Predicting Recurring Crash Stacks

Hyunmin Seo and Sunghun Kim
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
{hmseo, hunkim}@cse.ust.hk

ABSTRACT
Software crash is one of the most severe bug manifestations and developers want to fix crash bugs quickly and efficiently. The Crash Reporting System (CRS) is widely deployed for this purpose. Even with the help of CRS, fixes are largely by manual effort, which is error-prone and results in recurring crashes even after the fixes. Our empirical study reveals that 48% of fixed crashes in Firefox CRS are recurring mostly due to incomplete or missing fixes. It is desirable to automatically check if a crash fix misses some reported crash traces at the time of the first fix.

This paper proposes an automatic technique to predict recurring crash traces. We first extract stack traces and then compare them with bug fix locations to predict recurring crash traces. Evaluation using the real Firefox crash data shows that the approach yields reasonable accuracy in prediction of recurring crashes. Had our technique been deployed earlier, more than 2,225 crashes in Firefox 3.6 could have been avoided.

Categories and Subject Descriptors
D.2.5 [Software Engineering]: Testing and Debugging—debugging aids, tracing

General Terms
Management, Reliability, Verification

Keywords
Crash, crash reporting system, bug

1. INTRODUCTION
Software crash is one of the most obvious and severe bug manifestations. Crashes immediately stop software execution and often cause data loss. If a crash happens in a critical part of the operating system kernel, the entire computer may stop working and require rebooting. Researchers and developers have put in significant efforts to fix crash bugs quickly and efficiently. The Crash Reporting System (CRS) is an outcome of efforts made in this context. Many CRSs such as Windows Error Reporting [27], Mozilla Crash Reporting System [25] and Apple Crash Reporter [2] have been widely deployed and are actively used in practice.

When a crash is detected on the client side, CRS generates a crash report by collecting crash related data and sends it to a server maintaining all crash reports. The report typically includes crash point and stack traces [22]. Since often too many crashes are reported [22, 23], similar crash traces are grouped together and put into a crash bucket [12, 17]. Then, developers focus on top crash buckets which contain the most frequent crashes [15]. A bug report is filed for top crashes and is linked to the corresponding crash bucket. Crash traces in the same bucket are investigated to localize and fix the crash.

Even with the help of CRS, fixing crashes largely relies on manual effort and this process is error prone. Gu et al. found that bad fixes account for as much as 9% of all bugs [14]. Yin et al. investigated post-release bug fixes in three large operating systems and found that at least 14.8%~24.4% of them were incomplete [28]. In addition, our empirical study revealed that 47% of fixed crashes in Firefox 3.6 are recurring, and many of them are due to incomplete or missing fixes.

Figure 1 shows an example of a recurring crash. Two crash traces are collected from one crash bucket in Firefox 3.6b1. A developer made a patch to fix this crash in Firefox 3.6b2 by changing the code in function nsTextToSubURI::UnEscapeAndConvert, shown in trace a. However, the developer missed a crash bug in trace b. As a result, the same crash recurred, following trace b. Unfortunately, this crash became one of the top crashes in the official release of Firefox 3.6. It was re-fixed in 3.6.4 by changing function nsCacheEntryDescriptor::GetDeviceID in trace b. After the second fix, the crash disappeared.

To avoid recurring crashes, it is desirable to automatically check if all reported crash traces in a bucket are covered by a crash fix. If any crash traces are not covered by the proposed fix, the crash may recur due to the missed traces as shown in Figure 1.

In this paper, we propose a technique to predict recurring crash traces. From crash traces in a crash bucket and a proposed fix for the bucket, we automatically check if the fix covers all crash traces in the bucket. We first extract unique stack traces from the bucket and expand the traces based on program call relations. Then, we compare the original and expanded traces with crash fix locations to identify missing traces.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE’12, September 3–7, 2012, Essen, Germany
Copyright 2012 ACM 978-1-4503-1204-2/12/09 ...$15.00
2. RECURRING CRASHES

This section reports the results of our empirical study on recurring crashes. We first introduce CRS and explain CRS related terminologies. We use Mozilla’s crash reporting system (MCRS) [6,22,25] as a typical CRS example. Then we present statistics of recurring crashes and discuss the reasons for recurring crashes.

