Software CrashLocator: Locating the Faulty Functions by Analyzing the Crash Stack Information in Crash Reports

Divya R S[1], Pushpalatha M N[2]

[1]Dept. of Information Science Engineering, M S Ramaiah Institute of Technology (Autonomous Institute Affiliated to VTU) Karnataka, India
[2]Assistant Professor, Department of ISE, M S Ramaiah Institute of Technology, Karnataka, India

Abstract— *In recent years, studies have been dedicated mainly in the analysis, of crashes in real-world related to large-scale software systems. A crash in terms of computing can be termed as a computer program such as a software application that stops functioning properly. Software crash is a serious problem in production environment. When crash happens, the crash report with the stack trace of software at time of crash is sent to the developer team. Software development team may receive hundreds of stack traces from all deployment sites and many stack traces may be due to same problem. If the developer starts analyzing each trace, it may take a longer duration of time and redundancy many happen in terms of two developers fixing the same problem. This motivates us to present the solution to analyze the stack traces and find the important functions responsible for crash and rank them, so that development resources can be optimized. In this paper we have proposed the solution to solve the problem by developing Software CrashLocator.*

Keywords*— *Crash Locator, Windows Error Reporting, Crash Report, Mozilla Crash Reporter.

I. INTRODUCTION

Software crashes are the severe manifestation of software faults. Software Crashes are required to be fixed with a higher priority. Many crash reporting systems to name a few includes Windows Error Reporting [14], Apple Crash Reporter [2], and Mozilla Crash Reporter [25] have been proposed and deployed. These Error Reporting systems automatically collect relative information like crashed modules and crash stack) at the time of crash,later cluster the similar crash reports that are likely to be caused by the same fault into buckets (categories), and then present the crash information to the developers for debugging.

Existing crash reporting systems [2, 14, 25] focus on collecting and later bucketing crash reports. The collected crash information is mainly useful for debugging purpose, but these systems do not support automatic localization of crashing faults. As a result, for debugging crashes non-trivial manual efforts are required.

Various fault localization techniques (e.g., [1, 18, 21, 22]) over the years, have been proposed so that might help the developers to locate faults. By static analysis of both the failing and passing execution traces of test cases, these techniques suggest list of suspicious program entities. Later the developers can examine the ranked list of suspicious entities to locate faults. However, these techniques for fault localization requires complete information of passing and failing execution traces, in case of crash reports, typically contain only information of crash stacks that are dumped at the time of crashes.

In paper [26], authors proposed a novel technique named CrashLocator, for locating the crashing faults based on static analysis techniques and crash stacks. The proposed technique mainly targets an locating faulty functions as functions are commonly used in unit testing and are helpful for crash reproducing [5, 16]. In case of widely-used system, one crashing fault might result in triggering a large number of crash reports. A sufficient number of crash stacks can therefore be used by CrashLocator for locating the crashing faults. CrashLocator initially expands the crash stacks into approximate crash traces (the failing execution traces that lead to crash) using static analysis techniques including call graph analysis, backward slicing and control flow analysis. For the purpose of effective fault localization, CrashLocator applies the concept termed term-weighting [24]: locating crashing faults is treated as the problem of term weighting, i.e., calculating importance of the functions (term) for a bucket of crash traces (documents). CrashLocator considers several factors to weigh a function: the frequency of the function appearing in the bucket of crash traces, the frequency of a function appearing in the crash traces of different buckets, the distance between the crash point and function, and the size of a function. Using the listed factors, CrashLocator calculates the suspiciousness score for each function in approximate crash traces. Finally, a ranked list of suspicious faulty functions is given to developers. This helps the developers to examine the top N returned functions that helps them to locate crashing faults.

In crash locator, for calculating the score, lines of code parameter is used. This parameter is not effective to rank the functions as number of lines is not a indicator that function is error prone. Considering these problems in CrashLocator, we propose Software CrashLocator with better ranking metrics than CrashLocator.

