Challenges and Issues of Mining Crash Reports

Le An, Foutse Khomh
SWAT, Polytechnique Montréal, Québec, Canada
{le.an, foutse.khomh}@polymtl.ca

***Abstract*—Automatic crash reporting tools built in many software systems allow software practitioners to understand the origin of field crashes and help them prioritise field crashes or bugs, locate erroneous files, and/or predict bugs and crash occurrences in subsequent versions of the software systems. In this paper, after illustrating the structure of crash reports in Mozilla, we discuss some techniques for mining information from crash reports, and highlight the challenges and issues of these techniques. Our aim is to raise the awareness of the research community about issues that may bias research results obtained from crash reports and provide some guidelines to address certain challenges related to mining crash reports.**

***Index Terms*—Crash report, bug report, mining software repositories.**

I. Introduction

Nowadays, crash reporting tools are embedded in many software systems to collect information about crashes in the field (*i.e.*, when a program stops functioning properly in a user environment). Crashes with the same crashing signature, the stack trace of the failing thread, will be grouped into a *crash type*. Usually, software quality managers prioritise crash types by the number of crash occurrences, then file the top crash types into bug reports. Crash reports provide useful reference to analyse and resolve bugs. They could help software practitioners locate erroneous code, understand the impact of failures, and prioritise crash-related bug reports.

Leveraging crash reports, previous studies propose new ideas to improve the current software maintenance techniques. Khomh et al. [1] analysed crash reports of Mozilla Firefox and proposed an entropy metric that reveals the distribution of the occurrences of crashes among end users. Wang et al. [2] studied the crash reports of Firefox and stack traces of Eclipse. They proposed five rules to automatically identify correlated crash types. In a recent study, we also analysed crash reports from Mozilla products to identify bugs that affect a large population of users [3].

However, some inherent properties of crash reports may challenge the research conclusions if researchers and software practitioners do not take care of them when they mine information from crash reports. For example, user information is not always available in crash reports, and the many-to-many relationship between crashes and their related bugs increases the difficulty of analysis. Furthermore, since there are not enough publicly opened crash collecting databases, researchers can hardly estimate the latent peril due to some of these issues (*e.g.*, lack of precise users' information) or validate the generalisability of their findings.

In the rest of this paper, we describe the structure of a crash report before presenting some analytic techniques to mine implicit information from crash reports. For each of the techniques, we discuss its challenges and issues. After giving an overview of the related work, we conclude the paper with suggestions for future studies.

II. Structure of Crash Reports

In this section, we take crash reports from Mozilla Socorro server (Mozilla's crash collecting database) [4] as an example to describe the typical crash collecting system.

Mozilla delivers its applications with a built-in automatic crash reporting tool: Mozilla Crash Reporter, which sends a crash report to the Mozilla Socorro server, once end users encounter an unexpected halt of the application. Each crash report records a stack trace of the falling thread and other information about the user's environment. A stack trace is an ordered set of frames where each frame indicates a method signature and a link to the corresponding source code. Source code information is not always available in the frames, especially when a frame belongs to a third party binary [1].

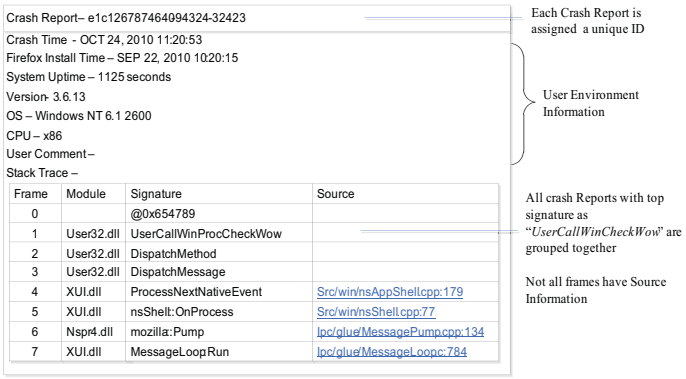

Fig. 1: A sample crash report from Firefox

Figure 1 shows a sample crash report from Mozilla Firefox. Among all information provided in a crash report, researchers and software practitioners may be interested in the following attributes when they study crashes for software maintenance:

- *crash signature*: top method signature of a crash. Crashes with the same crash signature are grouped into one crash type in the Socorro server [1].
- *Install age*: the time in seconds from the installation until the crash occurred.
- *Crash date*: the point of time when the crash was reported.

- *OS name*: name of the operating system on which the crash occurred.
- *OS version*: version of the operating system on which the crash occurred.
- *CPU type*: family model of the CPU of a machine on which the crash occurred.
- *Bug list*: list of bugs related to the crash.
- *Up time*: the amount of seconds since the application startup.

Software researchers and practitioners can refer to the Socorro documentation [5] for the description of other attributes.

