Empir Software Eng (2018) 23:2866–2900
https://doi.org/10.1007/s10664-017-9567-4

ChangeLocator: locate crash-inducing changes based on crash reports

Rongxin Wu[1] · Ming Wen[1] · Shing-Chi Cheung[1] · Hongyu Zhang[2]

Published online: 11 November 2017

Abstract Software crashes are severe manifestations of software bugs. Debugging crashing bugs is tedious and time-consuming. Understanding software changes that induce a crashing bug can provide useful contextual information for bug fixing and is highly demanded by developers. Locating the bug inducing changes is also useful for *automatic program repair*, since it narrows down the root causes and reduces the search space of bug fix location. However, currently there are no systematic studies on locating the software changes to a source code repository that induce a crashing bug reflected by a bucket of crash reports. To tackle this problem, we first conducted an empirical study on characterizing the bug inducing changes for crashing bugs (denoted as *crash-inducing changes*). We also propose ChangeLocator, a method to automatically locate crash-inducing changes for a given bucket of crash reports. We base our approach on a learning model that uses features originated from our empirical study and train the model using the data from the historical fixed crashes. We evaluated ChangeLocator with six release versions of Netbeans project. The results show that it can locate the crash-inducing changes for 44.7%, 68.5%, and 74.5% of the

Communicated by: Martin Monperrus and Westley Weimer

✉ Rongxin Wu
wurongxin@cse.ust.hk

Ming Wen
mwenaa@cse.ust.hk

Shing-Chi Cheung
scc@cse.ust.hk

Hongyu Zhang
hongyu.zhang@newcastle.edu.au

[1] Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Hong Kong, China

[2] School of Electrical Engineering and Computing, The University of Newcastle, Newcastle, Australia

bugs by examining only top 1, 5 and 10 changes in the recommended list, respectively. It significantly outperforms the existing state-of-the-art approach.

Keywords Crash-inducing change · Software crash · Crash stack · Bug localization

1 Introduction

Software crashes are usually considered as severe bugs. They cause unintended program behaviour, bad user experiences, or even disasters for safe-critical systems. Various crash reporting systems such as Windows Error Reporting (Glerum et al. 2009), Apple Crash Reporter (Technical note tn2123: Crashreporter 2015), Mozilla Crash Reports (Mozilla crash reports 2015) and Netbeans Exception Reports (Netbeans exception reports 2015), have been developed and deployed. These systems (Technical note tn2123: Crashreporter 2015; Glerum et al. 2009; Mozilla crash reports 2015; Netbeans exception reports 2015) automatically collect crash reports from end users and group similar crash reports into buckets, which are proved to be useful for prioritizing debugging effort (Glerum et al. 2009). However, automatic debugging for crashing bugs is not supported by existing crash reporting systems.

Over the years, various bug localization techniques have been proposed to help developers debug (e.g., Abreu et al. 2007: Jones et al. 2002; Liblit et al. 2003; Saha et al. 2013; Ye et al. 2014; Zhou et al. 2012; Wong et al. 2014). These techniques rank suspicious statements or modules based on execution traces or bug reports. Although these techniques could be useful in some situations, their effectiveness is largely unsatisfactory in practice and they are rarely used by developers (Parnin and Orso 2011). One reason is that locating buggy statements or modules in isolation may not provide adequate information for developers to understand the bugs (Parnin and Orso 2011).

Our previous study (Wen et al. 2016) found that, in order to understand and fix a bug, developers often refer to the bug inducing changes (i.e., the commits that initially introduce the bug) in the discussion of bug reports. In our investigation, we found that 1,851 bug reports (Bug report list 2015) about crashes in Mozilla include developers' discussion on bug inducing changes made to the repository. Developers extensively discussed "regression range" (i.e., the last known good revision to the first known bad revision) (Regression range 2015) in bug reports. We sampled some developers' discussions and their desire to learn about bug inducing changes as follows.

> "given the regression range (for which I'm most grateful) ...[1]
>
> Finding a regression range to find the change that busted it could be useful.[2]
>
> Alternatively, you could check if you can reproduce on some of the older (archived) UX builds and/or otherwise get an idea of the regression range.[3]
>
> And to be useful, the range (of course) needs to be narrow ... In my experience, given that narrow a regression range, it's often possible to guess which patch triggered a

[1] https://bugzilla.mozilla.org/show_bug.cgi?id=448608

[2] https://bugzilla.mozilla.org/show_bug.cgi?id=589191

[3] https://bugzilla.mozilla.org/show_bug.cgi?id=941044

problem. Then I could do a test build with that patch backed out, and ask you guys to test it.[4]

... help in narrowing down the regression range and finding the right product/component is appreciated".[5]

Based on our investigation of developers' discussions and source code repository, we summarize four major benefits that developers can derive from understanding bug inducing changes. First, since a bug inducing change records the developer who committed it, it facilitates the triage of the bug to the developer who is familiar with the corresponding code. As shown in Fig. 1, the bug was fixed by the developer *Vladimir Kvashin* who was also the committer of the bug inducing changes. Our previous study (Wen et al. 2016) confirmed that 77.86% of the bugs were fixed by the committer of the bug inducing changes. Second, the modified code in the bug inducing changes provides the contextual clues that can help explain and fix the bugs. Take the bug #244574 as an example. This bug is an NullPointer Exception (Fig. 1a), and it crashes at Line 317 as shown in Fig. 1b. This crash is due to the `null` value of `p.second()`. From the bug inducing change, we can get the clue why `p.second()` is `null`. At Line 449, the return value is an object of `Pair` class whose field `second` is assigned with the null value. At Line 425, the object `cur` can be assigned with an object of `Pair` class which is returned from Line 449 (dotted line 1). The variable `cur` then can be assigned to another object `result` in the method `getEnv` at Line 428 (dotted line 2), and then `result` is returned (dotted line 3). At Line 315, the object `p` is assigned with the return value of the method `getEnv` and then used at Line 317 (dotted line 4). By examining the bug inducing code, we can understand how a `null` value is propagated to the variable used in crash point. Moreover, in Fig. 1c, the bug-fixing code at Line 317 and 319 is semantically equivalent to the deleted code before Line 315 in the bug inducing change, which indicates that developers can get hints of how to write the bug-fixing patch from the bug inducing change. Third, reverting bug inducing changes is one of the ways that developers resolve the bugs. In our investigation of NetBeans project, there are more than 1,069 bug inducing changes that were reverted for fixing bugs in the source code repository. Furthermore, the ability to locate the bug inducing changes is also beneficial for *automatic program repair* techniques (e.g., Le Goues et al. 2012; Arcuri and Yao 2008; Weimer et al. 2010), since it narrows down the root causes and reduces the search space of bug fix location.

Although bug inducing changes are important to program debugging in practice, there are no systematic studies on how to locate these changes for the bugs discovered in the field. One relevant study is delta debugging (Zeller 1999), which intensively runs the test cases on the different combination of changes and locates the bug introducing changes. Similar to spectrum-based bug localization techniques, this technique assumes the availability of failing test cases, which is not generally valid for the bugs happened in the field. The other relevant studies (Kamei et al. 2013; Kim et al. 2008; An and Khomh 2015; An et al. 2017) are to predict whether a committed change is buggy. However, these techniques cannot distinguish which bug a buggy-prone change is blamed for, and developers may not know what to do without actionable information.

[4] https://bugzilla.mozilla.org/show_bug.cgi?id=400291

[5] https://bugzilla.mozilla.org/show_bug.cgi?id=446630

```
Bug: #244574
User:Exception Reporter
Reported Date: May 18 18:16 2014 UTC
Description: NullPointerException at org.netbeans.modules.cnd.makeproject.ui.
RemoteSyncActions$BaseAction.enable
```

(a) Bug #244574

```
changeset:e10374c2df360
user:Vladimir Kvashin
date: Fri Feb 28 14:56:50 2014
summary: additional fix for #242416 Hide "upload to" action in the case remote host
uses "System-level-file-sharing"
    -  ExecutionEnvironment execEnv = getEnv(activatedNodes);
    -  If (execEnv != null && execEnv.isRemote()) {
    -     enabled = ServerList.get(execEnv).getSyncFactory().isCopying();
315 +  Pair p =<ExecutionEnvironment, RemoteSyncFactory> = getEnv(activatedNodes);
316 +  if(p != null && p.first()!= null&& p.first().isRemote()){
317 +     enabled = p.second().isCopying();   Crashing line
318    } else {
319       enabled = false;
320    }

421 +  private static Pair<ExecutionEnvironment,
          RemoteSyncFactory> getEnv(Node[] activatedNodes) {
422 +     Pair<ExecutionEnvironment, RemoteSyncFactory> result = null;

425 +     Pair<ExecutionEnvironment, RemoteSyncFactory> curr = getEnv(project);
426 +     if (curr != null) {
427          if (result == null) {
    -        result = env;
428 +        result = curr;
429          }
437       return result;
438    }
439 +  private static Pair<ExecutionEnvironment,RemoteSyncFactory> getEnv(Project
          project) {

449  +  return Pair.of(ServerList.getDefaultRecord().getExecutionEnvironment(),null);
450    }
```

(b) Bug Inducing Change for Bug #244574

```
changeset:65f38533c2ea
user:Vladimir Kvashin
date: Tue May 20 17:42:33 2014
summary:fixed #244574-NullPointerExcetion at org.netbeans.
modules.cnd.makeproject.ui.RemoteSyncActions$BaseAction.
enable
315    Pair p<ExecutionEnvironment, RemoteSyncFactory> = getEnv(activatedNodes);
316    if(p != null && p.first() != null && p.first().isRemote()){
   -     enabled = p.second().isCopying();
317 +    RemoteSyncFactory sync = p.second();
318 +    if (sync == null) {
319 +      sync=ServerList.get(p.first()).getSyncFactory();
320 +    }
321 +    enabled = (sync == null)?false:sync.isCopying();
322    } else {
323       enabled = false;
324    }
```

(c) Bug-Fixing Change for Bug #244574

```
Class: Pair
public final class Pair<First,Second> {
   private First first;
   private Second second;
   private Pair(First first, Second second) {
        this.first = first;
        this.second = second;
   }
   public First first() {
        return first;
   }
   public Second second() {
        return second;
   }
   public static Pair of (First first, Second second) {
        return new Pair<First, Second>(first, second);
   }
}
```

(d) Class Pair

Fig. 1 An example of bug inducing and fixing change in NetBeans

In this paper, we target at locating the bug inducing changes for a specific type of bugs – crashing bugs, given only buckets of crash reports. We denote the target changes as ***crash-inducing changes***. We select crash-inducing change which is at the commit level as the target, because commits are the units frequently used by developers in the bug reports. To understand crash-inducing changes, we conduct an empirical study on the characteristics of crash-inducing changes on the popular and widely used open source project NetBeans. Based on that, we propose ChangeLocator, a novel technique for locating crash-inducing changes given only buckets of crash reports. Our approach is based on a learning model that uses features originated from our empirical study and is trained by the historical fixed buckets. To evaluate ChangeLocator, we conduct an experimental study using more data from NetBeans projects. The evaluation results are promising: using ChangeLocator, we can locate 44.7%, 68.5%, and 74.5% of crash buckets by examining only top 1, 5 and 10 changes in the ranked list. Besides, ChangeLocator significantly outperforms the state-of-the-art, IR-based bug localization approach Locus (Wen et al. 2016).

