CrashLocator: Locating Crashing Faults Based on Crash Stacks

Rongxin Wu[§], Hongyu Zhang[†], Shing-Chi Cheung[§], and Sunghun Kim[§]

[§]Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
{wurongxin, scc, hunkim}@cse.ust.hk

[†]Microsoft Research
Beijing 100080, China
honzhang@microsoft.com

ABSTRACT

Software crash is common. When a crash occurs, software developers can receive a report upon user permission. A crash report typically includes a call stack at the time of crash. An important step of debugging a crash is to identify faulty functions, which is often a tedious and labor-intensive task. In this paper, we propose CrashLocator, a method to locate faulty functions using the crash stack information in crash reports. It deduces possible crash traces (the failing execution traces that lead to crash) by expanding the crash stack with functions in static call graph. It then calculates the suspiciousness of each function in the approximate crash traces. The functions are then ranked by their suspiciousness scores and are recommended to developers for further investigation. We evaluate our approach using real-world Mozilla crash data. The results show that our approach is effective: we can locate 50.6%, 63.7% and 67.5% of crashing faults by examining top 1, 5 and 10 functions recommended by CrashLocator, respectively. Our approach outperforms the conventional stack-only methods significantly.

Categories and Subject Descriptors

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

General Terms

Measurement, Reliability

Keywords

Crashing fault localization, crash stack, software crash, statistical debugging, software analytics.

1. INTRODUCTION

Software crashes are severe manifestations of software faults. Crashes are often required to be fixed with a high priority. Recently, many crash reporting systems such as Windows Error Reporting [14], Apple Crash Reporter [2], and Mozilla Crash Reporter [25] have been proposed and deployed. These systems automatically collect relevant information (such as the crash stack and crashed modules) at the time of crash, cluster similar crash reports that are likely caused by the same fault into buckets (categories), and present the crash information to developers for debugging.

Existing crash reporting systems [2, 14, 25] mostly focus on collecting and bucketing crash reports effectively. Although the collected crash information is useful for debugging, these systems do not support automatic localization of crashing faults. As a result, debugging for crashes requires non-trivial manual efforts.

Over the years, various fault localization techniques (e.g., [1, 18, 21, 22]) have been proposed to help developers locate faults. These techniques suggest a list of suspicious program entities by statistically analyzing both the passing and failing execution traces of test cases. Developers can then examine the ranked list to locate faults. However, these fault localization techniques assume the availability of complete information of failing and passing execution traces, while crash reports typically contain only the crash stacks dumped at the time of crashes.

In this paper, we propose CrashLocator, a novel technique for locating crashing faults based on crash stacks and static analysis techniques. Our technique is targeting at locating faulty functions as functions are commonly used in unit testing and helpful for crash reproducing [5, 16]. For a widely-used system, one crashing fault could trigger a large number of crash reports. Therefore, a sufficient number of crash stacks can be used by CrashLocator for crashing fault localization.

CrashLocator expands crash stacks into approximate crash traces (the failing execution traces that lead to crash) using static analysis including call graph analysis, control flow analysis, and backward slicing. For effective fault localization, CrashLocator applies the concept of term-weighting [24]: locating crashing faults is treated as the problem of term weighting, i.e., calculating the importance of a function (term) for a bucket of crash traces (documents). CrashLocator considers several factors to weigh a function: the frequency of a function appearing in a bucket of crash traces, the frequency of a function appearing in crash traces of different buckets, the distance between a function and the crash point, and the size of a function. Using these factors, CrashLocator calculates the suspiciousness score of each function in the approximate crash traces. Finally, it returns a ranked list of suspicious faulty functions to developers. The developers can examine the top *N* returned functions to locate crashing faults.

We evaluate CrashLocator using real crash data from three different Mozilla products, Firefox, Thunderbird and SeaMonkey. The evaluation results are promising: using CrashLocator developers

ISSTA'14, July 21–25, 2014, San Jose, CA, USA
Copyright 2014 ACM 978-1-4503-2645-2/14/07...$15.00
http://dx.doi.org/10.1145/2610384.2610386

can locate 50.6%, 63.7% and 67.5% of crashing faults by examining top 1, 5 and 10 functions in the returned ranked list, respectively. In addition, the evaluation results show that our approach outperforms the conventional stack-only approaches.

The main contributions of this paper are as follows:

- We propose a novel technique for locating crashing faults. Our framework is based on crash stacks only and does not require additional information from deployment sites or program instrumentation. To our best knowledge, this is the first time such a technique is proposed.
- We implement our technique and evaluate our approach using Mozilla products, which are real and popular projects.

The remainder of this paper is organized as follows: We introduce background information in Section 2. Section 3 introduces some observations based on our empirical study and formulates the crashing fault localization problem. Section 4 describes our approach. Section 5 presents our experimental design and Section 6 shows the experimental results. We discuss issues involved in our approach in Section 7 and threats to validity in Section 8. Section 9 surveys related work followed by the conclusion in Section 10.

2. BACKGROUND

Despite immense efforts spent on software quality assurance, released software products are often shipped with faults. Some faults manifest as crashes after software deployment. The crash information from deployment sites is useful for debugging crashes.

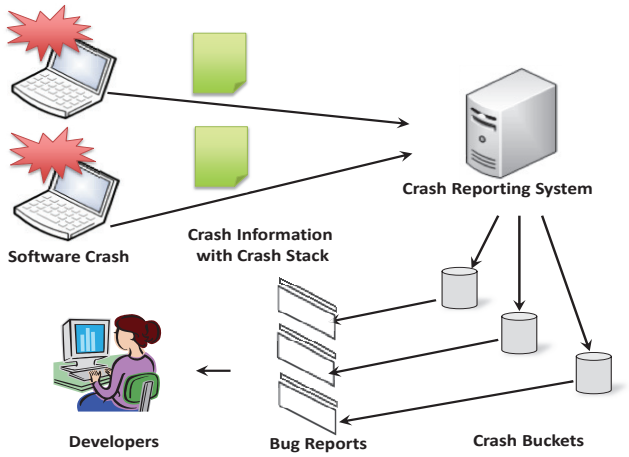

Figure 1. An overview of crash reporting system.

To collect crash information, many crash reporting systems such as Windows Error Reporting [14], Apple Crash Reporter [2], and Mozilla Crash Reporter [25] have been proposed and widely deployed. Figure 1 gives an overview of crash reporting systems. When a crash happens at a deployment site, the system collects crash-related information such as product name, version, operating system, crashed method signature, and crash stack. The collected crash information is sent to the server side upon user permission. Crash reporting systems could receive a large number of crash reports over time. Since multiple crash reports are caused by the same fault, the server checks the duplication of crash reports and assigns those likely caused by the same fault to a bucket. Finally, the crash reporting system presents the bug reports to developers.