III. Analysis of Crash Reports and Challenges

In this section, we describe some analytic techniques of crash reports for software maintenance and discuss their challenges and issues.

A. User identification

When we study crash reports, an important process is to identify unique users who encounter crashes in a software system. Mapping crashes to different users can help software practitioners understand the dispersion of these crashes in the user base and determine the priority of related crash types and bugs. However, user identity is not always available. For example, Mozilla crash reports do not contain personal information indicating unique users reporting the crashes due to privacy concerns. Khomh et al. [1] used a heuristic where they took installed point of time (subtraction of crash date from install age) of the studied system to identify different users (noted as *install profiles*). In our recent work [3], which consisted in studying the crashing impact of bugs on the user population base, we used a vector of computer configuration (combination of CPU type, OS name, and OS version) to represent different "users" (noted as *machine profiles*) in a system.

However, these user identification heuristics may lead to inconsistent results. For example, when we apply the heuristic by install profile, two users who install an application at the same point of time (by second) may encounter quite different types of crashes. When we apply the heuristic by machine profile, users with the same computer configuration may also exhibit different crashing characteristics. In other words, users that happened to install a software at the same time or in the same model of computer, especially for enterprise users whose computers are uniformly configured or software is simultaneously installed, may behave differently, and should not be clustered together in terms of crash analysis.

B. Mapping between crashes and bugs

When we study the characteristics of bugs related to field crashes, we should map crash reports to their corresponding bugs in order to understand the distribution of crash-related bugs in the user base. Using the information provided in crash reports and bug reports, there are two general ways to link related bugs and crashes.

Algorithm 1: Map different crashed users to a crash type, and map different crash types to a bug

```
Input: signature, user, buglist
if signature in dict_crash then
    stack_user ← dict_crash[signature]
    add user to stack_user
else
    stack_user ← new stack with user
    dict_crash[signature] ← stack_user
foreach bug in buglist do
    if bug in bug_dict then
        set_signature ← dict_bug[bug]
        add signature to set_signature
    else
        set_signature ← new Set with signature
        dict_bug[bug] ← set_signature
```

Algorithm 2: Map crashed user occurrences to their bugs

```
Input: dict_bug
foreach bug in dict_bug do
    set_signature ← dict_bug[bug]
    foreach signature in set_signature do
        stack_user ← dict_crash[signature]
        concatenate stack_user to occur_user
```

On the one hand, we can use the bug lists indicated in crash reports to directly map crashes to bugs. However, some crashes are reported before the opening of their corresponding bugs. If we apply the direct mapping only, we may omit the information about these crashes.

On the other hand, we can also perform an indirect mapping from crashes to bugs via crash types. In other words, we perform a two-step mapping as following: we map each bug to a set of crash type (*i.e.*, a group of crashes with the same crash signature), then map each crash type to a set of crash reports. Concretely, we build two hash tables, $dict_{bug}$ and $dict_{crash}$. In any item of $dict_{bug}$, the key is a bug ID and the value is a set of crash signatures. In any item of $dict_{crash}$, the key is a crash signature and the value is a stack[1] of crash reports. By associating the two hash tables, we can finally map a bug to a set of corresponding crash reports, even including those lacking bug information (*i.e.*, crashes reported before the creation of the bugs). Similarly, to investigate the crash distribution in the user base, we can also link different users (represented by install profiles or machine profiles) to a bug by analysing the bug's related crash reports. Algorithm 1 and 2 show the pseudocode to link a bug to the users suffering its related crashes.[2] However, in a software system, a crash report may bring about several bugs, a bug may be also related to more than one crashes.

[1] Every new crash report / crash occurrence will be appended at the end of the structure.

[2] The mapping script in Python and example data are available in the following repository: http://swat.polymtl.ca/anle/data/SWAN2015/

The many-to-many relationship increases the difficulty to map a bug to its corresponding crash reports. Compared with the direct mapping, the concatenation operations (in Algorithm 2) in the indirect mapping require more execution time. Usually, crash analysis involves a big dataset (*e.g.*, Mozilla receives 2.5 million crash reports on the peak day each week, namely, around 50GB of data need to be processed every day [6]), reducing time complexity of the mapping technique can help researchers and software practitioners improve their analytic efficiency to early understand the impact of the crash-related bugs.

