uncommon that duplicate crashes are spread to multiple buckets in these systems. The existence of misclassifying duplicate crash reports in these systems reduce the effectiveness of effort prioritization and fault diagnosis. Sung et al. [12] proposed to leverage call stack information to build crash graph models and identify duplicate crash reports based on graph similarity. Crash graphs aggregate all the crash stack in a bucket produced by WER and treat each function in crash stacks equally. Our proposed approach ReBucket leverages call stack information also, but measures the similarity of two call stacks using Position Dependent Model. Different from crash graphs, our similarity model treats functions differently, since the insight is that the functions near the top of the call stack are more likely to be buggy.

Reproduce crash: Reproducing crash is a debugging step to recover the dynamic information about failed executions. ReCrash [5] was proposed to generate unit tests that reproduce a given crash by capturing the state of method arguments. J. Bell et al. [6] proposed to capture the non-deterministic inputs and reproduce the crashes. However, such captured information may expose the users' sensitive data and raise the privacy issues. Jin and Orso proposed a failure reproducing tool named BugRedux [10]. BugRedux collects different kinds of execution data from end users and reproduces field failures using symbolic analysis. Experience with BugRedux shows that the call sequence (the sequence of method calls during the program execution) is the most effective data for reproducing faults, but the overhead of collecting call sequences may not still be affordable in production software. Toward this direction, we propose a technique to collect call sequence with less overhead. Compared with BugRedux, our proposed technique only instruments partial call sites and infer the rest call sites offline, which greatly reduces the runtime overhead.

Locate crashing faults: Locating crashing faults is an expensive activity in debugging. Over the years, various fault localization methods (e.g., [1] [9] [13] have been proposed. The suspicious statements in programs are ranked based on the percentages of failed and successful test cases that execute the statements. The published methods differ mainly in the type of execution information collected (e.g., statements [9] or predicates [13]), and in the way they compute suspiciousness scores [1]. However, those methods are not directly applicable to crash fault localization in production software, due to the lack of passing and failing traces. In existing crash reporting systems, only crash stack is available. The crash stacks do not contain complete successful and failed execution traces. An alternative is to employ symbolic analysis techniques generating possible test suites from program crashes, and apply the test suites at the deployment site to collect successful and failed execution traces [3] [4] . This, however, requires precise specification of all library calls and imposes difficulties on end users if program crashes can lead to serious side-effects. Furthermore, symbolic analysis is expensive and may not scale up to large programs such as Firefox. Another way to collect complete successful and failed execution traces is to deploy an instrumented version. The instrumented version could monitor program execution at the deployment site and send execution traces as well as debugging information to developers. However, such monitoring, which considerably increases execution overheads, is rarely adopted in deployed systems. Different from existing techniques, our technique CrashLocator does not require test runs and instrumentation, and it can locate crashing faults based on crash stacks only.

3. PROPOSED TECHNIQUES FOR CRASHING FAULT DIAGNOSIS

The diagnosis of crashing faults include those software quality activities towards understanding, locating, and fixing crashing faults. We summarize the possible activities of diagnosing crashing faults in Figure 1.

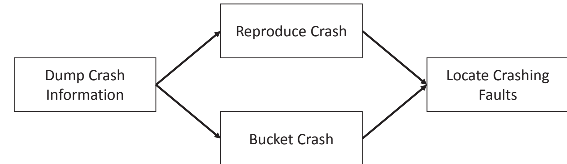

Figure 1: Activities of diagnosing crashing faults

The goal of our diagnosis tools is to locate crashing faults. Towards this goal, there are a series of activities supporting it. In this section, we propose a number of techniques to support the crashing fault diagnosis in these activities.

Dump Crash Information: Dumping crash information is the process of collecting essential information for the crashing fault diagnosis. In the existing crash reporting systems, one of the most useful information is crash stack. However, BugRedux [10] showed that, reproducing crash based on the crash stack may be difficult, while call sequence is the most effective data for reproducing faults. To collect call sequence, BugRedux simply instruments the entry of call sites and the instrumentation overhead is from 1% to 50%, on average 17.4%. Besides, call sequence is also beneficial to locate crashing faults directly. Towards the research direction, we propose a technique to collect call sequence with less runtime overhead.