Figure 2 is an example of crash stack in a Firefox crash report (crash ID: *f2f55573-e2cd-4ce9-92be-b16e72130904*). The program crashed at Frame 0. Each frame contains a full-qualified function name and the source file position (source file name and line number).

In large-scale systems such as Microsoft Windows and Mozilla-Firefox, developers receive a large number of crash reports sent by users at deployment sites. These crash reports are automatically grouped into different buckets, according to the crashed method signatures [25]. The grouping is based on the assumptions that a crashing fault results in the same crashed method or similar stacks. Ideally, each bucket should correspond to a unique crashing fault. In this paper, our research goal is to locate the faulty function given a bucket of crash reports.

Frame 0 Crash Signature	nsWindow::ClearCachedResources() widget/windows/nsWindow.cpp:7821
Frame 1	nsWindow::ClearResourcesCallback(HWND__ *,long) widget/windows/nsWindow.cpp:7812
	...
Frame 63	mozilla::ipc::MessagePump::Run(base::MessagePump::Delegate *) ipc/glue/MessagePump.cpp:117
Frame 64	MessageLoop::RunHandler() ipc/chromium/src/base/message_loop.cc:201
Frame 65	xul.dll@0x16d800

Figure 2. A crash stack example.

3. CRASH LOCALIZATION PROBLEM

3.1 Challenges for Locating Crashing Faults

A crash report typically contains limited information such as crashed method signature and crash stacks. The execution context, including structural coverage, under which the fault was triggered, is unavailable. Localization of faulty functions based on such limited information is challenging. More specifically, we have identified the following two main challenges: incomplete crash trace and uncertain fault location.

Incomplete crash trace

Crash stacks only contain partial crash (failing) traces and do not contain information of complete passing execution traces. Therefore, many conventional fault localization techniques cannot be applied. These fault localization techniques allow programmers to locate faults by examining a small portion of code [1, 18, 21, 22]. They contrast the passing execution traces with the failing ones, compute fault suspiciousness of individual program elements (such as statements and predicates), and present a list of program elements ranked by their fault suspiciousness. These techniques are not directly applicable to crashing fault localization because the crash stacks do not contain complete passing and failing execution traces. An alternative is to employ symbolic analysis techniques to generate possible test suites from program crashes, and apply the test suites at deployment site to collect passing and failing execution traces [3, 4]. This, however, requires the availability of a precise specification for all library calls and imposes difficulties to 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 for collecting complete passing and failing execution trace is to deploy an instrumented version at production. The instrumented version could monitor program execution at deployment sites and send execution traces as well as debugging information to developers. However, such monitoring, which considerably increases execution overheads, is rarely adopted in real production. For example, earlier studies [23, 32] analyzed dynamic binary instrumentation overhead, and found that the average performance overhead is

around 30%-150% for simple instrumentation. A recent method, BugRedux [16], introduces at function call level an average of 17.4% instrumentation overhead, which is still high for many applications.

Uncertain fault location

Schröter et al. [29] investigated crash stacks of Eclipse project. They showed that the bug fixing points of many Eclipse crashing faults involve the functions in stacks, and the bug fixing points are more likely to be found in one of the top 10 stack frames. However, our empirical study on Firefox projects (to be shown in next section) shows that many crashing faults also reside at functions that are popped out from the crash stack. Therefore examining only functions in crash stacks is inadequate. The location of faulty function is uncertain – it may or may not appear in the crash stack.

3.2 Empirical Study on Crashing Faults

To find out the amount of crashing faults that reside in the functions of crash stacks, we performed an empirical study on Firefox. We selected three Firefox released versions, from 4.0b4 to 4.0b6 as shown in the Table 1. To identify the actual faulty functions that are responsible for the crashes, we mined crash reports, bug reports and change logs as follows:

(1) We analyzed the crash reports collected by Mozilla Crash Reporter [25], and identified the crash reports that have links to bug reports in Mozilla's bug track system. Since the number of crash reports in a bucket can be large, we followed the practice of Dhaliwal et al. [12] and randomly sampled 100 crash reports in each bucket. If a bucket contains less than 100 crash reports, we collected all the crash reports in it.

(2) For each crashing fault, we collected the corresponding bug report whose status is either "RESOLVED FIXED" or "VERIFIED FIXED". We further validated the links between the crash reports and the bug reports by manual inspection. Given a link between a crash report and a bug report, we exclude the link in our experiments if they describe different product versions.

(3) For each bug report, we identified corresponding bug-fixing changes by mining the source code repository. We then obtained the relevant functions that have been modified to fix the bug.

Following the above steps, we obtained a total of 1107 crash reports corresponding to 51 crashing faults. We found that about 59% to 67% of the crashing faults can be located in the functions of crash stacks. About 33% to 41% of crashing faults reside in other functions.

Table 1. The number of crashing faults in stacks (Firefox)

Release Version	# Crashing Faults	# Crashing Faults Found in Stack	% Crashing Faults Found in Stack
4.0b4	9	6	66.7%
4.0b5	17	10	58.8%
4.0b6	25	15	60.0%

Through the empirical study, we also make the following observations about the functions that contain crashing faults:

Observation 1: Crash reports in each bucket are mostly caused by the same crashing fault. The function containing the fault appears frequently in the crash traces leading to these crash reports.

Intuitively, if a function appears frequently in multiple crash traces caused by a certain fault, it is likely to be the cause of this fault. Here, we denote crash trace as the failing execution trace triggered by a crashing fault. In our empirical study, we investigated how frequently the buggy functions appear in each bucket, and found that for 89%-92% crashing faults, their associated faulty functions appear in 100% of the crash traces in the corresponding bucket. For 89%-96% crashing faults, the associated faulty function appears in at least 50% of crash traces. The empirical results show that functions that contain crashing faults appear frequently in the crash stacks. Therefore, the function frequency can be an indicator of the suspiciousness of functions.

Observation 2: In a crash trace, functions that contain crashing faults appear closer to the crash point

In our previous work [11], we found that for some Microsoft products, the faulty functions are closer to the crash point than the "clean" functions in a crash trace. This motivates us to investigate how close the faulty functions of Firefox 4.0 are to the crash point. The distance to crash point is defined as the position offset between the current function and the crashed function. Our empirical study shows that for 84.3% of crashing faults, the distance between the associated faulty function and the crash point is less than 5, and for 96.1% of crashing faults, the distance to crash point is less than 10. The results show that the functions that contain crashing faults appear closer to the crash point. Therefore, the distance to crash point can be an indicator of the suspiciousness of functions.

Observation 3: Functions that do not contain crashing faults are often less frequently changed

Intuitively, if a function has not been changed over a long period, it is less likely to cause a crash. We investigated the number of changes made to the crashing faults, and found that 94.1% of faulty functions have been changed at least once during the past 12 months. The results show that "clean" functions are often less frequently changed. Therefore, the function change information can be a useful indicator of the suspiciousness of functions.

The above observations can be applied to locate suspicious functions. We utilize these observations in the design of CrashLocator, which will be introduced in Section 4.