II. RELATED WORK

In recent years, analysis of crashes of real-world, large-scale software systems, many studies have been dedicated. In order to automatically collect the crash information from field, many crash reporting system have been deployed. For example, Microsoft deployed the distributed system called Windows Error Reporting (WER) [14]. It has collected over billions of crash reports [14] during its ten years of operation. These crash reports have helped the developers diagnose problems. On receiving the crash reports, crash reporting system needs to organize these crash reports into categories. This process of organizing the similar crash reports that are caused by the same problem is often termed as bucketing [14]. Dang et al. [11] based on call stack similarity a method was proposed for finding the similar crash reports. Sung et al. [19] also proposed a method to identify the duplicate crash reports based on similarity of crash graphs.

Ganapathi et al. [13] analyzed crash data of Windows XP kernel and found poorly-written device driver code are predominant cause for OS crashes. Several methods are proposed by researchers, for reproducing the crashes. For example, ReCrash [5] a method to generate unit tests that reproduce the given crash based on captured program execution information was proposed. Csallner and Smaragdakis also proposed methods for unit test case generation for reproducing the crashes [9, 10].

The work described above mainly deals with the construction of a crash reporting system, the causes for crashes, and the reproduction of crashes. And also a focus on software crash reports analysis is done. Unlike the above described work, we also address the problem of locating crashing faults, to facilitate debugging activities

Besides statistical techniques of fault localization, many other techniques have been proposed inorder to facilitate debugging [27]. For example, consider Yoo et al. proposed Information Theory based techniques that can help reduce fault localization costs and help improve the effectiveness [25]. Zhou et al. proposed information retrieval based approach, which can help locate faulty files based on the initial bug reports. Jiang et al. [15] proposed context-aware statistical debugging method that can help not only in locating the bug but also provide faulty control flow paths. Delta debugging simplifies failed test cases and preserves the failures, producing cause-effect chains and linking them to the suspicious statements. Program slicing techniques were applied by Zhang et al. for fault localization by identifying the set of program entities that could affect the values of variables in a given program point. Artzi et al. [3, 4] proposed methods for fault localization that leverage combined concrete and symbolic executions. F. Servant and J. Jones leveraged statistical fault localization results and history of source code to assign the faults to the developers. Many inputs are required by these techniques such as test cases, complete initial bug reports and execution traces. Our approach utilizes only the crash stack information.

Liblit et al. [20, 21] proposed a sparse sampling based statistical debugging method that can reduce the overhead of instrumentation in released program. Their sampling instrumentation technique incurs less than 5% slowdown at 1/1000 sampling rate. However, as they pointed out, lower sampling rate means that more sampling traces from users are required in order to observe the rare events (i.e., the observation of faulty entity execution). Therefore their method is more suitable for popular and widely used software, while our approach only relies on crash stacks collected by a crash reporting system. Furthermore, their approach requires users to execute specially instrumented software releases, while our approach requires only the normal releases of software.

Chilimbi et al. proposed an adaptive and iterative profiling method called Holmes [8] to locate post-release faults. Holmes also considers functions in stack that are closer to the crash point as more important ones. Our approach is different in that Holmes needs to instrument the program and collect the dynamic information from end-users. Also, our approach considers more factors such as the frequency as well as the inverse bucket frequency of a function. Ashok et al. proposed a tool called DebugAdvisor [6], which can facilitate debugging by searching for similar bugs that have been resolved before. DebugAdvisor requires the users to specify their debugging context as a “fat query”, which contains all the contextual information such as bug descriptions. Unlike DebugAdvisor, our work only requires source code and crash stacks.

Jin and Orso proposed a failure reproducing tool named BugRedux [16]. BugRedux collects different kinds of execution data from end users and reproduces field failures using symbolic analysis. The exploration study of BugRedux shows that function call sequence is the most effective data for reproducing faults. To collect function call sequence, the instrumentation overhead is from 1% to 50%, on average 17.4%. Based on BugRedux, Jin and Orso also proposed the F3 approach [17] for localizing field failures. F3 uses the collected execution data to generate multiple passing and failing executions, which are similar to the observed field failures. Both BugRedux and F3 focus on failure reproduction or localization by analyzing an observed failure report one at a time. Our work targets at crashing fault localization by statistically analyzing a large amount of crash data collected from different users. Besides, our work is different from BugRedux and F3 in that our approach does not require code instrumentation and would not cause performance overhead.