To mitigate the above mentioned problems, we can combine the direct and indirect mapping techniques together. In other words, if the bug list is available in a crash report, we will use the direct mapping; otherwise, the indirect mapping will be applied. The combined approach can effectively reduce the mapping algorithm's time complexity while linking a bug to all its possible crash reports. Besides, according to our case study [3], direct mapping and indirect mapping may lead to results with subtle difference. Suppose a bug B is due to a crash C_1 pertaining to the crash type CT. Another crash C_2 also classified as type CT might not be related to the bug B. Because most software organisations, such as Mozilla, group crashes with the same "top frames" (in a stack trace of failing thread) into a crash type. When a stack trace contains a lot of frames, different software organisations may extract different numbers of "top frames" as crashing signatures. Different criteria for the selection of the "top frames" may result in different distributions of crash types, and translate into different mappings between crashes and bugs. Therefore, to make sure the mapping results with a high precision, efficiency, and completeness, we suggest that software practitioners prioritise the direct mapping before resorting to the indirect mapping using crash types.

C. Relationship between crashes and bug fixes

Furthermore, we can also map crash reports to bug fixes to investigate whether developers efficiently addressed reported field crashes and estimate the effort required to fix the crashes. There are two ways to link a crash type to its corresponding bug fixes, via bug reports and by text analysis. In our previous work, we described a technique to map fixing commits to bug reports [7]. So we can use bug reports as an intermediary to link crashes and commits. On the other hand, if a bug contains different crash types that are then related to many bugs, it is better to directly link a crash type (in the crash collecting system) to bug fixes (in the version control system) by comparing the stack trace of the failing thread in crash reports and the revised code in bug fixes. Namely, we check whether the crash-related code (or in a coarse level, the file of this code) is revised in any revisions. Linking a crash to its revisions, we can compute the fixing duration and the number of involved developers of the crash, *i.e.*, the crash's fixing effort.

Although bug fixing commits could be used to assess the effort (or estimated effort) required to fix a crash type, some factors may challenge the results of this approach. Especially, if we use bug reports to link a crash type and its corresponding bug fixes, some kinds of bugs may "exaggerate" the required effort of a crash type. For example, some re-opened bugs, *i.e.*, bugs that have been opened more than once, may be due to an inactive attitude of software developers (*i.e.*, these bugs have been prematurely closed) [7]. The bug fixes ported before the re-opening to these bugs should not be taken into consideration when computing the required effort. Moreover, crashes due to non-reproducible bugs [8] may bias effort estimations as well, because these bugs only show up on certain machines (from certain users), and do not affect other users. Joorabchi et al. [8] found that many non-reproducible bugs are mislabelled, because the available resolutions in the repositories do not cover all possible scenarios. These "defects" are mainly due to wrong configuration of a software, and should not have been classified as bugs. If software managers compute the fixing effort of the false positive bugs as other bugs, they may overestimate the trivial issues while omitting the serious ones.

D. Localisation of buggy files

Crash reports can help for fault localisation as well. There exist plenty of studies on automatic bug localisation. For example, Liblit et al. [9] studied predicate patterns in correct and incorrect execution traces. They proposed an algorithm to separate the effects of different bugs and identify predictors associated with individual bugs. Nessa et al. [10] introduced a bug localisation algorithm based on N-gram analysis to rank the executable statements of a software by level of suspicion. Wang et al. [2] proposed an algorithm based on crash correlation groups to locate and rank buggy files by analysing the stack traces in correlated crash types. Crash reports provide failing source code location in their stack traces, but this information is not always complete (*e.g.*, in the sample shown in Figure 1, source code information is not available in some frames). There are generally three types of source code information given in a crash report. In the best case, buggy files' path are directly indicated in the frames of a stack trace. Sometimes, crashed method signatures are provided in the stack trace. We should iteratively search the signatures in every file of source code repository to locate the possible buggy files. In the worst case, just crashing memory address or a textual description is given in the frames from which we can hardly locate the source of a bug.

Nevertheless, we should not only rely on crash reports to locate erroneous code, because this technique is merely applicable to crash-related bugs while not all crash reports contain related files' location or crashed methods. Bug fixes are another source to discover buggy files. We can apply the heuristic described in [7] to link a bug to its related bug fixes then identify the changed files in the bug fixes. In our case study with Mozilla, Eclipse, Netbeans, and Webkit, we found that bug reference is not always available in bug fixes, *i.e.*, some bugs may not be able to get linked to their bug fixes by this heuristic. Therefore, crash reports could be used as a complementary source to detect buggy file locations.

E. Suggestions

When conducting empirical studies with crash report data, researchers should pay attention to the above mentioned issues. Before drawing any conclusion, the threats to validity (*e.g.*, user identification technique) should be carefully discussed. In addition, different software organisations may design their crash reports in different format and structure. At the time of writing this paper, too few crash collecting databases have been publicly available to researchers. Mozilla Socorro is the only source that we can explore for a pertinent case study.

We hope that more software organisations could share their crash collecting databases to allow researchers to verify the generalisability of our proposed approaches as well as other related work and verify whether the above mentioned issues are relevant in a larger context (*i.e.*, beyond the case of Mozilla). By analysing crash reports from different organisations, we can also propose ideas to improve the current structure of crash reports and crash collecting systems to help software organisations augment their productivity, user satisfaction, and reduce their maintenance effort.