3.3 Problem Definition

The crash traces for multiple crashing faults can be combined into a matrix as shown in Figure 3, where each column indicates a function and each row indicates a crash trace. When a function is covered by a crash trace, it is marked as 1; otherwise as 0. Each crash trace is associated with a specific bucket.

	f_1	f_2	…	f_{i-1}	f_m	***bucket***
T_1	**1**	**0**	…	**1**	**1**	B_1
T_2	**1**	**1**	…	**1**	**1**	B_1
…	…	…	…	…	…	…
T_{k-1}	**1**	**1**	…	**0**	**1**	B_n
T_k	**1**	**0**	…	**0**	**1**	B_n

Figure 3. The crash traces for buckets.

The goal of crashing fault localization is to identify the faulty functions that cause a crash. We need to assign a suspicious score to each function in crash traces. Developers can locate faults by examining the functions highly ranked by their suspiciousness scores.

If we treat the crash traces derived from the crash stacks of a bucket as a category of document and a function as a term, such a problem can be considered as a term-weighting problem [24], i.e., computing the importance of a term for documents. Formally, for

a set of buckets B ($B_1, B_2, \ldots B_n$) and a set of functions F ($f_1, f_2, \ldots f_m$), a weight w_i is computed for each function f_i ($1 \le i \le m$) to represent the suspiciousness of f_i with respect to B_j ($1 \le j \le n$), in such a way that a more suspicious function gets a higher weight.

4. CRASHLOCATOR

4.1 Overview

In this section, we describe the proposed CrashLocator approach. The overall structure of CrashLocator is shown in Figure 4. Given a set of bucketed crash reports, CrashLocator first recovers an approximate crash trace for each reported crash using the program's static function call graph. Since the associated crash stack does not contain all runtime information, the recovered trace is an approximation of the real crash trace. CrashLocator then calculates the suspiciousness score for each function in the recovered traces. Finally it sorts the functions by their suspicious scores and outputs a ranked list for developers to locate the fault. We describe the process of recovering crash traces in Section 4.2 and the method for ranking suspiciousness functions in Section 4.3.

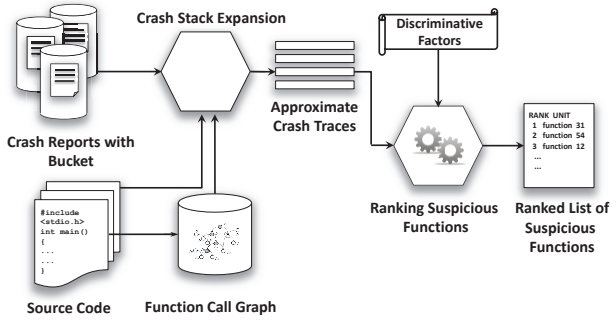

Figure 4. The overall structure of CrashLocator.

4.2 Recovering Crash Traces

4.2.1 Basic Crash Stack Expansion

Functions that contain crashing faults do not always reside in crash stacks. Therefore, to locate such crashing faults, the first step is to recover crash traces based on crash stack. Crash stack only records the stack information at the time of crash, but the functions that were popped out of crash stack during execution are not recorded. To infer possible functions executed before crash, we can leverage the static call graph [28], which captures the function call relationships of a crashing program. A static call graph consists of call pairs. Each *call pair* consists of the caller and callee functions. As an example, Figure 5(A) shows a static function call graph of a simple program, in which each node represents a function (f_i). We use the notation "$f_i \rightarrow f_j$" to denote that f_i invokes f_j.

Suppose the program starts from f_1 and crashes at f_{11}. The crash stack (Figure 5(B)) $f_1 \rightarrow f_3 \rightarrow f_{12} \rightarrow f_{11}$ actually is not a complete crash trace. Intuitively, we propose a simple crash stack expansion method, which expands the given crash stack based on a function call graph. For example, for f_1, there are two call pairs $f_1 \rightarrow f_3$ and $f_1 \rightarrow f_4$, since f_4 could be called before f_3. f_4 is included in the approximated crash trace. Once f_4 is included, f_9 and f_{13} can be in turn included in the approximated crash trace. Similarly, by expanding other functions (f_3, f_{12} and f_{11}) in the crash stack, an approximated crash trace can be obtained.

To facilitate the description of crash stack expansion, we define a concept called *Call Depth*:

Definition: The *Call Depth* of function f_i with respect to a given crash stack S is the least number of function call steps from any functions in S to f_i.

As an example, Figure 5(C) shows the call depths of the functions in Figure 5(A). f_1 is in the crash stack so its depth is 0. f_{13} can be called either by f_4 in two steps ($f_1 \rightarrow f_4 \rightarrow f_{13}$) or by f_{12} in one step ($f_{12} \rightarrow f_{13}$). In this case, its call depth is 1 as we choose the least number of steps to be the call depth. The call depth controls the set of functions to be analyzed for fault localization. The value of call depth can be set empirically; greater depth value means that more functions would be analyzed.

In crash stack expansion, we also consider virtual functions. The virtual functions are dynamically mapped to a specific method at program runtime. We apply the class hierarchy analysis method [7] to statically identify virtual function calls. We first use the static analysis tool *Understand* [33] to extract the overridden relationships among all virtual functions, and then replace each virtual function by all of its overridden functions.

4.2.2 Improved Crash Stack Expansion

In crash stack expansion, the number of functions can increase exponentially with the increasing value of call depth. Many irrelevant functions could be introduced in this process, which may affect crash localization adversely. We improve the basic crash stack expansion algorithm by reducing the number of irrelevant functions using control flow analysis, backward slicing, and function change information.

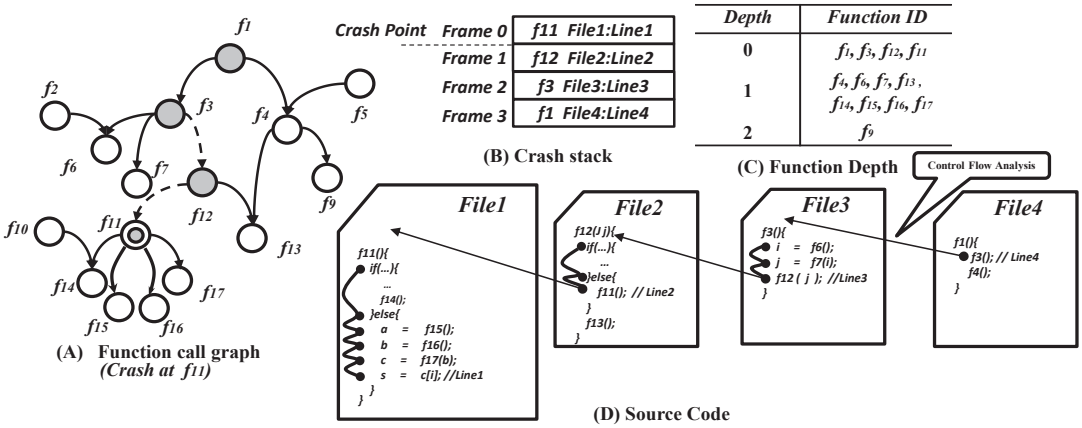

Figure 5. An Example of Recovering Crash Traces.