2.1 Mozilla Crash Reporting System

When a crash occurs on the client side, the CRS generates a crash report and sends it to a server. For example, MCRS generates a crash report including crash point, crash time, product version, operating system and its version, hardware information, optional user comments and all thread stack traces from the crash. The crash report is sent to the MCRS server.

On the server side, crash reports having similar crashes are grouped together and put in the same crash bucket (Figure 2). MCRS’s grouping is based on the crash point. That is, crash reports having the same crash point are grouped together and put into the same crash bucket. Then, developers investigate the crash buckets to fix them. Developers usually focus on top crash buckets first [12, 15]. Each crash bucket is linked to a bug report to monitor and track the progress of crash fixes (Figure 2).

Table 1: Terminologies used in the paper

<table>
<thead>
<tr>
<th>Name</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>crash report</td>
<td>When a crash occurs on the client side, a crash report will be generated and sent to a CRS server. A crash report includes crash related information such as stack traces and a crash point.</td>
</tr>
<tr>
<td>crash point</td>
<td>The crashed location. It consists of file name, function name, and line number.</td>
</tr>
<tr>
<td>stack traces</td>
<td>A list of stack frames captured at the crash moment. Each stack frame corresponds to information of a function including the function name, source file name, and line number. A stack trace shows the function call sequence at the moment of crash.</td>
</tr>
<tr>
<td>crash bucket</td>
<td>A group of crash traces. Similar traces are put in the same bucket. Mozilla crash traces are grouped according to their crash point.</td>
</tr>
<tr>
<td>bug report</td>
<td>Bug reports fixing crash bugs are linked to the crash buckets. Developers communicate via comments in bug reports. The status of a fix can be checked via the comments and the history of bug report resolutions.</td>
</tr>
<tr>
<td>crash fix</td>
<td>When a bug is fixed, a patch file is usually attached to the bug report. Once reviewers accept the patch, it is reflected to the source code, and the next release contains the patch.</td>
</tr>
</tbody>
</table>
week period in March 2011, we identified crash reports and crash buckets from nineteen versions of Firefox 3.6, including five beta versions. Then, from the bug reporting system, we identified the corresponding bug reports linked to these crash buckets. Among the identified bug reports, we used only the reports in FIXED status for our study. In total, we collected 70 bug reports. These reports were corresponding to bugs in 79 crash buckets that had been fixed\textsuperscript{1}.

Then, we checked whether crashes in the crash buckets had disappeared after the fixes. For 79 crashes, we identified the number of crashes reported in each version. We also identified the version where the crash was fixed. By looking at the number of crash reports in different versions before and after the fixes, we manually checked if the crashes had disappeared.

Table 2 lists three example cases. The first column shows bug report IDs and the second column shows crash points. The next columns show the number of crash reports at different versions. Gray color cells represent the version where the crashes were fixed.

<table>
<thead>
<tr>
<th>Bug Id</th>
<th>Crash Point</th>
<th>Ver 1</th>
<th>Ver 2</th>
<th>Ver 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>538722</td>
<td>nsHtml5ElementName::initializeStatics</td>
<td>3.6.8</td>
<td>3.6.9</td>
<td>3.6.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>677</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>554544</td>
<td>nsTextFrame::reflow</td>
<td>3.6.6</td>
<td>3.6.7</td>
<td>3.6.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>773</td>
<td>186</td>
<td>497</td>
</tr>
<tr>
<td>528311</td>
<td>nsXULTreeAccessible::GetTreeItemAccessible</td>
<td>3.6b3</td>
<td>3.6b4</td>
<td>3.6b5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>168</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: Number of crash reports in three different versions for three crash fixes. Gray color cells represent bug fixed versions.

For 79 crashes, we identified the corresponding bug reports linked to these crash buckets. Among the identified bug reports, we used only the reports in FIXED status for our study. In total, we collected 70 bug reports. These reports were corresponding to bugs in 79 crash buckets that had been fixed\textsuperscript{1}.

Then, we checked whether crashes in the crash buckets had disappeared after the fixes. For 79 crashes, we identified the number of crashes reported in each version. We also identified the version where the crash was fixed. By looking at the number of crash reports in different versions before and after the fixes, we manually checked if the crashes had disappeared.