III. PROBLEM DEFINITION

Given a set of stack traces and the source code, the system must find the core functions responsible for the crash and rank them in order of importance.
The solution to fixing the crashes from analyzing each trace is now translated to fixing important functions responsible for crash and it happens in most of software, same functions are responsible for many crashes. So developer effort to analyze each stack trace to fix the crash is now reduced.

I. Software CrashLoactor - Proposed Solution

The Software CrashLocator solution consists of three important modules

1. Static Analysis
2. Dynamic Analysis
3. Scoring and Ranking Functions

Static Analysis

In static analysis, the source code is taken as input and call graph is created for the source code.
Call Graph is of form
< From Classname, From Functionname, To Classname, To Functionname>

The call graph is created by visiting each class in the code and traverse each function in the class to find the class and functions invoked.

Dynamic Analysis

In dynamic analysis, each crash trace is analyzed to find the calling order of function in the stack trace.
Say below is an example stack trace.
C.fun3
B.fun2
A.fun1
The stack trace may not be complete.

Say A.fun1 has called A.fun2 which returns a output parameter and that output parameter is passed as input to B.fun2 and from there to C.fun3 and crash has happened.
Now the reason for crash is the output parameter from A.fun2 which is not covered in the stack trace as stack trace gives only the snapshot at time of crash.
To complete the stack trace, information of call graph obtained from static analysis is used to fill the uncovered functions in the stack trace

So due to dynamic analysis, complete stack trace like below is generated
C.fun3
B.fun2
A.fun2
A.fun1

Scoring and ranking functions

In this step a matrix of crash report ID vs functions is made. In this matrix, if function is covered in the stack trace of the crash report value in matrix is set as 1 else it is 0.

Table.1: The Crash Traces

	f_1	f_2	…	f_{i-1}	f_m
T_1	**1**	**0**	…	**1**	**1**
T_2	**1**	**1**	…	**1**	**1**
….	….	….	…	…	…
T_{k-1}	**1**	**1**	…	**0**	**1**
T_k	**1**	**0**	…	**0**	**1**

Each functions is given a score based on following metrics

1. Functional Frequency(FF)
2. Inverse Bucket Frequency(IBF)
3. Cyclometric Complexity(CC)
4. Inverse Average Distance to Crash Point (IAD)
5. Number of times where function is referred in static call graph.(NC)

The final score of the function is given as
FS = FF * IBF * CC* IAD * NC
FF is the function frequency. The number of times function occurred in crash.

$$FF(f,B) = \frac{N_{f,B}}{N_B}$$

It is calculated as Number of times functions appears divided by number of crash.
IBF is inverse bucket frequency.

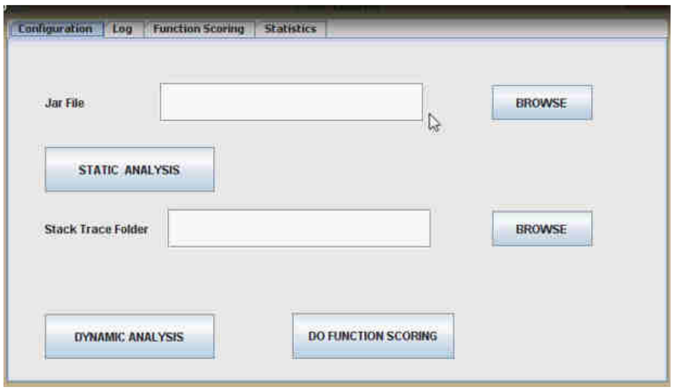
$$IBF(f) = log(\frac{\#B}{\#B_f} + 1)$$

B is the number of crash and Bf is the number of crash where function occurs.
CC is the cyclometric complexity
IAD is inverse average distance to crash points which gives the measure of distance to crash point.
NC is the number of times where function is referred in static call graph.