In summary, the main contributions of this paper are as follows:

- We conducted an empirical study on characterizing the crash-inducing changes, and proposed a novel technique to locate the crash-inducing changes based on buckets of crash reports. To the best of our knowledge, it is the first study on locating the crash-inducing changes based on buckets of crash reports.
- We implemented our technique as the tool ChangeLocator and evaluated it using six release versions of NetBeans, a popular, large-scale and open source system.

The remainder of this paper is organized as follows. First, we briefly introduce the background information in Section 2. Section 3 presents our empirical study on crash-inducing changes. Based on the results of the study, we propose our technique in Section 4. Section 5 presents our experimental design and Section 6 shows our evaluation results. We discuss issues involved in our approach in Section 7 and the threats to validity in Section 8. We survey the related work in Section 9. Finally, Section 10 concludes this paper.

2 Background

2.1 Crash Reporting System

Software crashes are one of the most severe manifestations of software bugs. Due to their severity, software crashes are often to be fixed at a high priority. Since the crash information in the field is useful for debugging, many crash reporting systems such as Windows Error Reporting (Glerum et al. 2009), Apple Crash Reporter (Technical note tn2123: Crashreporter 2015), Mozilla Crash Reports (Mozilla crash reports 2015) and Netbeans Exception Reports (Netbeans exception reports 2015) have been developed and deployed. Once a crash happens in the deployment site, a crash report capturing crash related information (including crash stack, build id, component name, version, operation system and so on) is generated and sent to crash reporting system. Among all the crash related information, one of most important information is crash stack. Figure 2 gives an example of a crash stack trace in NetBeans (Exception ID: 2793). A crash stack trace is composed of several frames, with the most recently executed frame at the top and the least recently executed at the bottom. Each frame contains a full-qualified method name and source file position. To facilitate the explanation below, we call the line in each frame as *crashing line*, call the crashing line at the top frame as *crashing point*, and call the method in each frame as *crashing method*.

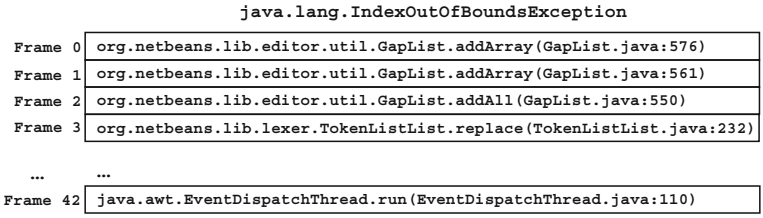


Fig. 2 An example of a crash stack

In large-scale and widely-used systems such as Windows, Mozilla Firefox and Netbeans, crash reporting systems would receive a large number of crash reports in one day. For example, in Microsoft, Windows Error Reporting system has collected billions of crash reports during their ten years operation. Many crash reports are duplicate since they are caused by the same bug. To prioritize the debugging effort, crash reporting systems automatically group duplicate crash reports into **buckets**, based on crash signatures (Mozilla crash reports 2015) or similar crash stacks (Dang et al. 2012). Then a bug report will be generated for a crash bucket, when the crash bucket is serious to catch the developers' attention (e.g., the crash occurrence for a bucket exceeds a threshold (Dang et al. 2012; Netbeans report exception faqs 2015)). Ideally, each bucket should correspond to a unique crashing bug.

2.2 Collecting Bug Inducing Changes Based on Bug-Fixing Changes

As bug inducing changes are widely used in change-based software defect prediction (Kamei et al. 2013; Kim et al. 2008, 2011), some techniques (Kim et al. 2006; Śliwerski et al. 2005) are proposed to identify bug inducing changes based on bug-fixing changes. Sliwerski et al. proposed the SZZ algorithm (Śliwerski et al. 2005). The algorithm mainly works in three steps. First, it finds the bug-fix changes through the links between bug reports in bug tracking system (e.g., BUGZILLA) and committed changes in the source code repository (e.g., CVS, Git, or Mercurial). Then, it identifies the modified source code in the bug-fixing changes. Finally, it traces back to the revisions where the modified source code was introduced, and filters out the revisions that are impossible to induce the bug. Kim et al. (2006) improved the SZZ algorithm by removing non-semantic source code changes and outlier fixes.

Note that, the existing techniques to collect the bug inducing changes are under the assumption that, the bugs have been fixed and the bug-fixing changes are available. In contrast to these techniques, our paper aims at locating inducing changes before the bug is fixed, given only buckets of crash reports. In this paper, since both our empirical study and experimental evaluation require bug inducing changes, we also leveraged the improved SZZ algorithm (Kim et al. 2006) to collect data.

2.3 Predicting Bug Inducing Changes

To help avoid the introduction of bugs when a change is committed, some researchers propose "Just-In-Time Quality Assurance". This line of studies builds the models that predict if a change is likely to be bug inducing (Kamei et al. 2013; Kim et al. 2008, 2011) or crash-inducing (An and Khomh 2015; An et al. 2017) before a change is integrated into the code repository. These techniques are useful for prioritizing the limited software quality

assurance resources to those buggy-prone changes. Although these techniques can predict changes as bug inducing or crash-inducing, they cannot distinguish which bug an inducing change should be blamed for. In other words, these techniques are not applicable to the targeted problem in this paper, which locates the crash-inducing changes given a bucket of crash reports.

2.4 Challenges

For large-scale projects, developers may commit a large number of changes. Thus, locating bug inducing changes is non-trivial. We investigated three releases from the Netbeans project. As shown in Table 1, each release includes 110K - 176K revisions.

One possible method to reduce the candidate changes is to extract only the revisions from which every line of source code in current release is introduced to the source code repository.

The intuition behind this method is that, most of bug inducing changes would insert some pieces of code in the current release and this allows us to trace the inducing changes. Although it is possible that a bug inducing change may only delete some pieces of code and cause a bug, we found that this case is very rare. In our study, none of crashing bugs are caused by a crash-inducing change which only deletes pieces of code. The command ***annotate*** facilitates us to check the changes from which each line of a given source file is introduced. In this way, we can get a smaller set of candidate bug inducing changes from Mercurial annotate command. However, the number of candidate changes by this method is still large, ranging from 47–62 K, as is shown in Table 1.

3 Empirical Study

3.1 Setup of Empirical Study

Although crash-inducing changes provide developers with useful contextual information for debugging, there are no systematic studies on characterizing the crash-inducing changes, to the best of our knowledge. In this paper, we aim to fill in this gap by conducting an empirical study on crash-inducing changes. We try to answer the following two questions:

– **Question 1:** Can we narrow down the candidate set of the crash-inducing changes based on crash reports?

  Crash-inducing changes exist in the code repository. Since there are a large number of software changes in code repository, exploring the crash-inducing changes directly from the whole code repository is non-trivial. Thus, we propose the first research question to find out whether we can narrow down the searching scope of crash-inducing changes by leveraging the crash reports. Crash reports provide abundant crash relevant information

Table 1 Number of potential crash-inducing changes

Subject	Release date	#Revisions committed before release	#Revisions from mercurial annotate command
Netbeans 6.5	Nov. 20, 2008	110,887	47,537
Netbeans 6.7	Jun. 29, 2009	139,189	56,017
Netbeans 6.8	Jun. 15, 2010	158,936	62,417

(e.g., crash stack). The prior studies (Schroter et al. 2010; Wu et al. 2014) have shown that, crash stacks are useful in finding faulty modules. Inspired by these studies, we propose to leverage crash reports to explore the candidates of crash-inducing changes.

– **Question 2:** What are the characteristics of crash-inducing changes?

The second research question is to facilitate the understanding of crash-inducing changes. The characteristics of crash-inducing changes are useful in building the statistical models for locating crash-inducing changes. Based on the results, we propose an automatic technique to locate the crash-inducing changes in Section 4.

3.2 Data Collection

We chose NetBeans as the subject of our empirical study for three reasons. First, it is a popular and active open source project. Second, it maintains public tracking systems for both bug reports (Netbeans bugzilla 2015) and crash reports (Netbeans exception reports 2015). Third, the project is well-maintained. The source code repository is hosted in the online site (Netbeans source code repository 2015), and developers maintain change logs in good quality.

Especially, 82.5% (9211/11170) of bug reports have links to bug-fixing changes. This facilitates us to identify the crash-inducing changes for crashing bugs.

To identify the crash-inducing changes for crash buckets, we mined the crash reports, bug reports, and software changes as follows:

1. We collected crash reports and their bucket information from NetBeans Exception Reports (Netbeans exception reports 2015), and then identified the crash buckets that have been assigned to a bug whose status is *FIXED*. Note that, in NetBeans, each crash bucket is assigned to only one bug. Thus, each of our collected crash buckets has an explicit link to a fixed bug.
2. For each bucket extracted in Step (1), we leverage the ID of its linked bug, mine the change logs in the code repository as described in our previous work (Wu et al. 2011), and identify the crash-fixing changes for the linked bug.
3. Since we have obtained the bug-fixing changes, we leveraged the improved SZZ algorithm (Kim et al. 2006) to identify the bug inducing changes. Finally, we obtained the crash-inducing changes for each crash bucket.

We use the extracted crash buckets and their corresponding crash-inducing changes in the following empirical study, which enables us to answer the two research questions.

3.3 Data Validation

Since the quality of collect data is important to draw valid conclusions in our empirical study, it is important to ensure the data quality. To minimize this threat, we conduct manual inspections on the collected data, which includes the subject Netbean 6.5, 6.7 and 6.8 as shown in Table 1. There are two steps that may affect the data quality. One is the step to build the links between the crashing bugs and the bug-fixing changes (Step 2 in Section 3.2). The other is the step to identify the crash-inducing changes from the bug-fixing changes (Step 3 in Section 3.2).

To guarantee the quality of the links between crashing bugs and the bug-fixing changes, we manually inspect the link data. First, we verify whether the ID of linked bug appears in the log message of the linked changes. Second, we verify whether the number appearing in the log message indeed represents a bug ID, by searching for the keywords such as

"issue #", "bug #", and so on. Only if two conditions are satisfied, we will label a link as a true link.