We observe that the execution of some call sites is implied by the execution of other call sites. For example, if the successor call site of call site a on the control flow graph (CFG) is unique, namely b, it is unnecessary to instrument call site b because the occurrence of a in the trace implies the execution of b. This unique derivation observation can also be extended to the case where a has multiple successor call sites. Therefore, it is sufficient to instrument a small subset of the call sites to collect a partial call sequence, and infer the missed call sites offline with an linear algorithm. This auxiliary inference phase can shift a large portion of

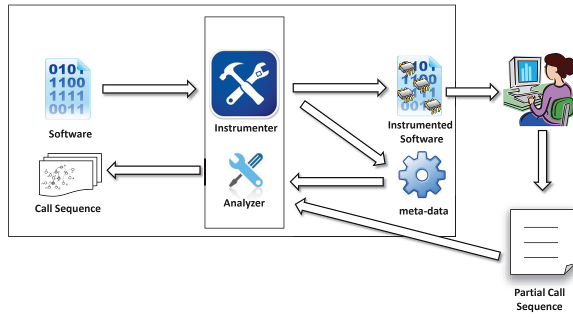

Figure 2: A framework of collecting call sequence

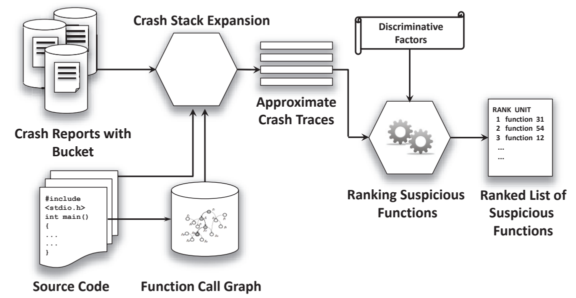

Figure 4: A framework of locating crashing faults

runtime overhead to static time. The overall process of our technique to collect call sequence is shown in Figure 2.

Bucket crash: The existing crash reporting systems leverage heuristics to bucket crash. However, the existence of misclassifying duplicate crash reports in these systems reduce the effectiveness of effort prioritization and fault diagnosis. Therefore, we propose a better crash bucketing approach ReBucket based on call stack similarity. Crash stack information is used in our technique. The overall process of ReBucket is shown in Figure 3.

Our call stack similarity measure is called Position Dependent Model (PDM), which is based on the insights that: (1) More weight should be put into frames whose position is closer to the top, since the frame that is blamed for the bug is likely to occur near the top of the call stack; (2) The alignment offset between two matched functions in two similar call stacks is likely to be small.

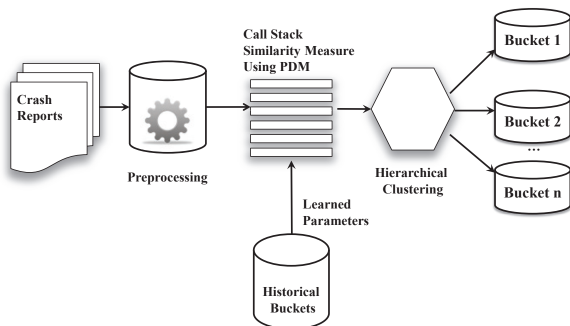

Figure 3: A framework of crash bucketing

Locate crashing faults: Due to the limited information provided in existing crash reporting system, the conventional automatic fault localization techniques may be directly applicable in production software. Therefore, we propose a novel technique CrashLocator for locating crashing faults at the function level based crash stack only. The overall process of CrashLocator is shown in Figure 4.

Different from the conventional fault localization techniques, CrashLocator does not rely on the passing and testing run. Instead, by using static analysis, it expands crash stack into the approximated crash traces so that it can cover the faulty functions that do not reside in crash stack. In this step, function call graph information is used to expand stack trace, while the control flow analysis and backward slicing are used to prune irrelevant execution traces. Then, CrashLocator computes the suspiciousness scores of all functions in the approximate crash traces, and returns a ranked list of suspicious functions. The proposed suspiciousness score is composed of discriminative factors which are based on our empirical studies in Mozilla products as well as some prior studies [16] [18]. It should be noted that, CrashLocator leverages the bucketing information, and the discriminative factors are calculated based on buckets of crash reports rather than a single crash.

Reproduce crash: Reproducing crash is important to understand and locate crashing faults. In this thesis, I do not propose a new technique to reproduce crash. However, I provide two possible extensions towards reproducing crashes. First, when dumping crash information, our technique to collect call sequence provides the fundamental data for crash reproducing. The existing study [10] has shown its effectiveness. Second, our crashing fault localization technique CrashLocator is based on crash stack only and targeting at finding faulty functions without reproducing crashes. To understand the crashing faults further and locate the faults in statement level, to reproduce crash is helpful. Although reproducing the crash [10] based on symbolic analysis is expensive and may not be effective. However, since the suspicious faulty functions are provided, reproducing partial crash stack from the faulty functions is less expensive and more feasible.