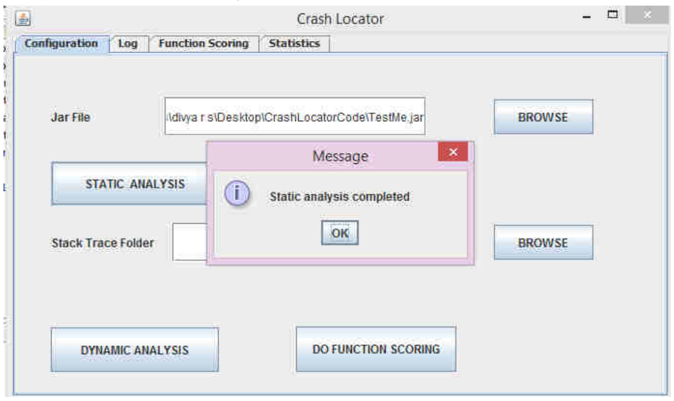
After calculating the score, the functions are ranked in descending order of score.

IV. RESULTS

Software CrashLocator solution is implemented in JAVA. The snapshots of the system is below



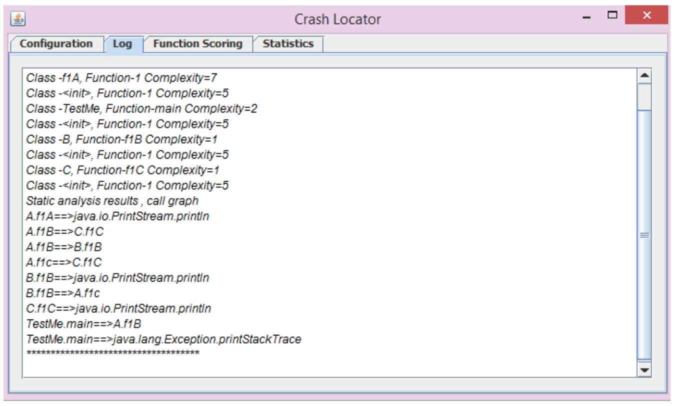
The jar files of entire source code is given as input and from this jar file, static analysis is done



\divya r s\Desktop\CrashLocatorCode\TestMe.jar

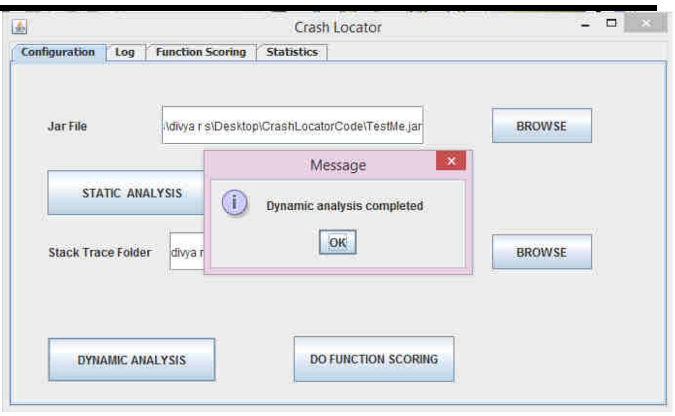
Static analysis completed

The result of static analysis is done



```
Class -f1A, Function-1 Complexity=7
Class -<init>, Function-1 Complexity=5
Class -TestMe, Function-main Complexity=2
Class -<init>, Function-1 Complexity=5
Class -B, Function-f1B Complexity=1
Class -<init>, Function-1 Complexity=5
Class -C, Function-f1C Complexity=1
Class -<init>, Function-1 Complexity=5
Static analysis results , call graph
A.f1A==>java.io.PrintStream.println
A.f1B==>C.f1C
A.f1B==>B.f1B
A.f1c==>C.f1C
B.f1B==>java.io.PrintStream.println
B.f1B==>A.f1c
C.f1C==>java.io.PrintStream.println
TestMe.main==>A.f1B
TestMe.main==>java.lang.Exception.printStackTrace
************************************
```