To guarantee the quality of the collected crash-inducing changes, we also conduct manual inspection. We adopt the manual examination approach similar to the previous study (Kim et al. 2006). Two graduate students performed the manual verification independently. One manually marked each collected crash-inducing change as true or false. The other reviewed the marks. Only if the two students reach on a consensus that a change is a true crash-inducing change, we will use it in our empirical study. In total, 58.6% (242/417) of candidate crash-inducing changes are considered as noise data and filtered out in our empirical study.

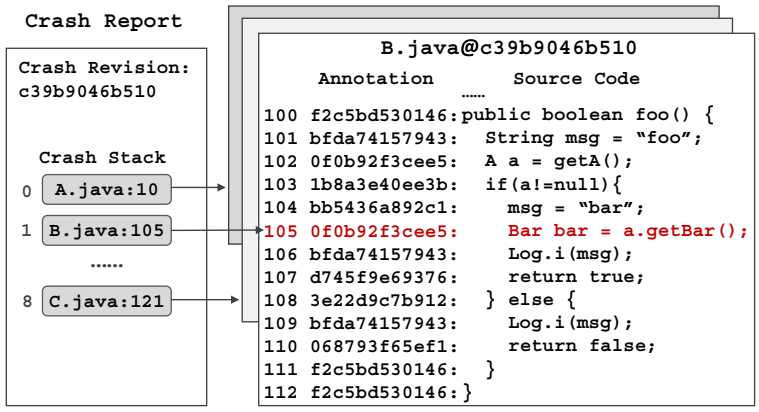
3.4 Results of Empirical Study

Observation 1 We can narrow down the candidate set of the crash-inducing changes based on crash reports.

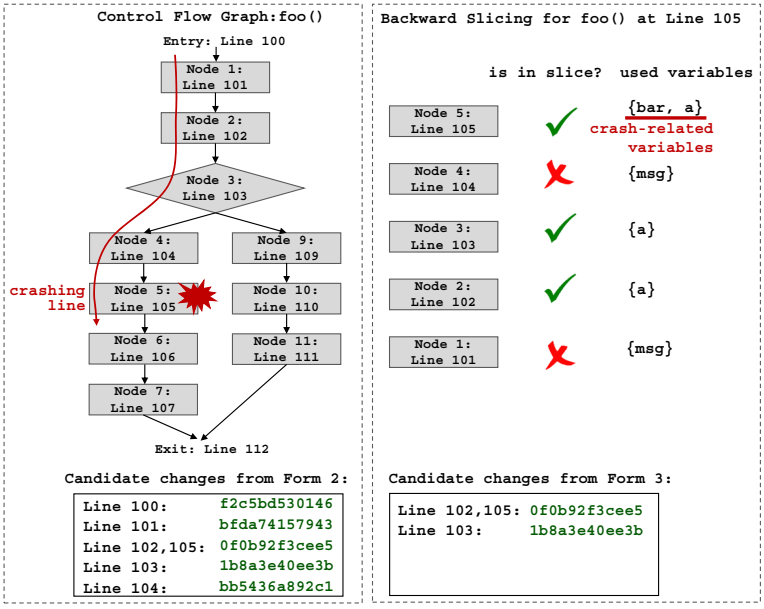
Crash reports provide crash relevant information including the crash stack and the revision number where the crash happens (we call it crashing revision for short). Take the crash report in Fig. 3 as an example, it contains an 8-frame crash stack and the crashing revision is `c39b9046b510`.

The Mercurial built-in command ***annotate*** facilitates us to check the revision from which each line of a given source file is introduced in the source code repository. For example, Fig. 3 shows the annotated information of the source file `B.java` in the revision `c39b9046b510`. Based on those annotated information of each frame in the crash stack, we explore three forms of crash-inducing change candidates as follows:

- ***Form 1: Candidates associated with the methods in a crash stack***. This candidate set considers all the revisions where lines in crashing method of each frame are introduced. For instance, for the second frame in Fig. 3, the crashing method is `foo()`, thus the whole `foo`'s annotated revisions from Line 100 to Line 112 will be included.
- ***Form 2: Subset of candidates from Form 1 which exclude irrelevant revisions using control flow analysis.*** As some statements in crashing methods are not reachable to the crashing line, the revisions where these statements are introduced are unlikely to be crash-inducing changes. Therefore, we follow our previous work CrashLocator (Wu et al. 2014), and use control flow analysis to reduce the candidate inducing changes from Form 1. For example, Fig. 3b shows the control flow analysis for the crashing method `foo()`. Through the analysis, we can extract the reachable statements which are Line 100 to 105. Finally, we can get a smaller candidate set of inducing changes than candidates from Form 1, including only 5 changes in crashing method `foo()`.
- ***Form 3: Subset of candidates from Form 1 which exclude irrelevant revisions using backward slicing.*** As some statements have no impact on the crash-related variables (i.e., the variable that are used in the statements in crashing lines), the revisions where these statements are introduced are unlikely to be crash-inducing changes. Therefore, we follow our previous work CrashLocator (Wu et al. 2014), and use the backward slicing (Venkatesh 1991) to reduce the candidate inducing changes. For example, in Fig. 3b, variable *bar* and *a* are crash-related. By backward slicing, we exclude Line 101 and 104, since they do not affect the crash-related variables. Through this analysis, we only include 2 changes in crashing method `foo()`. It should be noted that, backward slicing also utilizes the control flow analysis to explore only the reachable statements, so we can get a smaller candidate set of inducing changes than candidates from Form 2. In other words, the candidate set from Form 2 subsumes the candidate set from Form 3.



(a) Candidates from Form 1.



(b) Candidates from Form 2 and Form 3.

Fig. 3 Examples of extracting candidates of crash-inducing changes from crash stack

For each crash report, we can extract the candidate changes based the above three forms. One bucket may contain multiple crash reports. Thus, for each crash bucket, we merge the candidate changes from each crash report and use the merged set as the candidate changes for this bucket. Then, we verify whether the crash-inducing changes of a given bucket are in the above three candidate forms.

We take the three releases of Netbeans project shown in Table 1 as the subjects in our empirical study. The basic information of the subjects and the results are shown in Table 2. The three subjects include 155 buckets in total. The crash-inducing changes of 75.5%, 74.8% and 72.9% of the buckets can be found by the candidate set in Form 1,2, and 3 respectively. The candidates selected from these three forms differ in their sizes, as well as their potential capability of locating bugs. Figure 4 shows the number of candidates of these

Table 2 Subjects and results of the empirical study

(The column Form i ($i = 1, 2, 3$): the num/percentage of crash buckets whose inducing changes are in the candidate set off Form i.)

Subject	#Buckets	Form 1	Form 2	Form 3
Netbeans 6.5	42	25(59.5%)	24(59.5%)	23(54.8%)
Netbeans 6.7	61	51(83.6%)	51(83.6%)	49(80.3%)
Netbeans 6.8	52	41(78.8%)	41(78.8%)	41(78.8%)
Total	155	117(75.5%)	116(74.8%)	113(72.9%)

three forms. Candidates from Form 1 include the largest number of changes, on average 48.2 changes for each bucket. Candidates from Form 2 include comparably fewer changes, on average 36.2 changes for each bucket. Candidates from Form 3 include the fewest changes, on average 24.5 changes for each bucket. The three candidate forms of inducing changes differ slightly in the percentage of buckets that can be located, but they differ significantly in their size.

Observation 2 Crash-inducing changes exhibit certain characteristics.

Through the empirical study, we make the following observations about the characteristics of crash-inducing changes.

Observation 2.1: Crash-inducing changes appear closer to the crash point. Previous study showed that faulty functions are closer to the crash point than non-faulty functions in a crash stack (Dang et al. 2012), this motivates us to investigate if crash-inducing changes have the similar characteristics. We denote *distance to crashing point* for a change r as the offset from the topmost frame where we find r to the frame where the crashing point is. Figure 5a shows the results of the statistics of distances of the crash-inducing changes for the 155 buckets. The crash-inducing changes of 31.0% of the buckets have a distance of 0 to the crashing point, 67.7% are within the distance of 5, and 72.3% are within the distance of 10 to the crashing point. The results show that, crash-inducing changes appear closer to the crash point.

Observation 2.2: Crash-inducing changes appear closer to the crashing lines. In our study, we found that in a crashing method, the lines closer to the crashing line are more likely to be buggy, so the changes that introduced these lines are more likely to be crash-inducing changes. We denote *distance to crashing line* for a change r as the minimum offset

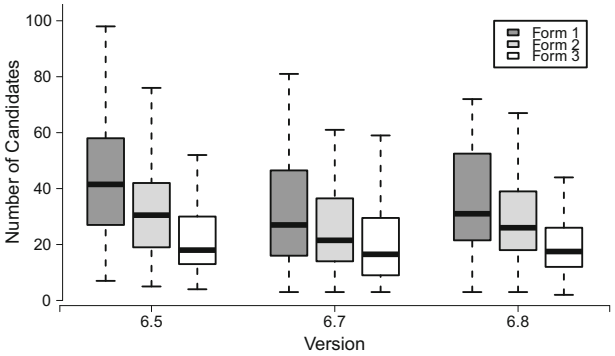


Fig. 4 Number of changes from three candidate forms

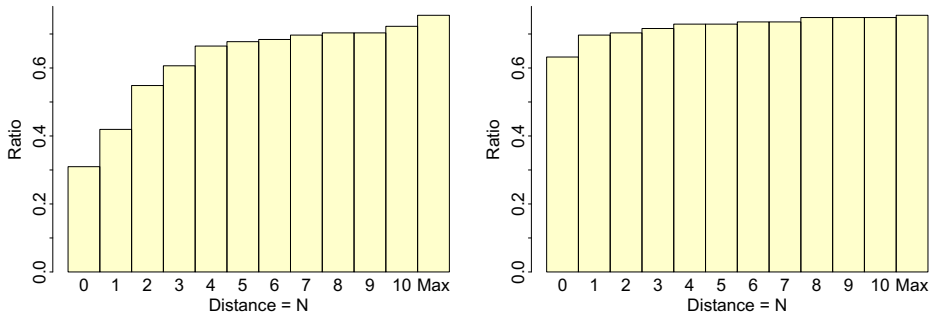


Fig. 5 The statistics of distances to crashing point and crashing line for crash-inducing changes

between lines which are introduced by r and the crashing lines. Figure 5b shows the results of the statistics of distances to crashing line for the crash-inducing changes of 155 buckets.