Our control flow analysis is similar to the approach described in [30], which aims to exclude the functions that are not reachable based on crash stack information. For example, in Figure 5(B) and (D), Frame 0 indicates that a crash happens in the function f_{11} in File1 at *Line1*. Using the basic expansion method f_{14}, f_{15}, f_{16} and f_{17} are included for further expansion. However, the control flow analysis indicates that f_{14} is not reachable, and can be treated as irrelevant functions. We apply the control flow analysis to each frame in crash stack and exclude the irrelevant functions from the expanded crash traces.

We perform backward slicing [34] for each frame in the crash stack, starting from the crash point. For example, in Figure 5(D), since crash happened at *Line1* of File1, variables *s* and *c* are relevant to the crash. First, we treat variables *s* and *c* as points of interest for slicing. Then, by backward slicing, we know that variable *c* is affected by variable *b*, so we include *b* in slicing. Finally, we include the functions f_{16} and f_{17} for further expansion because they are relevant to variable *s, b* and *c*. Although function f_{15} has been invoked at the time of crash, it has no impact on the crash-related variables and is excluded from the expanded crash traces.

We also leverage the function change information in our crash stack expansion algorithm. Our empirical study has shown that, if a function has not been changed over a long period, it is less likely to contain a crashing fault. Therefore, we select those functions that have been changed at least once within a given period for expansion. In our work, we empirically set the period value to 12 months.

Algorithm: *CrashStackExpansion* (*S*,*d*)

```
1: create HashSets ExpandSet, Set0, Set1, Set2,..., Setd
2: for each function fi ∈ S do
3:     insert fi into ExpandSet and Set0
4:     P ← all call pairs starting from fi
5:     for each pair <fi, fx> ∈ P do
6:       if (fx is called by fi via CFA and BSA) && (fx is not in ExpandSet)
7:           add fx into ExpandSet and Set1
8:       end if
9:     end for
10: end for
11: for k ← 2 to d do
12:    for each function fj ∈ Setk do
13:       P ← all call pairs starting from fj
14:       for each pair <fj, fx> ∈ P do
15:          if hasBeenChangedRecent(fx, period) //Check whether the function fx has been changed at least once within the given period
16:             if fx is not in ExpandSet
17:                add fx into ExpandSet and Setk+1
18:             end if
19:          end if
20:       end for
21:    end for
22: end for
23: return ExpandSet
```

Through the control flow analysis, backward slicing, and the analysis of function change information, we can reduce the number of irrelevant functions during crash stack expansion. The *CrashStackExpansion* algorithm iteratively expands the crash stack. It takes *S* and *d* as inputs, where *S* is a crash stack, and *d* is a predefined call depth value. First, we mark all the functions in the original crash stack as depth-0. Then for each function f_i in depth-0, we extract all the functions that are called by the function *f* via the control flow analysis (CFA) and backward slicing analysis (BSA), and add those functions that are not in *ExpandSet* into depth-1. After that, for each function f_j in depth-1, we extract all the functions that are invoked by f_j , and have been changed at least once within a certain period (e.g., in the past 12 months), and add those functions that are not in *ExpandSet*. This process is repeated until the predefined depth *d* is reached. Finally, all functions in all depths are collected to form the approximate crash trace.

4.3 Ranking Suspicious Functions

Based on the observations made in our empirical study (Section 3.2) and our prior research on software defect analysis [39, 40, 41], we consider the following four discriminative factors to identify the most suspicious functions that could cause crashes.

Function Frequency (FF):

If a function appears frequently in crash traces caused by a certain fault, it is likely to be the cause of this fault. We identify a factor *Function Frequency (FF)*, which measures the frequency of a function *f* appearing in crash traces of a specific bucket *B*:

$$FF(f,B) = \frac{N_{f,B}}{N_B} \quad (1)$$

where $N_{f,B}$ is the number of crash traces in bucket *B* that the function *f* appears. N_B is the total number of crash traces in bucket *B*.

Inverse Bucket Frequency (IBF):

If a function appears in crash traces caused by many different faults, it is less likely to be the cause of a specific fault. We identify a factor *Inverse Bucket Frequency (IBF)*, which measures the discriminative power of a function with respect to all buckets:

$$IBF(f) = log(\frac{\#B}{\#B_f} + 1) \quad (2)$$

where $\#B$ is the total number of buckets, and $\#B_f$ is the number of buckets whose crash traces contain the function *f*.

Inverse Average Distance to Crash Point (IAD):

If a function appears closer to the crash point, it is more likely to cause the crash. We identify a factor *Inverse Average Distance to Crash Point (IAD)* that measures how close a function is to the crash point:

$$IAD(f,B) = \frac{N_{f,B}}{1+\sum_{j=1}^{n} dis_j(f)} \quad (3)$$

where *n* is the number of crash traces in the bucket *B* that include the function *f*. $dis_j(f)$ represents the distances between the crash point and *f* in the *j*-th crash trace, which is defined as follows:

$$dis_j(f) = pos_j(f) + CallDepth_j(f) \quad (4)$$

where $pos_j(f)$ is the position offset between the crash point and the stack frame from which *f* is expanded, in the *j*-th crash trace. $CallDepth_j(f)$ is the call depth of *f* in the *j*-th crash trace.

Function's Lines of Code (FLOC):

Our prior research on software defect prediction [39] shows that larger modules are more likely to be defect-prone. Therefore, we use the function's size as a discriminative factor. We measure function in terms of lines of code and identify a factor FLOC as follows:

$$FLOC(f) = log(LOC(f) + 1) \quad (5)$$

where $LOC(f)$ is the number of lines of code of function *f*.

Combining the above four factors, we calculate the suspiciousness score of a function *f* with respect to a bucket *B* as follows:

$$Score(f) = FF(f,B) * IBF(f) * IAD(f,B) * FLOC(f) \quad (6)$$

Our method assigns higher scores to the functions that appear more frequently in crash traces in a bucket, less frequently in crash traces of other buckets, closer to the crash point, and larger in LOC. For each crash bucket, CrashLocator calculates the suspicious score of each function, ranks all the functions by the scores in descending order, and recommends the ranked list to developers. The developers can then examine the top *N* functions in the list to locate crashing faults.

5. EXPERIMENTAL DESIGN

This section describes our experimental design for evaluating CrashLocator.