IV. Related Work

In this section, we introduce some related work on mining crash reports.

Podgurski et al. [11] introduced an automated failure clustering approach for the classification of crash reports, in order to facilitate their prioritisation and the diagnostic of their root causes. By mining crash reports in Mozilla Firefox, Khomh et al. [1] proposed an entropy-based approach that can be used to identify crash types by not only their crashing frequency but also their crashing dispersion among users. Inspired by the work of Khomh et al., Wang et al. [2] analysed crash information in Firefox and Eclipse, and proposed an algorithm that can be used to locate and rank buggy files as well as a method to identify duplicate and related bug reports. Kim et al. [12] studied crash reports and impacted source files in Firefox and Thunderbird to predict top crashes before a new release of a software.

Most of these researchers used Mozilla Socorro crash reporting system as a subject system. Because, as far as we know, only the Mozilla Foundation has opened the crash reports to the public [2]. Though Wang et al. [2] studied another system, Eclipse, they could obtain crash information only through the stack traces contained in the bug reports (instead of using crash reports). However, stack trace information is not always available in bug reports for the majority of software systems. Dang et al. [13] proposed a method, ReBucket, to improve the current crash reports clustering technique based on call stack matching. But their subject crash collecting database, Microsoft's Windows Error Reporting (WER) system, is not accessible for every researcher.

V. Conclusion and Future Work

Previous studies on mining crash reports proposed approaches to improve crash triaging, locate buggy files and duplicate bug reports, as well as predict top crashes in future versions of a software system. However, researchers still have to face some challenges and issues when studying field crash reports: such as the lack of precise information about users or buggy files in crash reports, the many-to-many mapping between bugs and crash reports, which makes it difficult to link a crash to its bug fixing commits, and even the lack of open source crash report databases. These issues may challenge the conclusions drawn from these studies and should be carefully taken into account when conducting empirical studies in the future. Especially, the last issue greatly limits the generalisation of the ideas developed in previous studies. In this paper, we illustrate the structure of crash reports in Mozilla, then describe some crash analysis techniques and discuss their corresponding problems. We appeal that more software organisations share their crash report databases for future researchers in order to improve our knowledge of the problems. Various subject systems can also provide different perspectives to help crash report designers improve current crash reporting systems and further help software practitioners improve the satisfaction of their end users.

References

[1] F. Khomh, B. Chan, Y. Zou, and A. E. Hassan, "An entropy evaluation approach for triaging field crashes: A case study of mozilla firefox," in *Reverse Engineering (WCRE), 2011 18th Working Conference on*. IEEE, 2011, pp. 261–270.

[2] S. Wang, F. Khomh, and Y. Zou, "Improving bug management using correlations in crash reports," *Empirical Software Engineering*, pp. 1–31, 2014.

[3] L. An and F. Khomh, "An empirical study of highly-distributed bugs in mozilla firefox," École Polytechnique de Montréal, Tech. Rep., November 2014. [Online]. Available: http://swat.polymtl.ca/anle/technicalreports/highlydistributed.pdf

[4] "Socorro: Mozilla's crash reporting system," accessed 6th January, 2015. [Online]. Available: https://crash-stats.mozilla.com/home/products/Firefox

[5] "Socorro documentation," accessed 6th January, 2015. [Online]. Available: http://socorro.readthedocs.org/en/latest/index.html

[6] "Socorro: Mozilla's Crash Reporting Server," accessed 6th January, 2015. [Online]. Available: http://blog.mozilla.org/webdev/2010/05/19/socorro-mozilla-crash-reports/

[7] L. An, F. Khomh, and B. Adams, "Supplementary bug fixes vs. reopened bugs," in *Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Victoria, BC, Canada, September 2014.

[8] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah, "Works for me! characterizing non-reproducible bug reports," in *Proceedings of the 11th Working Conference on Mining Software Repositories*. ACM, 2014, pp. 62–71.

[9] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 15–26.

[10] S. Nessa, M. Abedin, W. E. Wong, L. Khan, and Y. Qi, "Software fault localization using n-gram analysis," in *Wireless Algorithms, Systems, and Applications*. Springer, 2008, pp. 548–559.

[11] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang, "Automated support for classifying software failure reports," in *Software Engineering, 2003. Proceedings. 25th International Conference on*. IEEE, 2003, pp. 465–475.

[12] D. Kim, X. Wang, S. Kim, A. Zeller, S.-C. Cheung, and S. Park, "Which crashes should I fix first?: Predicting top crashes at an early stage to prioritize debugging efforts," *Software Engineering, IEEE Transactions on*, vol. 37, no. 3, pp. 430–447, 2011.

[13] 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*. IEEE Press, 2012, pp. 1084–1093.