The crash-inducing changes of 72.9% of the buckets are within the distance of 5 to the crashing lines. The empirical results show that, crash-inducing changes appear closer to the crashing lines.

Observation 2.3: Crash-inducing changes for a bucket appear frequently in the crash reports of that bucket, and those changes that appear in multiple buckets are less likely to be crash-inducing changes. A bucket may contain multiple crash reports. Intuitively, the change which induced this crash bucket is likely to appear frequently in its crash reports. In our empirical study, we investigated how frequently the crash-inducing changes appear in the corresponding bucket. We found that, for 109(70.3%) of buckets, their corresponding crash-inducing changes appear in 100% of their crash reports.

The results showed that changes which induced a crash bucket appear frequently in the crash reports of that bucket.

A change may appear in multiple crash buckets. In this study, we found that, for the changes which appear in more than 10% of the buckets, 89.1% of them are not crash-inducing changes. For the changes which appear in more than 50% of the buckets, 100% of them are not crash-inducing changes. The results indicate that, if a change appears in many different buckets, it is less likely to be crash-inducing changes. This observation is similar to the observation on buggy functions for crashing bugs in our previous study (Wu et al. 2014).

Observation 2.4: If a change affects the source files in the component where the bucket of crash reports happened, it is more likely to be a crash-inducing change for that bucket. NetBeans is composed of hundreds of components. Since an execution of NetBeans is usually composed of the interactions between different components, the crash stacks in crash reports contain methods from different components. In the existing crash reporting system (Netbeans exception reports 2015), each bucket of crash reports will be automatically assigned to a specific component (Netbeans report exception faqs 2015). This raises a question that, whether this assigned component can help locate the crash-inducing changes. Thus, we investigated that, whether the crash-inducing changes of a bucket affect (add/delete/modify) any source file in the component which is assigned to that bucket. We found that, for 94.7% of the buckets, their crash-inducing changes affect at least one source

file in the assigned component. The results show that, the crash-inducing changes are more likely to affect the source files in the component assigned to the bucket.

Observation 2.5: The committed time of crash-inducing changes is closer to the reporting time of the crashes. Prior study (Kim et al. 2007) showed that, if an entity was changed recently, it will tend to introduce faults soon. This motivates us to study, whether a change is more likely to be crash-inducing if its committed time is closer to the crash reporting time. In our study, we found that, for 90.3% of the buckets, their inducing changes were committed within 12 months before the reporting time of the corresponding bucket. Furthermore, we ranked all the candidate changes (extracted from crashing methods) in a bucket based on the reverse chronological order of their committed time, and found that for nearly 70% of the buckets, their crashing inducing changes are ranked within top 20. The above empirical results shows that, the time of crash-inducing changes are closer to the reporting time of the crashes.

Although our study is based only on the Netbeans project, we believe the observations in our study can be also generalized to other projects, since these observations are consistent with with some previous studies on different projects (such as FireFox and Eclipse) (Wu et al. 2014; Schroter et al. 2010). For example, the observation that, crash-inducing changes appear closer to crash point, is consistent with the previous findings (Wu et al. 2014; Schroter et al. 2010) obtained from FireFox and Eclipse projects that, buggy functions/files appear closer to crash point. We also believe that, for different projects across different domains, languages and technologies, the characteristics of crash-inducing changes may vary. For example, the distance value considered as "close to" crash point may vary across different projects. This motivates us to use the supervised learning approach (See Section 4) that can well characterize the change-inducing changes for one project using the project's historical data.

4 ChangeLocator

4.1 Overview of ChangeLocator

In this section, we describe the proposed approach ChangeLocator. The goal of ChangeLocator is to locate the crash-inducing changes from the candidates found in the previous step. We need to assign a suspicious score to each candidate crash-inducing change. Then, the candidate changes can be ranked by their suspicious score.

Since the problem of ranking crash-inducing changes for a crash bucket is similar to the problem of ranking relevant documents for a query in IR(information retrieval), we propose to leverage the techniques used in information retrieval to locate the crash-inducing changes.

A class of IR techniques (Nallapati 2004; Robertson and Jones 1976) transforms the information retrieval problem into the classification problem, i.e., classifying the entire collection of documents into two classes: relevant and irrelevant. Then, it leverages different classification algorithms to estimate the probability that a document is relevant to a query, and ranks the documents based on the probability. Similarly, ChangeLocator transforms the problem of locating crash-inducing changes into the classification problem, i.e., classifying a candidate change as crash-inducing or non-crash-inducing. Then, classification algorithms are used to estimated the probability of a change being crash-inducing for a crash bucket. Finally, ChangeLocator ranks the candidate changes. As classification is a

supervised learning technique which requires training dataset, ChangeLocator collects the data from the historical crash buckets that have been fixed to build the training dataset.

Figure 6 gives the overall structure of ChangeLocator. First, ChangeLocator collects existing crash buckets that have been fixed and their candidate crash-inducing changes from crash reports (Section 3), and then identify the crash-inducing changes (Section 2.2).

The candidate changes can be derived from three candidate forms as described in Section 3.4. Since the candidate Form 3 includes the fewest candidate changes and can locate nearly the same number of crash buckets as the other two candidate forms, ChangeLocator by default extracts the candidate changes from Form 3. Second, for each candidate change in a crash bucket, if it is a crash-inducing change of the crash bucket, ChangeLocator labels it as ***TRUE*** (crash-inducing), otherwise labels it as ***FALSE*** (non-crash-inducing). At the same time, for each candidate change, we extract several features. Combining features and label of a change in a bucket, we create an instance. With the instances of existing buckets, we create a training corpus. Then, ChangeLocator uses a classification algorithm (e.g., Logistic Regression) and the training corpus to build the learning model. Finally, when a new crash bucket comes, the features of its candidate crash-inducing changes are created. For each change, the learning model estimates the probability that the label of the change is ***TRUE***. We use this probability as the suspicious score of each change, and rank the candidate crash-inducing changes for that bucket.

4.2 Feature Engineering

Since our approach ChangeLocator is based on the classification, its performance relies on the features used in the learning model. In this paper, we select the features based on the observations from our empirical study and some priori studies. These features are as follows.

Inverse Average Distance to Crashing Point (IADCP). Our empirical study showed that, if a revision appears in a crash frame which is closer to the crash point, it is more likely to be the crash-inducing change. Therefore, we identify a feature *Inverse Average Distance*

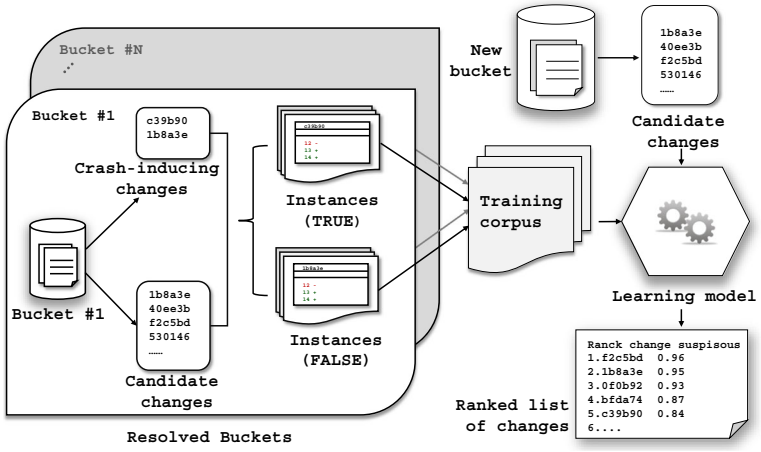


Fig. 6 Overall structure of ChangeLocator

to Crash Point (*IADCP*) that measures how close a revision r is to the crash point of crash reports in a crash bucket B. The formal definition is as follows.

$$IADCP(r, B) = \frac{1}{1 + \sum_{j=1}^{n} DCP_j(r)/n} \tag{1}$$

where n is the number of crash reports in the bucket B which contain revision r, and $DCP_j(r)$ represents the minimum distance between the revision r and the crash point in the j^{th} crash report of the bucket B.

Inverse Average Distance to Crashing Line (IADCL): As is shown in our study, if a revision appears closer to the crashing line of a frame, it is more likely to be the crash-inducing change. Thus, we identify a feature *Inverse Average Distance to Crashing Line* (*IADCL*) to measure how close a revision r is to the crashing lines of crash reports in a crash bucket B. The definition of *IADCL* is as follows.

$$IADCL(r, B) = \frac{1}{1 + \min_{j=1...n} DCL_j(r)} \tag{2}$$

where n is the number of crash reports in the bucket B, and $DCL_j(r)$ represents the minimum distance between the revision r and any frame in the j^{th} crash report of the bucket B.

Inverse Time Distance to Crash Revision (ITDCR): Our study found that, if the committed time of a revision is closer to the committed time of the revision where the crash occurred, it is more likely to be the crash-inducing change. Note that, this observation is based on the chronological order of the committed time of candidate crash-inducing changes. Therefore, we design a feature *Inverse Time Distance to Crash Revision* (*ITDCR*) as follows.

$$ITDCR(r, B) = \frac{1}{1 + Rank(r, B)} \tag{3}$$

where $Rank(r, B)$ represents the rank of the revision r among all the candidate crash-inducing changes, based on the inverse chronological order of the committed time of each change.

Revision Frequency (RF): In our empirical study, we found that, if a revision appears frequently in crash reports of a bucket, it is more likely to be the crash-inducing change. We identify a feature *Revision Frequency* (*RF*) that measures the frequency of a revision r appearing in crash reports of the crash bucket B.

$$RF(r, B) = \frac{N_{r,B}}{N_B} \tag{4}$$

where $N_{r,B}$ is the number of crash reports in the bucket B that the revision r appears. N_B is the number of crash reports in the bucket B.

Inverse Bucket Frequency (IBF): In our study, it is found that, if a revision appears in different buckets of crash reports, it is less likely to be the crash-inducing of a specific crashing fault. Based on this finding, we extract the feature *Inverse Bucket Frequency* (*IBF*) for a revision r as follows.

$$IBF(r) = log(\frac{\#B}{\#B_r} + 1) \tag{5}$$

where $\#B$ is the total number of buckets, and $\#B_r$ is the number of buckets whose candidate changes include the revision r. The design of *IBF* is analogous to *IDF* (Inverse Document Frequency) in information retrieval technology, which is used to decreased the importance of less meaningful terms (e.g., "a", "an", "the" and so on).