5.1 Experimental Setup

We choose three large-scale open source projects, namely Firefox[1], ThunderBird[2], and SeaMonkey[3], as our evaluation subjects, because all of them maintain publically available crash data collected by Mozilla Crash Reporter [25]. We have collected crash reports for 5 releases of Firefox, 2 releases of Thunderbird and 1 recent release of SeaMonkey. Each release contains about 9~20K source files and 120K~280K functions. The collected crash reports are organized into different buckets by the Mozilla Crash Reporter according to the crash signatures (i.e., stacks with the same crash points are categorized in the same bucket).

We followed the same process as described in Section 3 to collect the data. We collected altogether 160 unique crashing faults (buckets). The details of our dataset are listed in Table 2.

5.2 Evaluation Metrics

CrashLocator produces a ranked list of all functions according to their suspiciousness scores (the top most function has the highest suspicious score). Developers can examine the list and locate faulty functions. Clearly, it is desirable that the faulty functions are ranked higher in the list. We evaluate the performance of CrashLocator using the following metrics:

- Recall@N: The percentage of crashing faults whose relevant functions can be discovered by examining the top *N* (*N* = 1, 5, 10…) of the returned functions. A better crashing fault localization technique should allow developers to discover more faults by examining fewer functions. Thus, the higher the metric values, the better the fault localization performance.
- MRR (Mean Reciprocal Rank), which is a statistic for evaluating a process that produces a list of possible responses to a query [24]. The reciprocal rank of a query is the multiplicative inverse of the rank of the first relevant answer. The mean reciprocal rank is the average of the reciprocal ranks of results of a set of queries Q:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \quad (7)$$

  The higher the MRR value, the better the fault localization performance. In our experiment, we use MRR to measure the ability of CrashLocator in locating the first relevant faulty function for all crashing faults.

5.3 Research Questions

To evaluate our approach, we design experiments to address the following research questions:

RQ1: ***How many faults can be successfully located by CrashLocator?***

RQ1 evaluates the effectiveness of our approach. To answer RQ1, we evaluate CrashLocator on the dataset described in Table 2. For each crashing fault, we examine the ranked suspicious functions returned by CrashLocator. If any function relevant to the fault is discovered, we consider that the fault is localized. We use Recall@N and MRR to evaluate the performance of CrashLocator.

RQ2: ***Can CrashLocator outperform the conventional stack-only methods?***

As described in Section 3, Schröter et al. [29] observed that many crashing faults can be found by manually examining the original crash stacks[4]. This RQ compares the performance of CrashLocator with the performance of the following three crash stack based localization methods:

- StackOnlySampling method, in which we randomly select one crash report from each bucket, rank the functions based on their order in the crash stack, and calculate the percentage of crashing faults that can be localized within the top *N* functions. As the crash reports in a bucket can differ in stacks, the rank of the functions would be different when sampling different crash reports. To overcome the randomness, we repeat the above process 100 times, and compute the average Recall@N value as the final results.
- StackOnlyAverage method, in which we calculate the average distance to crash point of each function appearing in a bucket. The functions are ranked based on the reverse order of the average distance. We then calculate the percentage of crashing faults that can be localized within the top *N* functions, and compute Recall@N values for all faults.
- StackOnlyChangeDate method, in which we randomly select one crash report from each bucket, rank the functions based on the reverse chronological order of their last modified date, and calculate the percentage of crashing faults that can be localized within the top *N* functions. We repeat the above process 100 times, and compute the average Recall@N value as the final results.

RQ3: ***How does each factor contribute to the crash localization performance?***

In Section 4.3, we propose four factors, namely the function frequency (FF) factor, the inverse bucket frequency (IBF) factor, the inverse average distance to crash point (IAD) factor, and the function's lines of code (FLOC) factor. RQ3 evaluates the effectiveness of these factors on crash localization performance. To answer RQ3, we incrementally integrate the factors into CrashLocator, perform crash localization on all subject systems, and compare the results with the results obtained by Equation 6 (the combination of all the four factors).

RQ4: ***How effective is the proposed crash stack expansion algorithm?***

[1]http://www.mozilla.org/firefox/
[2]http://www.mozilla.org/thunderbird/
[3]http://www.mozilla.org/seamonkey/

[4] Our discussions with some Firefox developers have confirmed that they also locate crashing faults by examining stack frames in crash reports.

Table 2. The basic information of dataset

	Firefox 4.0b4	Firefox 4.0b5	Firefox 4.0b6	Firefox 14.0.1	Firefox 16.0.1	Thunderbird 17.0	Thunderbird 24.0	SeaMonkey 2.21
# Source Code Files	9523	9612	9518	10617	11601	16129	19608	19576
# Functions	123K	125K	121K	148K	165K	214K	280K	280K
# Lines of Code	2220K	2249K	2229K	2542K	2754K	3674K	4622K	4614K
# Crashing Faults	9	17	25	25	13	33	18	20
# Crash Reports	216	394	497	803	326	173	81	105

In Section 4.2, we describe a basic crash stack expansion method, which expands a crash stack based on a static call graph. We also propose an improved crash stack expansion algorithm (the *CrashStackExpansion* algorithm), which excludes irrelevant functions by utilizing control flow analysis, backward slicing, and function change information. RQ4 evaluates the effectiveness of the improved crash stack expansion algorithm, and compares it with the basic expansion algorithm. To answer RQ4, we run CrashLocator with the basic and improved expansion algorithms respectively, and use the Recall@N and MRR values to evaluate the performance improvement.

6. EXPERIMENTAL RESULTS

This section presents our experimental results by addressing the research questions.

RQ1: ***How many faults can be successfully located by CrashLocator?***

Table 3 shows the crashing fault localization results achieved by CrashLocator for the subject systems. The call depth is set to 5. For each studied subject, CrashLocator can locate the relevant faulty functions for 47.1% to 55.6% of the crashing faults and rank them as the top 1 among the returned results. The Recall@1 value for all subjects is 50.6%. For 53.8% to 78.8% of the crashing faults, CrashLocator can successfully rank their relevant faulty functions within the top 10 returned results. The Recall@10 value for all versions is 67.5%. The results show that using our approach we can locate a large percentage of crashing faults by examining just a few functions (out of more than 121K functions).

Table 3 also shows the performance of CrashLocator measured in terms of MRR. The MRR values range from 0.528 to 0.627. The MRR value for all subjects is 0.559. In general, the evaluation results show that CrashLocator is effective in identifying faulty functions based on crash stacks.