Table 2 lists three example cases. The first column shows bug report IDs and the second column shows crash points. The next columns show the number of crash reports at different versions. Gray color cells represent the version where the crashes were fixed.

<table>
<thead>
<tr>
<th>Bug Id</th>
<th>Crash Point</th>
<th>Ver 1</th>
<th>Ver 2</th>
<th>Ver 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>538722</td>
<td>nsHtml5ElementName::initializeStatics</td>
<td>3.6.8</td>
<td>3.6.9</td>
<td>3.6.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>677</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>554544</td>
<td>nsTextFrame::reflow</td>
<td>3.6.6</td>
<td>3.6.7</td>
<td>3.6.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>773</td>
<td>186</td>
<td>497</td>
</tr>
<tr>
<td>528311</td>
<td>nsXULTreeAccessible::GetTreeItemAccessible</td>
<td>3.6b3</td>
<td>3.6b4</td>
<td>3.6b5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>168</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: Number of crash reports in three different versions for three crash fixes. Gray color cells represent bug fixed versions.

For 79 crashes, we identified the corresponding bug reports linked to these crash buckets. Among the identified bug reports, we used only the reports in FIXED status for our study. In total, we collected 70 bug reports. These reports were corresponding to bugs in 79 crash buckets that had been fixed\textsuperscript{1}.

Then, we checked whether crashes in the crash buckets had disappeared after the fixes. For 79 crashes, we identified the number of crashes reported in each version. We also identified the version where the crash was fixed. By looking at the number of crash reports in different versions before and after the fixes, we manually checked if the crashes had disappeared.

Table 2 lists three example cases. The first column shows bug report IDs and the second column shows crash points. The next columns show the number of crash reports at different versions. Gray color cells represent the version where the crashes were fixed.

<table>
<thead>
<tr>
<th>Bug Id</th>
<th>Crash Point</th>
<th>Ver 1</th>
<th>Ver 2</th>
<th>Ver 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>538722</td>
<td>nsHtml5ElementName::initializeStatics</td>
<td>3.6.8</td>
<td>3.6.9</td>
<td>3.6.10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>677</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>554544</td>
<td>nsTextFrame::reflow</td>
<td>3.6.6</td>
<td>3.6.7</td>
<td>3.6.8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>773</td>
<td>186</td>
<td>497</td>
</tr>
<tr>
<td>528311</td>
<td>nsXULTreeAccessible::GetTreeItemAccessible</td>
<td>3.6b3</td>
<td>3.6b4</td>
<td>3.6b5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>70</td>
<td>168</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 2: Number of crash reports in three different versions for three crash fixes. Gray color cells represent bug fixed versions.

Figure 3 shows two different stack traces in the same crash bucket. The functions at the third stack frame are different.

We sub-grouped crash reports in the same bucket according to their respective crash stack traces. Then, we counted the number of crash reports in each sub-group. Figure 3(a) is an example of crash nsXULTreeAccessible::GetTreeItemAccessible. It shows the number of crash reports for five different sub-groups in the same crash bucket in 3.6b3, before fix #528311 is released. We excluded three sub-groups having only one crash report.

We counted the number of crash reports after a fix in 3.6b4 in each sub-group in Figure 3(a) to check if they had disappeared. Figure 3(b) shows the result. Interestingly, all other sub-groups have disappeared but crashes in sub-group #2 recurred. In fact, crashes in sub-group #2 account for 98% (164/168) of all crash reports in the corresponding bucket in Firefox 3.6b4.

When developers fix a crash bucket, sometimes the fix misses certain crash traces in the bucket. As a result of such incomplete fixes, some of the crash traces in the fixed crash bucket may recur, as shown in Figure 3(b). This is one of the main reasons of recurring crashes.

Bug report #528311 confirms that the first fix was indeed incomplete. A developer modified function nsXULTreeAccessible::GetChildAt in the first fix. All stack traces which include this function disappeared, while traces not including this function recurred in the next release. Another developer mentioned in the bug report that the crash had not disappeared in 3.6b4, and the first developer fixed a function in the recurring stack trace. After the second fix, the crash disappeared.