Crash Component (CC): Our study found that, if a revision affects (i.e., adds, deletes or modifies) source files of the component that the crash bucket belongs to, it is more likely to be the crash-inducing change. Thus, we design a feature *Crash Component* (*CC*) for a revision r in the bucket B as follows.

$$CC(r, B) = \begin{cases} 1 & \text{if } r \text{ affects source files in the} \\ & \text{component that } B \text{ belongs to} \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

Revision's Lines of Added/Deleted/Changed Code (RLOAC, RLODC, RLOCC): Prior studies (Moser et al. 2008; Nagappan and Ball 2005) showed that, the size of a change is good indicator of defect-prone modules. If the size of a change is larger, it is more likely to be defective. Therefore, we use *Revision's Lines of Added Code* (*RLOAC*), *Revision's Lines of Deleted Code* (*RLODC*), and *Revision's Lines of Change Code* (*RLOCC*) as features for a revision r. These features represent the number of lines of added/deleted/changed code a revision r commits.

Number of Affected Files (NAF): As shown in previous study (Nagappan and Ball 2005), the number of affected files by a revision is a good indicator of defect-prone changes. If the number of files that a revision affects is large, this revision is more likely to be defective. Therefore, we also adopt *Number of Affected Files* (*NAF*) as a feature.

The features described above can have very different range values. The existing studies (Kotsiantis et al. 2006; Al Shalabi et al. 2006) showed that the accuracy and the efficiency of classification algorithms would be improved if the features to be analyzed are normalized into a similar range. Therefore, we use the min-max normalization (Witten et al. 2016) method to normalize all the features in training and testing dataset, with the values of each feature ranging from 0 to 1.

4.3 Training a Classification Model

We constructed the training corpus from the historical crash buckets that have been fixed. Given unresolved crash buckets in the current version, we use the resolved buckets in the previous versions as training dataset.

Since the number of the crash-inducing changes is much smaller than the number of non-crash-inducing changes, the training dataset is typically imbalance. The imbalanced dataset mostly can compromise the performance of most standard learning algorithms in a significant way (Batista et al. 2004; Prati et al. 2004). To handle the imbalance dataset problem in the training process, there exist some typical techniques including over-sampling and under-sampling. The type of over-sampling techniques duplicate the sampling instances of the minority class (e.g., the instance whose label is TRUE in our case), while the type of under-sampling techniques remove the sampling instances of the majority class. In this paper, we adopt the random under-sampling technique, by which we randomly select and remove instances of the majority class such that the number of both classes are made equal. There are two reasons for the adoption of this technique: (1) there is some empirical evidence that the random sampling techniques performs better than or as well as other sophisticated

sampling techniques in handling imbalanced dataset problem (Mani and Zhang 2003); (2) over-sampling may result in much computational costs (Nallapati 2004).

Having collected the data, we train a classification model using a standard classifier (such as Logistic Regression, Decision Tree, Naive Bayes, etc), which can classify a candidate change into crash-inducing change (*TRUE*) or non-crash-inducing change. We use Logistic Regression as the default classifier.

4.4 Ranking Based on Classification

Having trained a classification model, when a new crash bucket comes, ChangeLocator first extracts the candidate crash-inducing changes from crashing methods. Then, it extracts the features of each candidate crash-inducing change as described in Section 4.2. By applying the learning model that has been trained, ChangeLocator outputs the probability that the label of each change is *TRUE* and uses the probability as the suspicious score of each change. Finally, we rank all candidate crash-inducing change by the suspicious scores in descending order.

5 Experimental Design

5.1 Subjects for Evaluation

NetBeans is a popular open source integrated development environment for application development cross different operating systems. We collected the crash-inducing changes dataset from Netbeans 6.5 to 7.2 (noted that, Netbeans does not have version 6.6). Table 3 shows the basic information of the subjects. Each subject contains 25,558∼53,264 source files, 5,524K∼11,632K lines of code, and 110,887∼235,526 revisions. There are 313 buckets in total. ChangeLocator requires dataset for training. For the subject Netbeans 6.5, there is no available training dataset, if we consider the chronological order of the collected Netbeans versions. Therefore, we do not conduct testing on Netbeans 6.5, and only utilize it for training. For other subject, we use all of the buckets in its previous versions as training dataset. For example, to locate crash-inducing changes for Netbeans 6.8, we use the buckets in Netbeans 6.5 and 6.7 as the training dataset.

In our previous work (Wu et al. 2014), we used Mozilla products as subjects, since they also have crash data publicly available. However, we found that it is difficult to collect crash-inducing changes for Mozilla projects. Due to the parallel development in multiple branches, developers often commit different bug-fixing patches for a same bug in different branches (This will not bring the problems in collecting buggy functions (Wu et al. 2014) and buggy source files (Wang et al. 2016), since these patches mostly modified the same functions). We found that the number of crash-inducing changes is often large and many of them contain noise. The noise data is due to two issues. (1) The bug-fixing patches for a same bug in different branches would modify different source lines, and the crash-inducing changes generated from different patches are very different sometimes. (2) Unlike the Netbeans project where each crash bucket is only linked to a single bug, a crash bucket in Mozilla often has links to multiple related bugs (these bugs are related, since they share the same crash signature). However, not all of these related bugs are valid to the given crash bucket. Take the crash bucket "`mozilla::ipc::FatalError | mozilla::layers::PLayerTransactionParent::Read`" in Firefox 45.0 as an example. There are four related bugs linked to this crash bucket, *Bug 1286437, 1248156, 1240975*, and 1208226. Our

Table 3 Subjects for evaluation

Subject	#Files	KLOC	#Revs	#Buckets
Netbeans 6.5*	25,558	5,524	110,887	42
Netbeans 6.7	48,044	10,331	139,189	61
Netbeans 6.8	49,918	10,717	158,936	52
Netbeans 6.9	53,264	11,632	176,495	40
Netbeans 7.0	46,189	9,745	196,886	38
Netbeans 7.1	38,900	8,429	217,206	41
Netbeans 7.2	39,674	8,666	235,526	39

*: Netbeans 6.5 is only used for training.
#Revs is the number of revisions committed before the release

manual validation based on the information in bug reports finds that *Bug 1248156* duplicates *Bug 1208226*, and they are valid bug reports that concern Firefox 45.0, while the other two are invalid to this version. Therefore, only *Bug 1248156* and *Bug 1208226* can be considered as the real bug of this crash bucket. If we use the fixing changes of invalid bugs to identify crash-inducing changes, we would include much noise. Eliminating all these noise requires much manual effort, and sometimes it is even impossible due to the insufficient information in bug reports. Due to the noise problem, we currently do not include Mozilla projects as our evaluation subjects.

5.2 Research Questions

To evaluate the effectiveness of our technique, we design experiments to address the following research questions:

RQ1: Can ChangeLocator effectively locate crash-inducing changes?

In this RQ, we evaluate the effectiveness of ChangeLocator in locating crash-inducing changes, using the evaluation metrics as described in Section 5.3. We extract the candidate changes from Form 3 as described in Section 3. We compare ChangeLocator with the state-of-the-art IR based approach **Locus** (Wen et al. 2016), which can locate bugs at the change level. Locus takes a bug report as input, then queries the code changes in the code repository, and locates the bug inducing changes. Since the crash reports can be used as a bug report, we then can adopt Locus to locate crash-inducing changes. To generate one bug report for each bucket of crash reports, we use the crash stacks as the content of the bug report. To verify whether it is reasonable to use only crash stacks as the bug reports for Locus, we manually inspect the bug reports that are corresponding to crash buckets in Netbeans. The bug description of most of the collected crashing bug reports includes only crash stacks or crash reports. Moreover, our proposed technique is to locate crash-inducing changes once buckets of crash reports are collected.

Therefore, to compare ChangeLocator with Locus fairly, we use crash stacks as input for both techniques. To narrow down the searching space of inducing changes, Locus also utilizes the candidate from Form 3 rather than all the changes in the repository, to locate crash-inducing changes.

Although there are many different kinds of bug localization techniques as described in the previous comprehensive literature survey (Dit et al. 2013), we select only Locus for comparison because of the following reasons. First, the input of our target problem is only crash stacks, and only IR-based bug localization techniques are applicable to this problem. Other techniques, such as spectrum-based approaches which require dynamic analysis to collect execution traces, are not applicable. Second, Locus is the state-of-the-art IR based approach. The recent study (Wen et al. 2016) showed that Locus overwhelms other IR-based approaches, such as BRTracer (Wong et al. 2014), BLUiR (Saha et al. 2013), and AmaLgam (Wang and Lo 2014). Third, Locus is designed to locate bugs at the change level.

RQ2: How does each feature contribute to the effectiveness of ChangeLocator?

We proposed 10 features which are correlated with crash-inducing changes in Section 3. This RQ aims at evaluating the effectiveness of each feature.

We evaluate the contribution of each feature through the two following steps. First, for all the evaluation subjects, we first run ChangeLocator using a single feature each time separately, and compare the learning results of each single feature with the result of learning with all features. Second, similar to the previous study (Wu et al. 2014), we run ChangeLocator by adding one feature incrementally based on the performance of each single feature, following the order from the worst one to the best one. In this way, we know how much contribution a feature has made to ChangeLocator.

RQ3: How do different candidate forms of crash-inducing changes affect the effectiveness of ChangeLocator?

As introduced in the empirical study (See Section 3), we explored three forms of candidate crash-inducing changes from a bucket of crash reports.

Form 1 includes the largest number of candidate changes, and subsumes candidates in Form 2 and Form 3. As we can see from Table 2, candidates in Form 1 can cover more bugs than the other two forms. This indicates that, Form 1 is better than the other two forms, in terms of the capability of locating bugs. However, the size of candidates in Form 1 is larger and would have side-effect in ranking crash-inducing changes. This is because the larger the size, the more difficult for ChangeLocator to rank the crash-inducing changes well among the candidates. Therefore, we need to find an ideal candidate set which achieves good balance between the capability of locating bugs and the ranking quality.

The goal of RQ3 is to evaluate the effectiveness of the above three forms on the performance of ChangeLocator. To answer this question, we run ChangeLocator with different candidates of crash-inducing changes extracted by the three forms, and then compare their performance.

5.3 Evaluation Metrics

ChangeLocator produces a ranked list of changes according to their suspicious score. Developers can then examine the list from top to bottom, and locate the inducing changes for a crash bucket. It is desirable that the crash-inducing changes are ranked higher in the list. We evaluate the performance of ChangeLocator using the following metrics. Note that, since we employ the random over-sampling in the training process, to ensure that the conclusions of our experiment are general, we repeat the sampling process for 100 times, run

ChangeLocator, and calculate the average value of the following evaluation metrics as the final results.