4. EVALUATION

To evaluate the technique to collect call sequence, I intend to conduct the empirical evaluations on the well-known benchmarks, as well as some real crashing faults in open source projects. I will measure both the runtime overhead and space overhead caused by the new instrumentation approach. Besides, I will validate whether the call sequence can be recovered from the partial call sequence theoretically and empirically. The outcome of this technique can be further served for other applications, such as crash reproducing and crashing fault localization. In future, we will also evaluate the usefulness of the technique.

To evaluate the proposed technique ReBucket [7], I conducted an empirical evaluations in Microsoft products. We implemented and applied our technique and some existing techniques [8] [12] [14] in Microsoft products. Our evalua-

tion results showed that, our proposed technique achieves better overall performance than the selected approaches. On average, the F-measure achieved by our approach is about 0.88. Besides, a Microsoft product team has confirmed the usefulness of 20 sampled buckets produced by our technique.

To evaluate the proposed technique CrashLocator [17] to locate crashing faults based on crash stack only , I collected 160 crashing faults in the three different Mozilla products, Firefox, Thunderbird, and SeaMonkey. In our study, CrashLocator can locate 50.6%, 63.7% and 67.5% of crashing faults by examining only top 1, 5 and 10 functions. The evaluation results show that our approach outperforms significantly the conventional methods that only examine functions in crash stack. In the future, we will evaluate CrashLocator on more projects including industrial projects.

5. PROGRESS

In the early stage of the thesis, I have implemented two techniques [7] [17] to assist the crashing fault diagnosis, based on the crash reports in the existing crash reporting systems. Both of the techniques are evaluated in the real systems, either industrial products or open source products in a large scale.

Currently, I am working on a lightweight instrumentation technique to collect call sequences in production software to assist the crashing fault diagnosis. In this thesis, I elaborated the challenges in production software. Therefore, the ongoing work should also satisfy the special demands on production software. In current stage, I have implemented a prototype of the new instrumentation technique and conducted a preliminary study on the well-known benchmarks and some crashing faults in open source projects. Later, I will validate its usefulness in the production software.

In addition, I envisioned two possible extension towards reproducing crashes, based on the my existing and ongoing work. However, the feasibilities of the extensions are needed to be validated in the future.

6. REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98. IEEE, 2007.

[2] Apple. ”technical note tn2123: Crashreporter”, developer.apple.com/library/mac/#technotes/tn2004/tn2123.html, 2010.

[3] 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*, pages 49–60. ACM, 2010.

[4] S. Artzi, J. Dolby, F. Tip, and M. Pistoia. Practical fault localization for dynamic web applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 265–274. ACM, 2010.

[5] S. Artzi, S. Kim, and M. D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP 2008–Object-Oriented Programming*, pages 542–565. Springer, 2008.

[6] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE ’13, pages 362–371, Piscataway, NJ, USA, 2013. IEEE Press.

[7] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. Rebucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 1084–1093. IEEE Press, 2012.

[8] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: Ten years of implementation and experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP ’09, pages 103–116, New York, NY, USA, 2009. ACM.

[9] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 184–193. ACM, 2007.

[10] W. Jin and A. Orso. Bugredux: reproducing field failures for in-house debugging. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 474–484. IEEE Press, 2012.

[11] W. Jin and A. Orso. F3: Fault localization for field failures. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 213–223, New York, NY, USA, 2013. ACM.

[12] S. Kim, T. Zimmermann, and N. Nagappan. Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Dependable Systems & Networks (DSN), 2011 IEEE/IFIP 41st International Conference on*, pages 486–493. IEEE, 2011.

[13] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *ACM SIGPLAN Notices*, volume 40, pages 15–26. ACM, 2005.

[14] N. Modani, R. Gupta, G. M. Lohman, T. F. Syeda-Mahmood, and L. Mignet. Automatically identifying known software problems. In *ICDE Workshops*, pages 433–441. Citeseer, 2007.

[15] Mozilla. ”mozilla crash reports”, http://crashstats.mozilla.com, 2012.

[16] A. Schroter, N. Bettenburg, and R. Premraj. Do stack traces help developers fix bugs? In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 118–121. IEEE, 2010.

[17] R. Wu, H. Zhang, S. C. Cheung, and S. Kim. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014.

[18] H. Zhang. An investigation of the relationships between lines of code and defects. In *ICSM 2009*, pages 274–283. IEEE, 2009.