With thousands of crash reports in a bucket, it is easy to miss some and end up with incomplete fixes, as shown in Figure 3(b). A comment in another bug report #523528 clearly shows another example of a crash recurring due to an incomplete fix. After observing that crash imgFrame::Draw is recurring after the first fix, the developer re-fixed it and left a comment:

\begin{quote}
"I don't know how this bit (crash trace) got lost from the patch I ended up checking in, but it's pretty essential..."
\end{quote}

To check if these missing stack traces can be recognized by developers immediately, we measured the time lag between the first incomplete fixes and the corresponding follow-up fixes. Unfortunately, we found the average time was 23 days. This indicates that these missing stack traces are not something developers catch immediately after the first incomplete fixes. Thus, this issue is non-neglectable.

To assist developers and prevent such incomplete crash fixes, we propose an automatic technique to predict recurring crash traces.

\footnote{Some bug reports fix more than one crash bucket.}
3. APPROACH

Since developers may miss some stack traces in the fixes, our approach tries to identify them when a fix is made, to prevent potential recurring crashes. We present the overview of our approach first, and explain each component in detail.

3.1 Overview

Figure 4 shows the overview of our approach. First, crash traces in the same crash bucket are sub-grouped according to their traces. From a given crash fix, the fix locations are identified. Then, we compare sub-grouped crash traces with bug fix locations. During this comparison, traces are expanded using their call relations acquired from static analysis of source code. If we find crash fix locations in original or expanded stack traces, we classify these traces as covered. If we cannot find fix locations in original or expanded stack traces, they are classified as missing. Missing crash traces are predicted to recur, which requires developers’ attention.

3.2 Grouping Crash Traces

The example cases in Figure 1 and Figure 3 show that developers miss some stack traces in the same crash bucket. To find such traces, we sub-group crash traces in a crash bucket. Two traces are grouped together when all stack frames of the traces are matched. We consider function name, source file, and line number of each frame for matching. This matching is quite strict, but other heuristics [7, 12, 20] for grouping crash traces can be applied in this process.

Firefox is a multi-threaded program having separate stack traces for each thread. We consider only the thread containing the crash point and use the stack trace of that thread for sub-grouping.

After the sub-grouping, we check if a fix for the bucket covers all sub-groups in the bucket.

3.3 Identifying Crash Fixes and Locations

We use bug fix locations to check if there are any missing traces in the crash bucket. Fix locations can be extracted from patch files attached to bug reports, or developers can specify them.

We use the links between crash buckets and bug reports to identify the corresponding crash fixes. As explained in Section 2.1, each crash bucket has links to the corresponding bug reports.

From the linked bug reports we extract patch files attached to the bug reports. We verify whether the fix code in the patch files had actually been released. We downloaded the source code of Firefox release versions and manually compared them with code in patch files to check if the changed code was really included in the releases.

After identifying and verifying fixes linked to crash buckets, we extract fix locations in functional levels. Patch files include the changed code with file names and line numbers. By looking at the source code corresponding to file names and line numbers, we acquire the function names.

3.4 Predicting Missing Traces

Finally, we compare each sub-group in a bucket described in Section 3.2 with fix locations identified in Section 3.3. Figure 5 sketches the comparison algorithm. Figure 5(a) shows an original stack trace with explicit call sequences. Each circle is a function corresponding to a stack frame and the arrows represent call sequences shown in the stack trace. The bug fix location is marked ✓. The crash point is drawn at the bottom.

Since the fix location is in the stack trace in Figure 5(a) (the fix modifies a function in the stack trace), we assume the fix covers the stack trace, and classify it as covered.

However, it is not always the case that fix locations are in original stack traces. A stack trace is not a complete execution trace; it only shows function call sequences at the moment of crash. Functions once called and successfully returned are not shown in the stack trace. These functions may include crash bugs and fixes may be located in these functions. To handle this case, we expand stack traces according to call relations and control flow graphs (CFG).

We acquired call relations and CFG from Firefox source code. Firefox supports multiple platforms and uses different files and implementations, depending on the target platform.

To acquire call relations, we built Firefox on the Windows platform and identified the files and declaration directives. Then, we used Understand [26] to get call relations with CFG. We also identified class hierarchy information using Understand to resolve virtual function calls.