1. ***Recall@N:*** This metric reports the percentage of buckets, whose crash-inducing changes can be discovered by examining the top $N(N$=1,2,3,...) of the returned suspicious list of changes by ChangeLocator. The higher the value, the less efforts required for developers to locate the crash-inducing changes, thus the better performance.
2. ***Mean Reciprocal Rank(MRR):*** This is a statistic for evaluating a process that produces a list of possible responses to a query (Manning et al. 2008). The reciprocal rank of a query is the multiplicative inverse of the rank of the first relevant answer found. The mean reciprocal rank is the average of the reciprocal ranks of the results of a set of queries Q. The formal definition of MRR is as follows:

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{FR_i} \tag{7}$$

   FR_i represents the rank of the first relevant answer found in the returned list. Clearly, the higher the value, the better the performance of locating crash-inducing changes. We use this metric to evaluate the ability of ChangeLocator to locate the first crash-inducing change for a bucket.
3. ***Mean Average Precision(MAP):*** MAP provides a single value measuring the quality of information retrieval performance(Manning et al. 2008). For a single query, it takes all the relevant answers into consideration with their ranks. The average precision is computed as:

$$AvgP = \frac{\sum_{k=1}^{n}(rel(k) * P(k))}{N} \tag{8}$$

   where k is the rank in the sequence of the retrieved answers, n is the total number of the retrieved answers and N is the total number of all relevant answers. In this formula, $rel(k)$ is an indicator function equaling one if the item at rank k is a relevant answer, and zero otherwise. $P(k)$ is the precision at the given cut-off rank k, which is computed as:

$$P(k) = \frac{N_r}{k} \tag{9}$$

   where N_r is the number of relevant answers in the top k of the returned list (Turpin and Scholer 2006).

   For a set of queries Q, the mean average precision is computed as:

$$MAP = \frac{\sum_{i=1}^{|Q|}AvgP(i)}{|Q|} \tag{10}$$

   This metric is used to evaluate the ability of ChangeLocator to locate all the crash-inducing changes for a bucket. Clearly, the higher the value, the better the performance.

6 Evaluation Results

RQ1: Can ChangeLocator effectively locate crash-inducing changes?

Table 4 shows the results of our evaluation. On average, ChangeLocator can sucessfully locate 44.7%, 68.5%, and 74.5% of crash buckets by examining only top 1, 5 and 10 changes. Table 4 also shows the comparison results between ChangeLocator and Locus for

Table 4 Results of ChangeLocator and locus

Subject	Approach	Recall@N			MAP	MRR
		N=1	N=5	N=10		
Netbeans 6.7	ChangeLocator	12 (19.7%)	35 (57.4%)	42 (68.9%)	0.353	0.360
	Locus	6 (9.8%)	22 (36.1%)	36 (59.0%)	0.222	0.233
Netbeans 6.8	ChangeLocator	20 (38.5%)	34 (65.4%)	39 (75.0%)	0.469	0.493
	Locus	8 (15.4%)	23 (44.2%)	28 (53.8%)	0.253	0.265
Netbeans 6.9	ChangeLocator	18 (45.0%)	26 (65.0%)	26 (65.0%)	0.520	0.528
	Locus	3 (7.5%)	12 (30.0%)	20 (50.0%)	0.168	0.182
Netbeans 7.0	ChangeLocator	19 (50.0%)	25 (65.8%)	27 (71.1%)	0.534	0.566
	Locus	3 (7.9%)	12 (31.6%)	21 (55.3%)	0.197	0.203
Netbeans 7.1	ChangeLocator	23 (56.1%)	33 (80.5%)	36 (87.8%)	0.642	0.672
	Locus	2 (4.9%)	14 (34.1%)	24 (58.5%)	0.201	0.205
Netbeans 7.2	ChangeLocator	23 (59.0%)	30 (76.9%)	31 (79.5%)	0.634	0.660
	Locus	4 (10.3%)	13 (33.3%)	19 (48.7%)	0.218	0.218
Average	ChangeLocator	44.7%	68.5%	74.5%	0.525	0.546
	Locus	9.3%	34.9%	54.2%	0.210	0.218

the six subjects. As mentioned above, we use *Logistic* as the classifier in ChangeLocator. The results indicate that ChangeLocator outperforms Locus in terms of all evaluation metrics. The improvement of ChangeLocator over Locus is 380.6% (89 more buckets), 96.3% (87 more buckets) and 37.5% (53 more buckets) for Recall@1, Recall@5 and Recall@10 respectively on average. As for MAP, ChangeLocator outperforms Locus by 150.6% on average, with the improvement ranging from 58.8% to 219.7%. For MRR, the improvement of ChangeLocator over Locus is 151.0% on average, ranging from 54.3% to 227.6%. We also applied the Mann-Whitney U-Test (Mann and Whitney 1947) on the comparison between ChangeLocator and Locus, and the results showed that ChangeLocator outperforms the existing method Locus significantly ($p < 0.01$).

RQ2: How does each feature contribute to the effectiveness of ChangeLocator?

Figure 7 shows the effectiveness of ChangeLocator with each feature, using the default classifier Logistic Regression, in terms of MAP and MRR metrics. We take the average value of evaluation metrics for all the subjects as the evaluation results. Among all these features, RF (Revision Frequency) and RLODC (Revisions's Lines of Added Code) are less effective than the other features, while IADCP (inverse distance to crashing point) and CC (Crash Component) are the most effective ones. It is also shown that, ChangeLocator using all features significantly outperforms the one using only a single feature, with the improvement over the most effective feature by 19.3% and 23.3% in terms of MAP and MRR metrics respectively.

To further verify the contribution of each single feature to the performance of ChangeLocator, we incrementally apply the features RLODC, DLOAC, RF, RLOCC, NAF, IBF, IADCL, ITDCR, CC and ITDCP to ChangeLocator. The order of incrementally applying features is the same as the order of the performance of each feature (from the worst feature to the best one). As shown in Fig. 8, MAP and MRR values are incrementally improved,

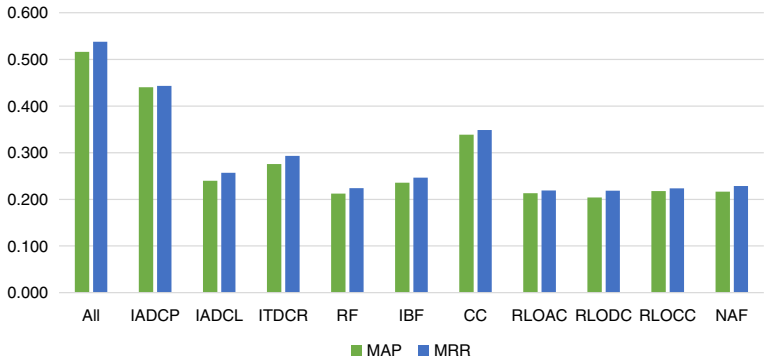


Fig. 7 The effectiveness of ChangeLocator with different features (in terms of MAP and MRR)

and the best performance is achieved when all the features are combined together. Overall, all the features can contribute to the performance of ChangeLocator.

RQ3: How do different forms of candidate crash-inducing changes affect the effectiveness of ChangeLocator?

We run ChangeLocator on the three candidate forms of crash-inducing changes. The results of Recall@N is shown in Fig. 9. Using the candidate Form 3, ChangeLocator achieves the best performance in terms of Recall@N when $N \leq 5$. For example, using Form 3, ChangeLocator can locate 18.3% more buckets than that using Form 1, and 10.9% more buckets than that using Form 2, by examining top 1 change. Using the candidate Form 2, ChangeLocator can achieve slightly better performance than that using the Form 1, with the improvement of 6% in terms of Recall@1. When N is increased to 6, ChangeLocator performs similarly on the three candidate forms.

Figure 10 shows the results of MAP and MRR for the three forms. Similar to Recall@N metrics, using the candidate Form 3, ChangeLocator achieves the best performance in terms of MAP and MRR. For example, the candidate Form 3 outperforms the candidate Form 1 by 10.3% and 9.0%, in terms of MAP and MRR respectively. Besides, the candidate Form 3 also outperforms the candidate Form 2 by 6.9% and 6.0% for MAP and MRR respectively.

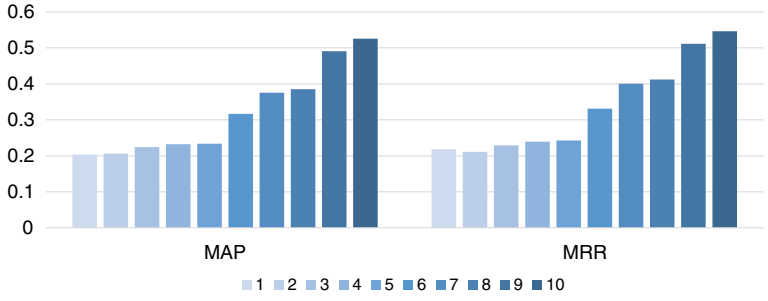


Fig. 8 The contribution of each feature (in terms of MAP and MRR). The label 1-10 in the figure represents the feature sets that incrementally add one feature each time in the order of RLODC, DLOAC, RF, RLOCC, NAF, IBF, IADCL, ITDCR, CC and ITDCP

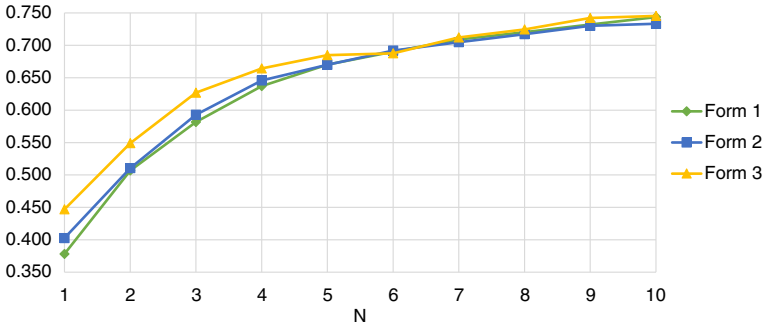


Fig. 9 The recall@N of the three forms of candidate crash-inducing changes

Overall, we can conclude that ChangeLocator performs the best when using the candidate Form 3.

7 Discussions

(1) How is the performance of ChangeLocator in locating crash-inducing changes using different classification algorithms?

As our proposed technique ChangeLocator is a framework in which different classification algorithms can be adopted, we evaluate the effectiveness our ChangeLocator with different classification algorithms. In this RQ, we adopt four popular and representative classification algorithms as the learning model of ChangeLocator, Logistic Regression (*Logistic*), Decision Tree (*J48*), Naive Bayes (*NaiveBayes*), and Bayesian Network (*BayesNet*), which are implemented on top of the open source software WEKA (Weka 2016). We use the default parameter settings of each classifier in WEKA.