For dynamic analysis, the folder where all stack traces is kept is given as input

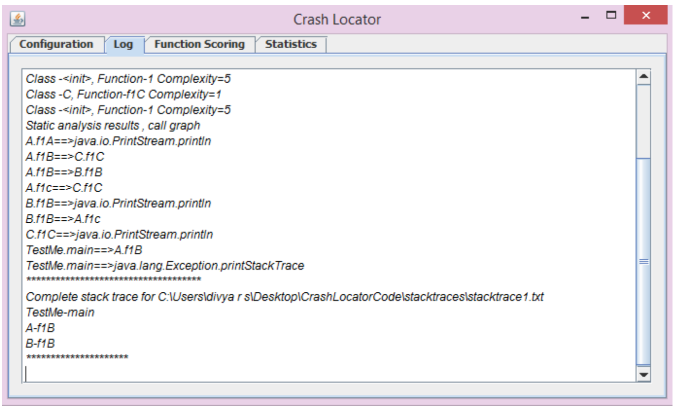


\divya r s\Desktop\CrashLocatorCode\TestMe.jar

divya r

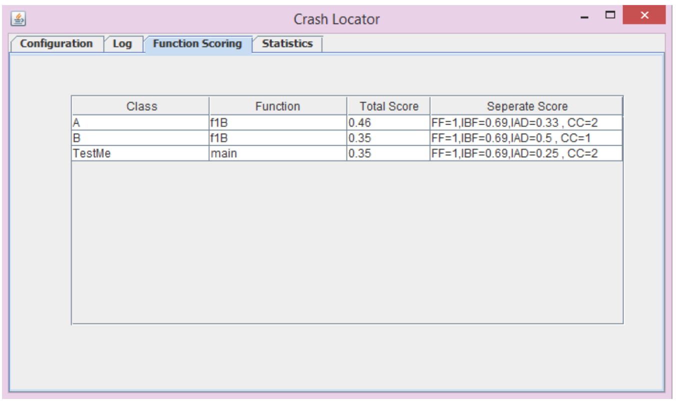
Dynamic analysis completed

After dynamic analysis, the complete stack trace is displayed.



```
Class -<init>, Function-1 Complexity=5
Class -C, Function-f1C Complexity=1
Class -<init>, Function-1 Complexity=5
Static analysis results , call graph
A.f1A==>java.io.PrintStream.println
A.f1B==>C.f1C
A.f1B==>B.f1B
A.f1c==>C.f1C
B.f1B==>java.io.PrintStream.println
B.f1B==>A.f1c
C.f1C==>java.io.PrintStream.println
TestMe.main==>A.f1B
TestMe.main==>java.lang.Exception.printStackTrace
************************************
Complete stack trace for C:\Users\divya r s\Desktop\CrashLocatorCode\stacktraces\stacktrace1.txt
TestMe-main
A-f1B
B-f1B
********************
```

After static and dynamic analysis, function scoring is done to rank the functions



Class	Function	Total Score	Seperate Score
A	f1B	0.46	FF=1,IBF=0.69,IAD=0.33 , CC=2
B	f1B	0.35	FF=1,IBF=0.69,IAD=0.5 , CC=1
TestMe	main	0.35	FF=1,IBF=0.69,IAD=0.25 , CC=2

Functions score are calculated and functions displayed in the descending order of score.
The function which appears first is most important to fix and functions which appears last is least important function to fix.

V. CONCLUSION

In this paper we have proposed the solution for finding the functions which are responsible for crash and ranking those functions by analyzing the stack trace of crash reports. Later Rank the function based on the scores obtained by using the metric listed above and reduce the developer effort in terms of analyzing each crash in fixing the functions that resulted in the occurrence of crash.

VI. ACKNOWLEDGEMENT

I thank all the faculty members of Department of Information Science, MSRIT and also thanks to the management of MSRIT, Bengaluru for providing the academic and research atmosphere at the institute.

REFERENCES

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. "On the accuracy of spectrum-based fault localization". In Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPARTMUTATION 2007), pages 89-98. IEEE Computer Society Press, 2007.