From the CFG, we perform path analysis to recover hidden function calls. For example, Figure 6 shows the CFG of
virtual functions: Virtual functions in C++ are dynamically dispatched. The actual function called from a virtual function call site is determined dynamically, based on the type of the object in run-time. We use class hierarchy analysis [4] to statically resolve virtual function calls.

Branches: In the actual program execution, only one path of a branch is taken. However, sometimes it is not obvious to figure out which path is taken statically. We use a conservative approach by considering all possible execution paths.

Virtual functions: Virtual functions in C++ are dynamically dispatched. The actual function called from a virtual function call site is determined dynamically, based on the type of the object in run-time. We use class hierarchy analysis [4] to statically resolve virtual function calls.

Leveraging more precise stack expansion techniques such as [18] and [19], including data-flow analysis and alias analysis is our future work.

Our prediction results may depend on the expansion level. If stack traces are not expanded enough, we may not find bug fix locations. Too much expansion introduces irrelevant functions and makes our predictions inaccurate. We present and discuss the effect of different expansion levels on prediction results in Section 5.

4. EXPERIMENTAL SETUP

This section presents our experimental setup including research questions, subjects, and evaluation measures.

4.1 Research Questions

We design our experiments to address the following research questions:

• RQ1. How accurately can our approach predict recurring crash traces? For each crash trace and given fix, we predict whether the trace may recur or not. We compare our prediction results with actual recurring crash traces in the Mozilla Crash Reporting System. To measure the accuracy of our prediction, we calculate precision, recall, and F-measure.

• RQ2. How can this prediction help developers? Our approach identifies missing crash traces for a given fix. We want to evaluate if this information could be helpful for developers to catch and fix missing crash traces. We present case studies and developers’ feedback for this evaluation.
Table 4: Characteristics of our subject

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subject</td>
<td>19 releases of Firefox 3.6</td>
</tr>
<tr>
<td>Programming Language</td>
<td>C / C++</td>
</tr>
<tr>
<td>LOC</td>
<td>3.2M ~ 4.4M</td>
</tr>
</tbody>
</table>

4.2 Subjects

We chose crash fixes in Firefox 3.6 on the Microsoft Windows platform as our experiment subject for two reasons. Firstly, its crash information, including a large number of crash reports and crash bug fixes, is publicly available. Secondly, Firefox 3.6 has five beta releases and several minor releases. Minor releases usually contain security updates and bug fixes rather than major updates to add new features. When software has major changes, it is not clear if recurring crashes are due to incomplete fixes or bugs newly introduced by the major changes. By choosing minor releases, we could clearly observe and evaluate recurring crashes caused by incomplete crash fixes.

Table 4 shows our subjects in detail. We used 19 versions of Firefox 3.6 which were released between October 2009 and March 2011. Firefox is written in C and C++ and has 3M ~ 4M LOC.

Table 5 shows the number of crash buckets and stack traces used in our evaluation. From 38 crash buckets identified in Section 2, we used 33 buckets in the Microsoft Windows platform in the experiment. We excluded five crash buckets in other operating systems since we had acquired call relations and CFGs for Firefox compiled in the Windows platform.

For the 33 crash buckets, we collected crash reports from the Firefox version right before the corresponding crashes were fixed. When a crash bucket included too many crash reports, we randomly chose 1,000 reports. We identified 1,159 unique stack traces (sub-group) out of 19,438 crash reports we collected.

To evaluate our prediction results by checking which subgroup traces were recurring, we collected crash reports in the next releases, after the fixes had been applied. This recurring crash data is used as an oracle set to evaluate our prediction results. From the 1,159 traces, we identified 354 traces that had actually recurred.

Table 5: The number of crash buckets and stack traces used in the experiment.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td># of crash buckets</td>
<td>33</td>
</tr>
<tr>
<td># of recurring stack traces (sub-groups)</td>
<td>354</td>
</tr>
<tr>
<td># of total stack traces (sub-groups)</td>
<td>1159</td>
</tr>
</tbody>
</table>

4.3 Prediction Accuracy Measures

We classify each stack trace as missing or covered and predict missing traces to recur and covered traces to disappear. Our prediction results can have four outcomes: (1) predicting a recurring stack as recurring (r→r); (2) predicting a recurring stack as to disappear (r→d); (3) predicting a disappeared stack as to recur (d→r); (4) predicting a disappeared stack as to disappear (d→d). We calculate commonly used performance measures \([1, 16, 21]\), including precision, recall and F-measure from the above outcomes to evaluate the accuracy of our approach.