Figure 11 shows the results of Recall@N using the four classification algorithms with N ranging from 1 to 10. The results are averaged on the six subjects. It shows that ChangeLocator can locate the crash-inducing changes and rank them as top 1 among the candidate changes for 44.7%, 34.4%, 29.8% and 47.7% of the buckets using *Logistic*, *Naive Bayes*, *Bayesian Network* and *J48* respectively. By examining top 5 recommended changes for each bucket, developers can successfully locate 66.9% to 69.7% of crash buckets using different classifiers. The differences among different classifiers are marginal in terms of Recall@5.

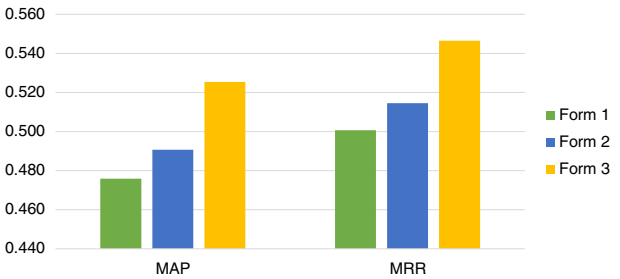


Fig. 10 The MAP and MRR of the three forms of candidate crash-inducing changes

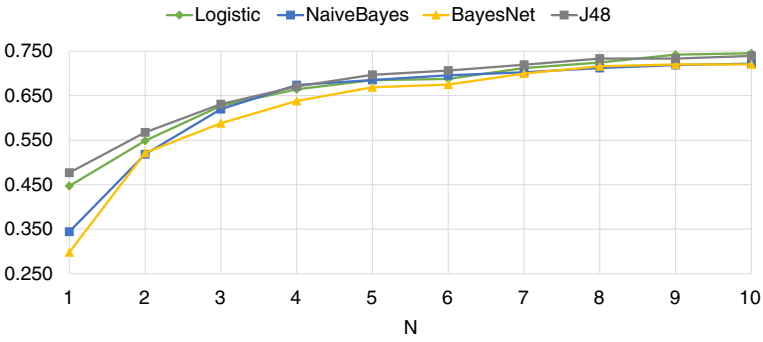


Fig. 11 Recall@N results of different classifiers

Figure 12 shows the performance of ChangeLocator in terms of MAP and MRR using different classifiers. Consistent with Recall@1 metric, *Logistic* and *J48* performs slightly better than the other classifiers. The results suggest that, we can use *Logistic* and J48 as the preferred classifiers in ChangeLocator.

(2) How does the training data size affect the performance of ChangeLocator?

As ChangeLocator is a learning-based approach which requires training data, the training data size may affect the performance of ChangeLocator. For example, as shown in Table 4, ChangeLocator performs worse in the version 6.7 and 6.8, compared with other versions. This is mainly because the size of training dataset for the versions 6.7 and 6.8 is smaller than that of other versions (we only have the training dataset from the version 6.6 when testing on the version 6.7, and the training dataset from 6.6 and 6.7 when testing on the version 6.8). To further evaluate the impact of the size of training dataset, we further conduct a control experiment on the subject Netbeans 7.2 which includes the largest size of training dataset. We use 100% of the buckets from the previous versions (i.e., Netbeans 6.7 to 7.1) by default. In this control experiment, we randomly sampled different ratios of buckets from the training set, from 0.1 to 1.0. Then, we used those sampled buckets as the training set and evaluated the performance of ChangeLocator in Netbeans 7.2. We repeat the sampling process for 100 times at each sampling ratio (from 0.1 to 1.0), and report the average performance on each evaluation metrics.

The results of MAP, MRR, Recall@1, Recall@5, Recall@10 are shown in Fig. 13. The results indicate that, the size of the training set does affect the performance of ChangeLocator initially. The performance of ChangeLocator is increased significantly with the

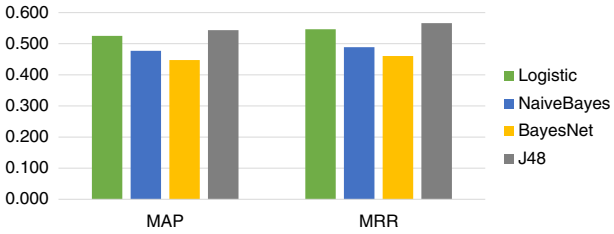


Fig. 12 MAP and MRR results of different classifiers

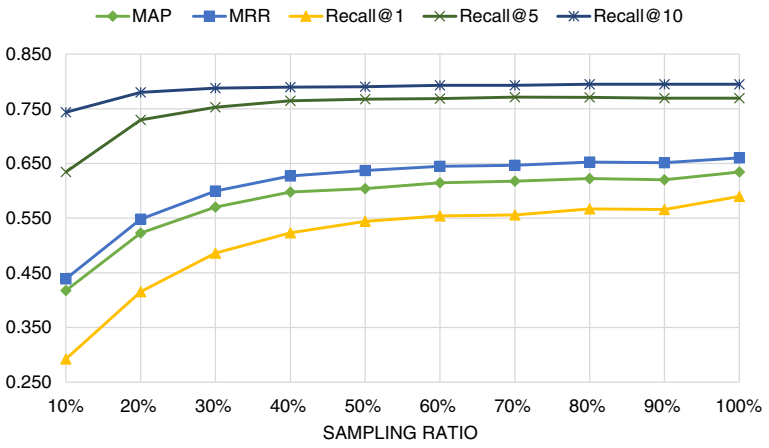


Fig. 13 The performance of ChangeLocator on different training data size in Netbeans 7.2 (using logistic regression classifier)

increasing of the training data size, when training dataset is small. For example, the MAP value is increased from 0.418 to 0.604, with the random sampling ratio of training dataset increased from 10% to 50%. However, when we use 50% of the buckets in the original training set, ChangeLocator can almost get its optimal performance. The results shown in Fig. 13 is evaluated on ChangeLocator using Logistic Regression. We also tried other classification algorithms, and obtained the similar results. This evaluation result indicates that, the effectiveness of ChangeLocator does not require a huge number of historical data for training. In our experiment, 50% of crash buckets from the version 6.5 to 7.1 (about 140 buckets) are sufficient for training an effective model to locate crash-inducing changes. The experimental results indicate that, the effectiveness of ChangeLocator does not require a large number of training instances, and ChangeLocator can perform reasonably well when the number of training instances reaches about 140.

(3) Can we locate crashing bugs at a finer granularity with the help of ChangeLocator?

ChangeLocator can locate crash-inducing changes for crash buckets. A crash-inducing change is a committed revision that developers submitted to source code repository, and it may affect multiple source files and lines of code. The prior studies (Kamei et al. 2013; Wen et al. 2016) showed that predicting or locating bugs at the change level can save much efforts to examine the code. In this study, we also investigate the examination effort required to locate crashing bugs giving the information of crash-inducing changes. As shown in Fig. 14, the median number of source files is 6, which indicates that developers are required to examine less than 6 source files for half of located crashing bugs. The median number of lines of code is 619.5, which indicates that developers are required to examine less than 619.5 lines of code for half of located crashing bugs. In some cases, a crash-inducing change could affect a large number of source files and lines of code. For example, the largest crash-inducing change modified 658 source files and 102,858 lines of code. Such case would result in much manual effort and may not save developers' effort.

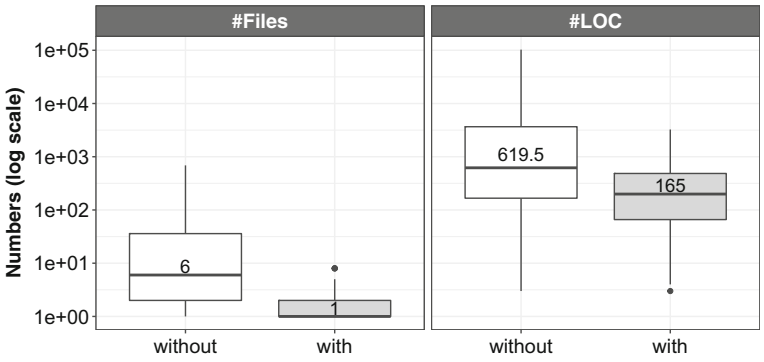


Fig. 14 Examination effort for locating crashing bugs. "without" means crash-inducing changes without crash stacks. "with" means crash-inducing changes with crash stacks

To further reduce the effort, we propose to combine the crash stack information and the crash-inducing changes. Our empirical study and the prior studies (Wu et al. 2014; Wong et al. 2014; Schroter et al. 2010) showed that, many buggy components (including files, methods and crash-inducing changes) reside in crash stacks. Besides, examining the crash stacks is a conventional way for developers to diagnose crashes (Wu et al. 2014). Inspired by this, we only examine the hunks (i.e., a group of contiguous lines that are changed in one commit (Wen et al. 2016)) of the crash-inducing changes which are in the source files covered by crash stacks. In this way, we can significantly reduce the manual effort. As shown in Fig. 14, combining crash stacks with crash-inducing changes, developers are only required to examine 1 source file and 165 lines of code to locate half of the crashing bugs. We conduct the Mann-Whitney U-Test and confirm that the effort (the number of source files and the lines of code) required for examining the hunks of the crash-inducing changes to crash stacks is significantly less than the effort for examining all the crash-inducing changes (p-value < 0.01).

Moreover, we find that, the hunks of crash-inducing changes to crash stacks are usually within the scope of a source file. We conduct the Mann-Whitney U-Test and confirm that the effort for examining the hunks of the crash-inducing changes to crash stacks is significantly less than the effort for examining the entire buggy files (p-value < 0.01). However, examining hunks in crash stacks may miss some bugs, since the buggy code may be executed but not recorded in crash stacks. For example, in our study, examining the hunks in crash stacks fails to locate 25 buckets (9.2% of all the crash buckets) which can be located by examining the entire crash-inducing changes.

In the future, we will consider to further improve ChangeLocator so that it can locate crashing bugs at a finer granularity, such as at the hunk level. We will consider to combine ChangeLocator with the existing crashing bug localization techniques such as CrashLocator (Wu et al. 2014), which can locate bugs that are out of crash stacks.

(4) Why are there some crash buckets that cannot be located by ChangeLocator?

Although ChangeLocator is effective, it still cannot locate all crash buckets. There are in total 23.2% of crash buckets that cannot be located by ChangeLocator. It is mainly because ChangeLocator extracts the candidate inducing changes from only crash stacks and crash-inducing changes may appear out of crash stacks. Our prior study (Wu et al. 2014) showed

that, the buggy code may be popped out during the execution and were not recorded in crash stacks. Similarly, if the crash-inducing changes introduce the buggy code that were popped out during the execution, ChangeLocator will fail to locate such changes. We give an example of such cases as shown in Fig. 15. Crash report 604701 which is caused by Bug #221173, crashed at the line 289 in the source file `InnerToOuterTransformer.java`. However, the buggy statement is at the line 528 of the same file, which was executed and popped out from the method `refactorInnerClass`. The buggy statement was introduced from the revision `a10666d51e59`. However, since this crash-inducing change does not modify the source code of the methods in the crash stack, ChangeLocator fails to locate the inducing change for it.