Table 3. The performance of CrashLocator

System	Recall@1	Recall@5	Recall@10	MRR
Firefox 4.0b4	5(55.6%)	6(66.7%)	7(77.8%)	0.627
Firefox 4.0b5	8(47.1%)	12(70.6%)	12(70.6%)	0.566
Firefox 4.0b6	12(48.0%)	16(64.0%)	16(64.0%)	0.540
Firefox14.0.1	13(52.0%)	13(52.0%)	14(56.0%)	0.528
Firefox16.0.1	7(53.8%)	7(53.8%)	7(53.8%)	0.542
Thunderbird17.0	16(48.5%)	22(66.7%)	26(78.8%)	0.568
Thunderbird24.0	9(50.0%)	12(66.7%)	12(66.7%)	0.544
SeaMonkey2.21	11(55.0%)	14(70.0%)	14(70.0%)	0.600
Summary	***81(50.6%)***	***102(63.7%)***	***108(67.5%)***	***0.559***

RQ2: ***Can CrashLocator outperform the conventional stack-only methods?***

Figure 6 shows that CrashLocator outperforms the three conventional fault localization methods that are based on stack only. In total (combining all studied subjects), 50.6% of the crashing faults can be located by examining the first returned result by CrashLocator, while only 32.6%, 35.6%, and 11.3% of faults can be located in the first function returned by the StackOnlySampling, StackOnlyAverage, and StackOnlyChangeDate method, respectively. Also, 67.5% of the crashing faults can be located by examining the top 10 functions returned by CrashLocator, while only 54.8%, 53.8%, and 46.3% faults can be located in the top 10 functions returned by the StackOnlySampling, StackOnlyAverage, and StackOnlyChangeDate method, respectively. Overall, the improvement of CrashLocator over the stack-only methods ranges from 23.2% to 45.8% in terms of Recall@10.

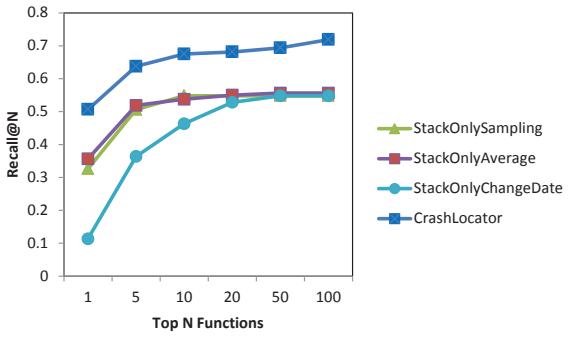

Figure 6. The comparisons between CrashLocator and three stack-only methods (Recall@N).

Table 4 also shows the Recall@1 values achieved by different methods for the studied subjects. The improvement of CrashLocator over StackOnlySampling ranges from 14.3% to 267%. Similarly, the improvement of CrashLocator over StackOnlyAverage ranges from 14.3% to 275%. The improvement of CrashLocator over StackOnlyChangeDate is even more significant, ranging from 148% to 1500%.

Table 4. The comparisons between CrashLocator and three stack-only methods (Recall@1)

System	Crash Locator	StackOnly Sampling	StackOnly Average	StackOnly ChangeDate
Firefox 4.0b4	55.6%	33.3%	33.3%	20.1%
Firefox 4.0b5	47.1%	41.2%	41.2%	19.9%
Firefox 4.0b6	48.0%	36.0%	36.0%	17.2%
Firefox14.0.1	52.0%	32.6%	40.0%	12.6%
Firefox16.0.1	53.8%	30.8%	38.5%	7.7%
Thunderbird17.0	48.5%	36.4%	39.4%	3.0%
Thunderbird24.0	50.0%	33.3%	33.3%	8.7%
SeaMonkey2.21	55.0%	15.0%	20.0%	10.0%

Figure 7 shows that, in terms of MRR, CrashLocator also outperforms the stack-only methods for each studied subjects. The improvement of CrashLocator over StackOnlySampling ranges from 28.9% to 139%, over StackOnlyAverage from 20.7% to 107%, and over StackOnlyChangeDate from 74.1% to 329%. Pair-wised t-tests also confirm the statistical significance of the results.

Overall, CrashLocator outperforms the conventional stack-only methods. This is because not all faults reside in crash stack. The stack-only methods have an inherent limitation (an upper bound exists) in locating all crashing faults.

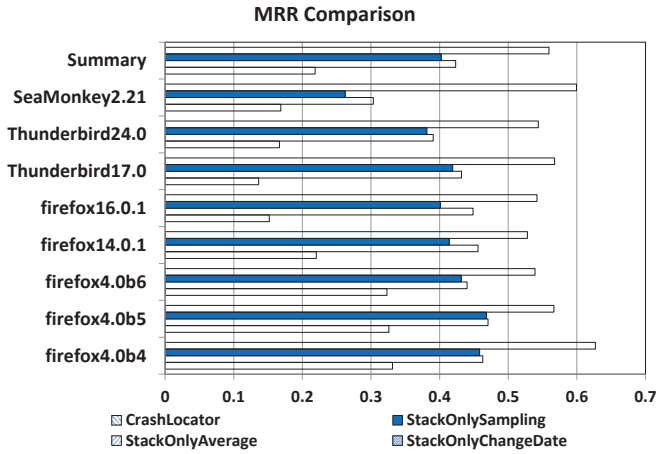

Figure 7. The comparisons between CrashLocator and three stack-only methods (MRR).

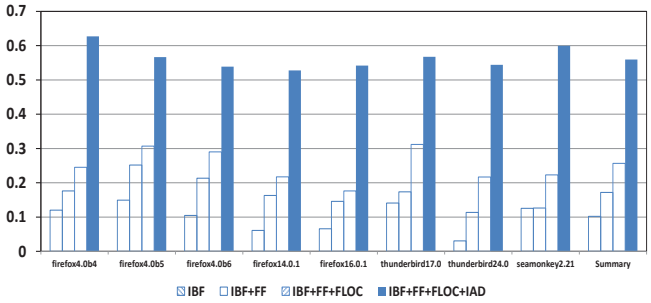

Figure 8. The contribution of each factor (MRR).

RQ3: ***How does each factor contribute to the crash localization performance?***

Figure 8 shows the performance (measured in terms of MRR) of CrashLocator by incrementally applying the IBF factor, the FF factor, the FLOC factor and the IAD factor. When the IBF factor is applied alone, the performance is the lowest (e.g., the overall MRR is 0.102). The performance is improved when the FF factor is combined with the IBF factor, as well as the FLOC factor. Finally, when all the four factors are considered (i.e., the CrashLocator approach), the performance is the best (e.g., the overall MRR is 0.559). Similar results can be observed when the performance is measured in terms of the Top *N* values. Overall, all the four factors, IBF, FF, FLOC and IAD can contribute to the performance of crash localization. The IAD factor has more significant contributions than the other factors.

RQ4: ***How effective is the proposed crash stack expansion algorithm?***

Figure 9 shows the comparison between the improved crash stack expansion algorithm and the basic expansion algorithm. The call depth is set to 5. In terms of Recall@N (*N*=1, 5, 10, 20, 50), the improved expansion algorithm outperforms the basic one by 13.3%- 72.3%. In terms of MRR, the improvement of the improved expansion algorithm over the basic one is 59.3%.

Overall, the improved crash stack expansion algorithm outperforms the basic one. This is because the irrelevant functions are filtered out by the improved expansion algorithm and the ranks of the faulty functions can be improved after the filtering.