- **Precision**: number of stack traces correctly classified as recurring \((N_{r→r})\) over the number of all stack traces classified as recurring.
  \[
  \text{Precision } P(r) = \frac{N_{r→r}}{N_{r→r} + N_{d→r}} \tag{1}
  \]
- **Recall**: number of stack traces correctly classified as recurring \((N_{r→r})\) over the total number of actual recurring stack traces.
  \[
  \text{Recall } R(r) = \frac{N_{r→r}}{N_{r→r} + N_{r→d}} \tag{2}
  \]
- **F-measure**: a composite measure of precision \(P(r)\) and recall \(R(r)\) for recurring stack traces.
  \[
  \text{F-measure } F(r) = \frac{2 \times P(r) \times R(r)}{P(r) + R(r)} \tag{3}
  \]

In addition, we calculate feedback – percentage of crash buckets that our approach makes predictions about, among all crash buckets used in the experiment. As discussed in Section 3.4, when our approach can not identify actual bug fix locations in any of the expanded stacks from a crash bucket, we do not make a prediction for that bucket.

- **Feedback**: the number of crash buckets for which we make predictions over the number of total crash buckets used in the experiment.
  \[
  \text{Feedback} = \frac{\# \text{ of predicted crash buckets}}{\# \text{ of total crash buckets}} \tag{4}
  \]

5. RESULTS

In this section, we present the experimental results by addressing the research questions.

5.1 Prediction Performance

To address RQ1 in Section 4.1, we present prediction accuracy in terms of precision, recall, F-measure and feedback. We applied our approach to stack traces described in Section 3.2 and predicted traces likely to result in recurring crashes. Predicting recurring crashes for 1,159 traces took less than 20 minutes at each expansion level. All experiments were conducted in a machine with Core 2 Duo 2.66GHz CPU and 8GB RAM.

Table 6 shows our prediction results when the expansion level is 4. Overall, the prediction accuracy is reasonable: precision and recall are 0.57 and 0.49, respectively, and F-measure is 0.53.

Table 6: Prediction results when the expansion level is 4

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Precision</td>
<td>0.57</td>
</tr>
<tr>
<td>Recall</td>
<td>0.49</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.53</td>
</tr>
<tr>
<td>Feedback</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Precision shows the percentage of recurring stack traces correctly predicted by our approach. At L-4 expansion, we predicted 292 stack traces as recurring traces. Among them, 167 are actually recurring traces (precision = 0.57). In our dataset, percentage of recurring stack traces among all stack traces is 31\% (354/1159). So the random guesser can only achieve 0.31 precision. Our precision is much higher than that.
The recall value represents the ratio of identified recurring traces among all actual recurring traces. Our approach found almost 50% of all recurring stack traces.

The feedback is 0.88. We did not make any prediction for 99 traces in four crash buckets since we could not find bug fix locations in any of the expanded stack traces from these buckets, as discussed in Section 3.4.

We hypothetically calculated how many recurring crashes could have been avoided if missing stack traces were detected by our technique and fixed by the developers in the first place. We counted the number of recurring crash reports after fixes for correctly predicted stack traces in L-4 expansion. By examining 292 stack traces, more than 2,225 recurring crash reports in our subject could have been avoided. Note that our dataset only covers crash reports over a period of two weeks. The total number of crashes that could have been avoided can be much higher over a longer period.

As we explained in Section 3.4, our prediction accuracy may depend on the expansion level. To observe accuracy at different expansion levels, we measured precision, recall and feedback with various expansion levels: 0, 1, 2, 3, 4, 5, 7, 10 and ∞. In L-0 expansion, unexpanded original stack traces were used. L-∞ expansion means stack traces are expanded until no more expansion is possible.