[2] Apple, "Technical Note TN2123: CrashReporter," 2010,developer.apple.com/library/mac/#technotes/tn2004/tn2123.html.

[3] S. Artzi, J. Dolby, F. Tip and M. Pistoia. Practical fault localization for dynamic web applications. In Proc. ICSE 2010, pp. 265 – 274, Cape Town, South Africa, 2010.

[4] S. Artzi, J. Dolby, F. Tip and M. Pistoia. Directed test generation for effective fault localization. In Proc. ISSTA 2010, pp. 49 – 60, Trento, Italy, 2010.

[5] S. Artzi, S. Kim, and M. D. Ernst, "ReCrashJ: a tool for capturing and reproducing program crashes in deployed applications". In Proc. ESEC/FSE'09, pp. 295-296, August 2009.

[6] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa and V. Vangala. DebugAdvisor: a recommender system for debugging. In Proc. ESEC/FSE'09, pp. 373-382, Amsterdam, The Netherlands, August 2009.

[7] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In Proc. OOPSLA, pp. 324-341, 1996.

[8] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani. "Holmes: effective Statistical Debugging via Efficient Path Profiling". In Proc. ICSE 2009, pp. 34-44, 2009.

[9] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," Softw. Pract. Exper., vol. 34, pp. 1025ʋ1050, 2004.

[10] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In Proc. ICSE 2005, pp. 422– 431.

[11] 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 Proc. ICSE 2012, pp.10841093, Zurich, Switzerland, June 2012.

[12] T. Dhaliwal, F. Khomh, and Ying Zou. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In Proc. ICSM 2011, pp. 333-342, Williamsburg, VA, USA, Sep 2011.

[13] A. Ganapathi, V. Ganapathi, and D. Patterson, "Windows XP kernel crash analysis," in Proceedings of the 20th conference on Large Installation System Administration. Washington, DC: USENIX Association, 2006, pp. 12-12.

[14] 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 Proc. SOSP 2009, Big Sky, Montana, USA, pp. 103-116, 2009.

[15] L. Jiang and Z. Su. Context-aware statistical debugging: from bug predictors to faulty control flow paths. In Proc. ASE 2007. ACM, 2007.

[16] W. Jin and A. Orso. "BugRedux: Reproducing field failures for in-house debugging." In Proc. ICSE 2012, pp. 474–484, Zurich, Switzerland, 2012.

[17] W. Jin and A. Orso. "F3: Fault Localization for Field Failures." In Proc. ISSTA 2013, pp.213-223, Lugano, Switzerland, 2013.

[18] J. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In Proc. ICSE 2002, pp. 467-477, Orlando, FL, USA, 2002.

[19] S. Kim, T. Zimmermann, and N. Nagappan, Crash graphs: An aggregated view of multiple crashes to improve crash triage, In Proc. DSN 2011, pp. 486 – 493, Hong Kong, June 2011.

[20] B. Liblit, A. Aiken, A. X. Zheng, and Michael I.Jordan. "Bug isolation via remote program sampling". In Proc. PLDI 2003, pp. 141–154, San Diego, CA, 2003.

[21] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. Jordan. "Scalable statistical bug isolation", In Proc. PLDI 2005, pp. 5-26, 2005.

[22] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. SOBER: Statistical model-based bug localization. In Proc. ESEC/FSE 05, pp. 286-295, Lisbon, Portugal, 2005.

[23] C. Luk , R. Cohn , R. Muth , H. Patil , A. Klauser , G. Lowney , S. Wallace , V. J. Reddi , and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation", In Proc. PLDI 2005, pp. 190-200, Chicago, Illinois, USA, June 2005.

[24] C. D. Manning, P. Raghavan and H. Schütze. Introduction to Information Retrieval, Cambridge University Press, 2008.

[25] Mozilla, "Mozila Crash Reports," 2012, http://crashstats.mozilla.com.

[26] [26] Rongxin Wu "CrashLocator: Locating Crashing Faults Based on Crash Stacks " ACM 2014