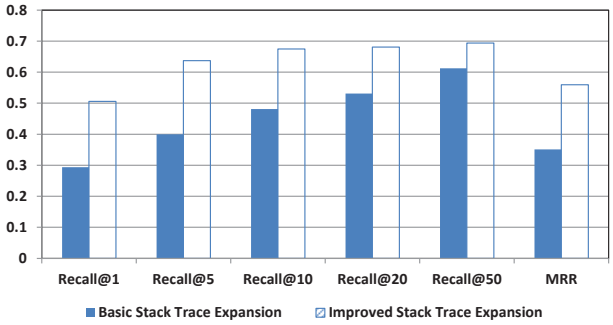

Figure 9. The comparisons between the improved crash stack expansion algorithm and the basic one.

7. DISCUSSIONS

7.1 How do different depths affect the results?

In our approach, the concept of call depth is adopted to include more functions into the ranked list of suspicious functions. The choice of call depth affects the results of crashing fault localization. Generally, increasing the depth allows to locate more faults, but it also includes more functions into the ranked list.

Table 5. Faults included in each call depth

	Depth 0	Depth 1	Depth 2	Depth 3	Depth 4	Depth 5	Depth 6	Depth 7
Recall @1	62 (38.8%)	79 (49.4%)	79 (49.4%)	79 (49.4%)	80 (50.0%)	81 (50.6%)	81 (50.6%)	82 (51.2%)
Recall @5	88 (55.0%)	93 (58.1%)	93 (58.1%)	99 (61.9%)	102 (63.7%)	102 (63.7%)	101 (63.1%)	101 (63.1%)
Recall @10	91 (56.9%)	95 (59.4%)	95 (59.4%)	103 (64.4%)	106 (66.2%)	108 (67.5%)	106 (66.2%)	106 (66.2%)

Table 5 shows the total number of crashing faults that can be discovered at different call depths, when examining the Top *N* (*N* = 1,5,10) returned functions for all versions. The call depth 0 represents the raw crash stacks without any expansion. For depth-0, when only the top 1 function is examined, 62 crashing faults can be located. This number increases with the increase of call depth.

When the call depth reaches 7, 82 crashing faults can be discovered by examining the top 1 returned function. For Recall@3 and Recall@5, we observe the same trend with the increase of depth. However, since deeper depths introduce more functions, the irrelevant functions among them would affect the ranking of relevant functions. For example, when the depth reaches 7, the Recall@5 and Recall@10 values are slightly worse than the values when the depth is 5.

7.2 Why does CrashLocator work?

CrashLocator uses a crash stack expansion approach to generate approximate execution traces, thus it could discover faulty func-

tions that do not reside in the crash stack. Our experimental results have shown that, the expanded traces are useful to localize crashing faults. As an example, let us examine the Firefox Bug 600079, which is associated with 21 crash reports. This bug was fixed at the function *mozilla::gl::GLContext::InitWithPrefix* that did not appear in the crash stack. Therefore, examining only the functions in crash stack is not sufficient. However, this function is called by the functions in the crash stack. CrashLocator can successfully locate this kind of faults via crash stack expansion. Actually the faulty function *mozilla::gl::GLContext:: InitWithPrefix* is ranked 4th by CrashLocator.

CrashLocator also considers four factors for locating relevant functions to a fault. The Function Frequency (FF) factor is based on the fact that, if a function is the root cause of a crashing fault, a large number of crash traces caused by this fault should contain this function after expansion. The Inverse Bucket Frequency (IBF) factor is based on the observation that, if a function appears in many different crashing faults' traces (such as a logging function), this function is less likely to be faulty. CrashLocator also considers the factor of the distance to crash point (IAD), under the assumption that functions closer to the crash point are more crash-prone. This has also been observed by others [11, 29].

We give an example of crashing fault that can be located using our approach. Let us examine the Firefox Bug 596245, which is associated with 9 crash reports. This bug was fixed at the function *GetStyleContextForElementNoFlush*. Similar to Bug 600079, the faulty function does not reside in the crash stacks. With the crash stack expansion algorithm, CrashLocator can successfully include the faulty function into the suspicious list by expansion. If only the IAD factor is considered, the faulty function *GetStyleContextForElementNoFlush* is ranked 20th, which is not a very high ranking. CrashLocator also leverages the information from the whole bucket of crash stacks. The FF factor gives lower scores to the functions that are occasionally close to the top of crash stacks and the IBF factor gives lower weight to the commonly-used functions. By combining the factors IAD, FF and IBF, CrashLocator reduces the rankings of irrelevant functions and promotes the rank of the faulty function *GetStyleContextForElementNoFlush* to 7th. The factor FLOC further increases the priority and finally CrashLocator can rank the faulty function in the 4th place.

In our approach, CrashLocator leverages the information of crash stacks associated with the same crashing fault (i.e., in the same bucket). A bucket usually contains many crash stacks collected from different clients. However, it is also possible that there is only one crash report in a bucket. CrashLocator still works in this extreme case. The FF and IBF factors would have no contribution to the assignment of weights to the faulty functions. However, the IAD and FLOC factors can still assign higher score to the suspicious functions. For example, in Firefox 4.0b8, 9 crashing faults (buckets) have only one crash report, 6 of them can be successfully located by CrashLocator.

Although CrashLocator is effective, it still cannot locate all crashing faults even after the expansion of crash stacks. For example, the multithread-related faults are difficult to locate. To illustrate this, we take the Bug 505059 as an example. This bug affects the constructor *nsStyleSheetService* and its destructor. According to the comments of developers recorded in Firefox Bugzilla, this crash is caused by multiple threads accessing the same reference of a service object. Since the initialization and destruction of the object are done in other threads, CrashLocator cannot locate this kind of fault. Discovering all types of crashing faults effectively remains an important future work.

7.3 Can a better bucketing algorithm improve crashing fault localization performance?

In our experiments described in Section 5, we use the crash data provided by Mozilla Crash Reporter [25]. The crash reports are organized into 160 buckets. The bucketing algorithm adopted by Mozilla is not perfect as some buckets are duplicate. We analyze the impact of bucketing accuracy on the performance of crashing fault localization. We manually examine the buckets and merge the duplicate ones (thus making each bucket corresponds to one unique fault), and run the CrashLocator again over the "perfect" buckets. The results show that the fault localization performance does not differ significantly (the differences are less than 0.002, which are not statistically significant), even a better bucketing algorithm is adopted. This is because the extra crash traces provided by duplicate buckets can be also used for effective crashing fault localization.