Figure 7 shows prediction accuracy at different expansion levels. The y-axis shows values of precision, recall, F-measure and feedback. The maximum value for precision and recall is 0.67 and 0.78, respectively. F-measure values are between 0.63 and 0.48.

After expanding the stack for more than two levels, the feedback ratio reaches its maximum value and does not increase any further. Precision and recall differ according to expansion levels. Precision increases from L-4 onwards, while recall continuously decreases as stacks are expanded further.

Feedback value at L-0 (0.58) indicates that more than 40% of crash buckets did not include any stack traces having fix locations in their original (unexpanded) stacks. This result implies that stack expansion is necessary to find fix locations and predict recurring crashes.

However, higher levels of stack expansion may include more irrelevant functions during the expansion. As a result, accuracy is affected as we expand the traces more.

The overall prediction results show that our approach is effective at predicting recurring stack traces. It is possible to predict with either high precision or high recall, based on the stack trace expansion level. Developers may apply our approach with high expansion levels first to get higher precision. Traces so predicted are more likely to recur. After checking predicted missing traces in high expansion levels, developers can reduce the expansion level to find more traces.

### 5.2 Case Studies

This section provides case studies to address RQ2, the usefulness of our approach. Figure 8 shows two crash fixes for the crash in Figure 1. Both fixes added code to check for the argument of NS_strdup. Since their root cause is similar (non-null value for the NS_strdup argument), and the fixes are simple, Bug #540566 could have been fixed earlier, had the developer noticed that there is a trace not covered by the first fix.

Figure 9(a) shows another example. Two stack traces from a crash bucket are shown. Each circle is a function in the trace with function names on the side. The bottom circle is the crash point. The arrows show two different execution traces. The crash was first fixed in 3.6b4 (Bug #522931) by changing function nsXULTreeAccessible: :GetChildAt in trace a. However, trace b was not covered by the fix and the crash trace recurred. The same bug report was re-opened and function nsXULTreeAccessible::GetTreeItemAccessible was fixed and then finally the crash disappeared in 3.6b5. Our approach correctly predicts trace b as recurring.

Figure 9(b) and Figure 9(c) show the first and second fixes. Since the fixes are almost identical (adding IsDefunct() checker), the bug could have been fixed in the first place, had the developer noticed that crash trace b was not covered by the first fix.

One more example is shown in Figure 10. In this case, the fix location is not in the original stack trace. Our approach finds the fix location in the expanded trace from trace a. However, we could not locate the fix in the expanded trace from trace b. Firefox 3.6 included the fix, but stack trace b recurred.

The above examples show that our approach can help developers. By knowing the existence of non-covered traces, developers can easily fix the missed crash traces.

### 5.3 Developers’ Feedback

We presented our approach and some recurring crash prediction examples briefly and asked Firefox developers for feedback about the usefulness of our approach. We sent emails to 151 individual developers who are related to crashes
6. DISCUSSION

This section discusses the limitations of our approach and threats to validity.

and bug reports we used in our experiments. In addition, we sent our survey to the Firefox developer and Mozilla static analysis mailing lists. Among 21 responses, 3 developers said our approach is very useful, 7 said it is useful and 10 developers requested more information to evaluate our approach. Overall, developers were very interested in our approach and considered it useful.

In addition, we have received promising and encouraging comments from Firefox developers:

"It should be an interesting feature and useful like any automation tool. It should make the engineering work easier and keep users less annoyed."

"We have been trying to get some stack searching techniques going for quite some time to help us analyze similar frames found across many different stacks. Definitely interested in any ideas that you have for static analysis of stacks to find additional problems."

"The first patch fixed the known steps but missed the fact that other routes led to the same state inconsistency. ... If you have a system that automates that process it would indeed be helpful."

6.1 Incorrect Crash Fixes

Our approach is based on crash fix locations assuming fixes are correct. Therefore, we predict stack traces covered by fixes will disappear. However, sometimes fixes are incorrect [14, 28]. We also found incorrect crash fixes in our dataset. In fact, many of our false negative cases are due to such incorrect fixes.

Figure 11 shows a stack trace with a bug fix (Bug #541828) as an example. Firefox crashed at Line 543, and the fix added code to check the value of buf inside function nsZi-pArchive::BuildFileList, which is in the original stack trace. Our approach assumed the fix was correct and predicted the trace will disappear. However the fix was incorrect. Firefox crashed at Line 543 again. Later, the developer added more code for verifying the value of buf to finally fix this crash.