8. THREATS TO VALIDITY

We identify the following threats to validity:

- **Subject selection bias**: In this experiment, we only use data from Mozilla products because of the difficulty in accessing real-world crash reports. We could not find any other open source projects having a large number of publicly available crash reports that are well collected and organized. In the future, we will evaluate CrashLocator on more projects including industrial projects.
- **Large amount of data**: Our approach works better when there is a large volume of crash reports. For a small or short-living system, the number of crash reports is often small, thus making the statistical analysis inappropriate.
- **Crash data quality**: In this experiment, we directly used the crash data provided by the Mozilla crash reporting system. The quality of crash data could affect fault localization performance. Although we collect crash reports for different Mozilla products, the data could be still biased. In the future, we will investigate techniques to measure and improve the crash data quality.
- **Empirical evaluation**: In this paper, we design experiments to evaluate the effectiveness of CrashLocator. Ultimately, the usefulness of a fault localization method should be evaluated by real developers in actual debugging practice. Performing a user study is an important future work.

9. RELATED WORK

9.1 Crash Analysis

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

Ganapathi et al. [13] analyzed Windows XP kernel crash data and found that OS crashes are predominantly caused by poorly-written

device driver code. Researchers have also proposed methods for reproducing crashes in house. For example, ReCrash [5] was proposed to generate unit tests that reproduce a given crash based on captured program execution information. Csallner and Smaragdakis proposed methods for generating unit test cases for reproducing crashes [9, 10].

The above work studied the construction of a crash reporting system, the causes of crashes, and the reproduction of crashes. Our work also focuses on analyzing software crash reports. Unlike the above work, we address the problem of locating crashing faults, in order to facilitate debugging activities.

9.2 Automated Debugging

Debugging software is an expensive and mostly manual process. Over the years, many automated debugging techniques have been proposed to support debugging tasks. Fault localization methods (e.g., [1, 18, 21, 22]) can rank suspicious statements in programs based on the percentages of failing and passing test cases that execute the statements. These methods mainly differ in the type of execution information collected (e.g., statements [18] or predicates [21, 22]), and in the way they compute suspiciousness scores [1]. Parnin and Orso did an investigation [26] on the effectiveness of fault localization techniques by comparing programmers debugging time with and without automatic debugging tools. Their results show that simply examining a faulty statement in isolation is not always enough for developers to detect a bug. Our technique leverages crash stack only and can achieve high accuracy in fault localization at the function level. Our experiments show that developer could locate more than 67.5% faults in different Mozilla products by examining only the top 10 returned functions.

Besides statistical fault localization techniques, many other techniques have also been proposed to facilitate debugging [27, 37]. For example, Yoo et al. proposed Information Theory based techniques that can reduce fault localization costs and improve the effectiveness [35]. Zhou et al. [42] proposed an information retrieval based approach, which can locate faulty files based on initial bug reports. Jiang et al. [15] proposed a context-aware statistical debugging method that can not only locate the bug but also provide the faulty control flow paths. Delta debugging [36] simplifies the failed test cases and yet preserves the failures, producing cause-effect chains and linking them to suspicious statements. Zhang et al. [38] applied program slicing techniques to fault localization by identifying a set of program entities that could affect the values of variables at a given program point. Artzi et al. [3, 4] proposed fault localization methods that leverage combined concrete and symbolic executions. F. Servant and J. Jones [31] leveraged the statistical fault localization results and the history of source code to assign the faults to developers. These techniques require many inputs such as test cases, complete execution traces and initial bug reports. Our approach utilizes only 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.

10. CONCLUSIONS

In this paper, we propose CrashLocator, a novel technique for automatically locating crashing faults. Based on crash stacks, CrashLocator generates approximate crash traces by stack expansions, computes the suspiciousness scores of all functions in the approximate crash traces, and returns a ranked list of suspicious functions. Our evaluations on three Mozilla products in eight versions show that the proposed approach is effective in locating crashing faults. Using CrashLocator, we 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 the conventional stack-only methods significantly, with the improvement in Recall@10 ranging from 23.2% to 45.8%.

In the future, we will evaluate CrashLocator on more projects, including industrial projects. We will also try to integrate CrashLocator into a crash reporting system and evaluate its effectiveness in practice.

11. ACKNOWLEDGEMENTS

This research is supported by the Hong Kong SAR RGC/GRF grants 611912 and 16201814, as well as NSFC grant 61272089. We thank Liang Gong, Hyunmin Seo, and Ming Wen for their help and comments on experiments and on an initial version of this paper. We also thank Mozilla Firefox developers who helped answer our questions.

12. 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 (TAICPART-*

MUTATION 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. 1025 - 1050, 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.1084-1093, 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://crash-stats.mozilla.com.

[26] C. Parnin, and A. Orso. "Are Automated Debugging Techniques Actually Helping Programmers?", In *Proc. ISSTA 2011*, pp. 199-209.

[27] M. Pezze and M. Young. *Software Testing and Analysis: Process, Principles and Techniques*. Wiley, 2007.

[28] B. Ryder, "Constructing the Call Graph of a Program", *IEEE Transactions on Software Engineering,* 5(3), pp. 216-226, 1979.

[29] A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?", In *Proc. MSR 2010*, pp. 118–121.

[30] H. Seo and S. Kim. Predicting recurring crash stacks. In *ASE 2012,* pp. 180 – 189, Essen, German, Sep 2012.

[31] F. Servant and J. Jones. "WhoseFault: Automatic developer-to-fault assignment through fault localization." in *Proc. ICSE 2012*, Zurich, Switzerland, June 2012.

[32] G. Uh, R. Cohn, B. Yadavalli, R. Peri, and R. Ayyagari. "Analyzing dynamic binary instrumentation overhead". In *WBIA Workshop at ASPLOS*, 2006.

[33] Understand for Java tool, available at www.scitools.com

[34] G. A. Venkatesh. The semantic approach to program slicing. In *Proc. PLDI 1991*, pp. 107-119, Toronto, Canada, June 1991.

[35] S. Yoo, M. Harman, and D. Clark. Fault Localization Prioritization: Comparing Information Theoretic and Coverage Based Approaches, *ACM Transactions on Software Engineering and Methodology*, to appear, 2013.

[36] A. Zeller. Isolating cause-effect chains from computer programs. In *Proc. FSE 2002,* pp. 1-10, Charleston, South Carolina, 2002.

[37] A. Zeller. *Why does my program fail? A guide to automated debugging*. Morgan Kaufmann, May 2005.

[38] X. Zhang, N. Gupta, and R. Gupta. Pruning Dynamic Slices With Confidence. In *PLDI 2006*, pages 169–180, June 2006.

[39] H. Zhang. An investigation of the relationships between lines of code and defects. In *Proc. ICSM 2009*, pp. 274–283, Edmonton, Alberta, Canada, 2009.

[40] H. Zhang, X. Zhang, and M. Gu, Predicting Defective Software Components from Code Complexity Measures, In *Proc. PRDC 2007*, Melbourne, Australia, 93-96, Dec 2007.

[41] H. Zhang, On the Distribution of Software Faults, *IEEE Trans. on Software Eng.*, 34(2), 2008.

[42] J. Zhou, H. Zhang, and D. Lo, Where Should the Bugs be Fixed? in *Proc. ICSE 2012*, Zurich, Switzerland, June 2012.