Gu et al. [14] proposed an approach to find incorrect fixes. From a concrete bug triggering input, the approach generates more inputs that can still trigger the bug. Then they used the generated inputs to verify a fix. We could not apply their approach directly to crash fixes because we could not obtain a crash triggering input from CRS. In fact, finding a crash triggering input is challenging. Developing a comprehensive fix verification technique to identify incorrect fixes remains as our future work.

6.2 Threats to Validity

We find the following threats to validity of our experiment:

- **The subject is open source software.** We use only Firefox as the subject in the experiment since it is publicly available. In addition, it has a large number of crash reports and bug fixes. Unfortunately, we could not find any other open source projects having a large number of publicly available crash reports. For this reason, we used only Firefox as our subject. Our approach may yield different results for other software projects and their crash data.

- **Collected crash data might be biased.** We chose 19 sub-releases of Firefox 3.6 to minimize the effect of major code changes on collection of recurring crashes due
functions from the top frame, and the offset distance between the matched functions. These approaches focus on finding similar traces. Our approach is different in that we focus on finding non-covered stack traces from already grouped similar traces.

Another area concerns about how to fix the reported crashes. Manevich et al. [19] proposed the PSE (Postmortem Symbolic Evaluation) technique to diagnose software failure. They adopted dataflow and alias analysis to track the flow of a single value from the failure point back to the points where the value may have originated. Kim et al. [17] introduced Crash Graphs combining multiple crash traces together to provide aggregate view of crashes. Liblit et al. [18] introduced a technique to reconstruct the execution path from a crash in several environments. Artzi et al. [3] introduced the ReCrash technique to reproduce software failure. Instead of using crash reports, ReCrash stores partial copies of method arguments in memory to reproduce the crash. Recently Kim et al. [15] introduced a technique to predict top crashes early. They extracted features of top crashes by a machine learner and predicted if a crash will be a top crash when only a small number of crash reports are submitted. All these techniques are applied before crashes are fixed to help developers. However, our approach checks if the fixes are incomplete and predicts crash traces not covered by the fixes.

7.2 Bug Fix Verification

Gu et al. [14] tried to identify incorrect fixes. They generated test cases from distance-bounded weakest precondition and used generated inputs to verify the fixes. Similarly, recent test case generation techniques using dynamic symbolic execution [9, 13, 24] can be used to verify bug fixes. Snugglebug [10] also computes weakest precondition. Starting from the bug location with a given buggy condition, Snugglebug traces back the program path to the entry point of the program and calculates the weakest precondition. If the acquired precondition cannot be satisfied, it is verified that the buggy condition can not happen and the bug is fixed. Our work also verifies bug fixes. Especially, it can find incomplete fixes in terms of fix locations. Yin et al. [28] investigated incorrect bug fixes from large operating system code bases. They found at least 14.8%–24.2% of sampled fixes for post-release bugs incorrect. Among several bug types concurrency bugs were the most difficult to fix. Our work also verifies bug fixes.

8. CONCLUSIONS

We found 48% of fixed crashes in Firefox were recurring. With the overwhelming number of crash reports, it is challenging to identify missed crash traces manually. We proposed an approach to automatically predict recurring crashes by comparing stack traces with crash fix locations. Our experimental evaluation on actual Firefox 3.6 crashes showed that our approach yielded reasonable prediction accuracy – 0.57 precision and 0.49 recall.

We believe that developers can use recurring crash prediction information to check if their crash fix covers all the reported crash traces. Had our approach been deployed earlier, more than 2,225 recurring Firefox 3.6 crashes could have been avoided. Feedback from Firefox developers involved in crash fixes shows our approach is useful.
We plan to improve our approach by removing irrelevant functions included in the stack trace expansion phase to achieve better accuracy. In addition, reducing false negatives of our approach by identifying incorrect fixes remains as future work.

9. ACKNOWLEDGMENTS

We thank anonymous reviewers, Hongyu Zhang, Yida Tao and Jindae Kim for their helpful comments on our draft.

10. REFERENCES