To locate crash-inducing changes out of crash stacks, in the future we will consider to leverage static call graph analysis adopted in the prior studies (Wu et al. 2014; Seo and Kim 2012) to explore the source code that may be executed before crashes.

8 Threats to Validity

There are potential threats to the validity of our work:

- **Subject selection bias:** We only use the data from Netbeans project in our experiment because of the publicly availability of crash data. Although it is also feasible to collect crash data from bug reporting system in some open source projects (e.g., Eclipse), crash stacks in bug reports are optional and not many users report the crash stacks manually. This makes it difficult to collect a large amount of crash report data. Mozilla also has publicly available crash data. However, we found that it is difficult to collect crash-inducing changes for Mozilla projects. Since developers often commit different bug-fixing patches for a same bug in different branches, this makes the number of crash-inducing changes very large. We manually checked some of the crash-inducing changes and found many of them are noisy. Therefore, we do not include Mozilla in this study. In the future, we will consider to clean the data and include Mozilla projects for evaluation.
- **Oracle dataset:** The oracle dataset is collected based on the existing techniques of collecting bug inducing changes (Kim et al. 2006). To minimize the threats of the data quality problem, we conduct manual validation of the collected data. However, it is still possible that, the oracle dataset may be incomplete or imprecise. In the future, we will investigate the techniques to improve the quality of the oracle dataset.
- **Empirical evaluation:** In this work, we conduct experiments to evaluate the effectiveness of the proposed technique. Although there are some evidences that the bug inducing changes are useful for developers, the usefulness of our technique should be ultimately evaluated by real developers in practice. In the future, we will conduct user study to further evaluate our technique.

9 Related Work

9.1 Crash Analysis

Recently, researchers have been dedicated to analyzing the software crashes in large-scale systems. Microsoft developed WER (Windows Error Reporting System) (Glerum et al. 2009) to automatically collect crash reports from end users to facilitate the debugging. Due to the large number of crash reports received daily, the crash reporting system needs

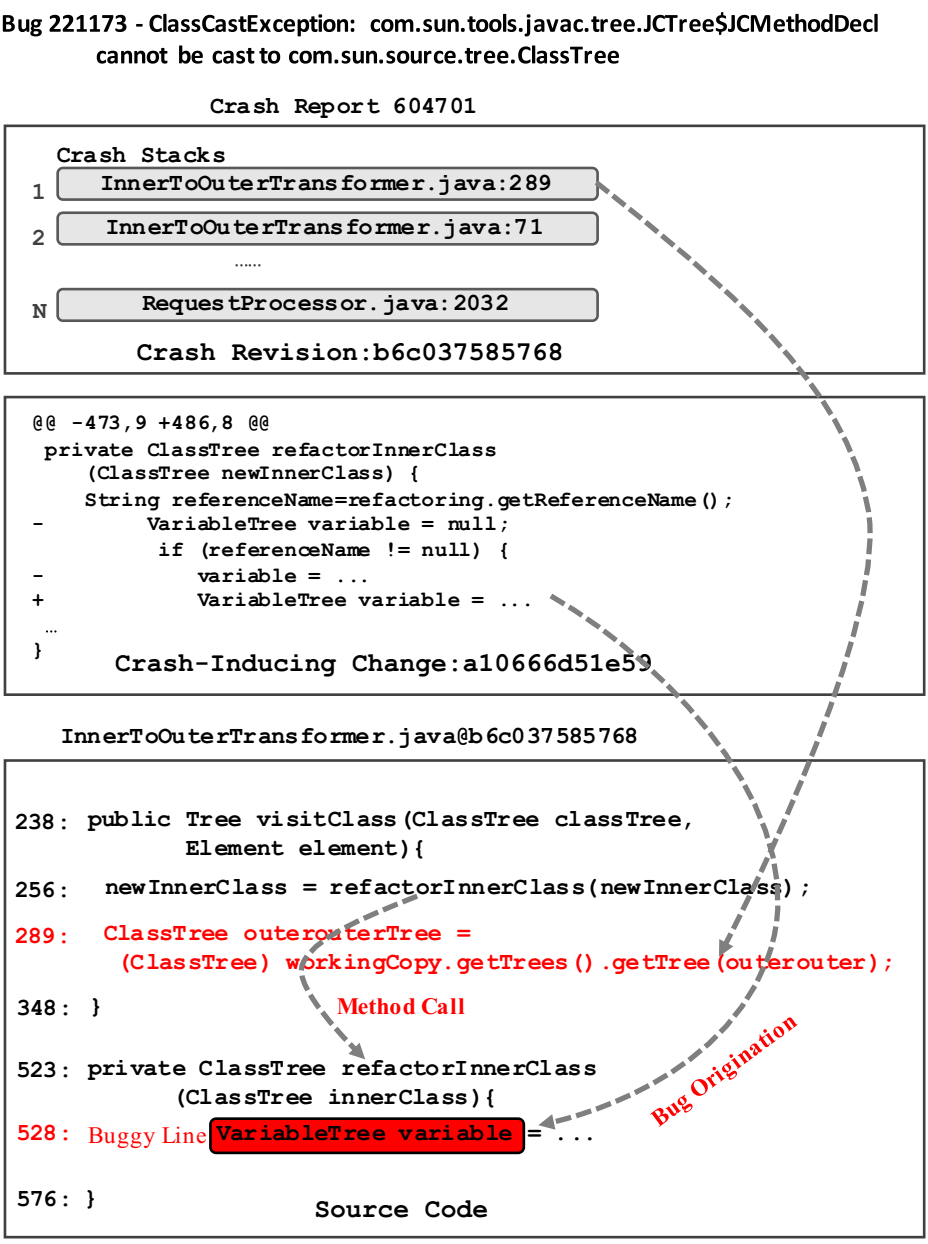


Fig. 15 An example that ChangeLocator fails to locate

to organize the duplicate crash reports into "buckets". Bucketing crash reports has been proved to be useful in practice according to the ten years' operation in WER (Glerum et al. 2009). To further improve the accuracy of bucketing, Kim et al. (2011) proposed to leverage the similarity of crash graphs to identify duplicate crash reports. Dang et al. (2012) proposed a method of bucketing duplicate crash reports based on call stack similarity. To better prioritize debugging efforts, Kim et al. (2011) proposed to predict the top crashes

when crash reporting systems receive a small number of crashes. To facilitate the maintenance of crashing faults, Seo and Kim (2012) proposed to predict the recurring crashes due to incomplete or missing fixes. To improve bug management for crashing bugs, Wang et al. (2016) proposed an algorithm to locate buggy files for crashes via crash correlation groups, as well as a method to identify duplicate and related crashing bug reports.

To facilitate the debugging of software crash, many researches have been working on reproducing crashes. For example, ReCrashJ (Artzi et al. 2008) generates unit tests for crashes by capturing the state of method arguments. Chronicler (Bell et al. 2013) captures the non-deterministic inputs and reproduces the crashes. BugRedux (Jin and Orso 2012) records different kinds of execution data, and reproduces crashes using symbolic analysis. Cao et al. (2014) records the return values of hard-to-resolve functions, and uses concolic execution to generate test cases for reproducing crashes. White et al. (2015) proposed to generate reproducible bug reports for Android application crashes, based on the predefined natural language descriptions for app features and the user execution profile information. Moran et al. (2016) proposed an approach to discover the crashes and generate reproducible bug reports for Android application crashes, which include screenshots and augmented natural language description steps.

The above work mainly studied the collecting and bucketing crash reports, the prediction of crash-prone module, and the reproduction of crashes. Our work also focuses on analyzing software crashes. Different from the above work, we focus on the problem of locating the crash-inducing changes for crash buckets.

9.2 Bug Localization

Bug localization is one of important steps in debugging, which is often tedious and non-trivial. In recent years, various bug localization techniques have been proposed. Spectrum-based bug localization techniques (Abreu et al. 2007; Jones et al. 2002; Liblit et al. 2003, 2005) statistically analyze the passing and failing execution traces of test cases, and rank the suspicious statements. Parnin and Orso (2011) conducted a comparative study on the effectiveness of these techniques by comparing the developers' debugging time with and without the ranked list of suspicious statements. Their study showed that, simply examining faulty statements in isolation may not be sufficient for debugging, and more contexts should be provided. Besides, most of the existing spectrum-based bug localization techniques require many inputs, such as passing and failing test cases, which are not available in existing crash reporting systems. IR-based bug localization techniques (Rao and Kak 2011; Wong et al. 2014; Zhou et al. 2012; Moreno et al. 2014) mainly apply or adapt the information retrieval techniques to query the relevant source files based on the textual descriptions of bug reports. Similar to our work, some of these techniques (Wong et al. 2014; Moreno et al. 2014) utilize crash stack traces as the input to enhance the performance of bug localization. Wang et al. (2015) investigated the usefulness of these IR-based techniques by conducting both experimental study and user study. Their study showed that, the quality of bug reports affect IR-based techniques significantly, and it is not sufficient for developers to understand and fix the bugs with the results produced by IR-based technique. Both of the studies in Parnin and Orso (2011) and Wang et al. (2015) showed that, developers expect more contextual information.

In this paper, we are targeting at locating crash-inducing changes for crashing faults in large-scale systems with crash reporting systems. As shown in our investigation, developers consider crash-inducing changes as useful context information for debugging. Besides, our work only requires source code repository and crash reports, which are easy to obtain in practice.

9.3 Bug Inducing Changes

Bug inducing change is one of important data for software analysis. To identify that, Śliwerski et al. (2005) proposed SZZ algorithm based on the bug-fixing changes. Kim et al. (2006) further improved SZZ algorithm by removing non-semantic changes and outlier fixes. da Costa et al. (2017) proposed a framework to evaluate the results generated by SZZ-based algorithms. Based on the bug inducing changes, researchers conducted many studies. For example, Śliwerski et al. (2005) performed a study on finding out the most bug-prone day. Kim et al. (2006) used the bug inducing changes to analyze the micro pattern changes in source code. Kamei et al. (2013) and Kim et al. (2008) proposed to predict whether a committed change is bug-prone based on the bug inducing changes. An et al. (2017) predicted whether a committed change would induce a crash in the future.

Our work is relevant to the above work, since we leverage the bug inducing changes in the training process. Unlike the above work, our goal is to locate crash-inducing changes for crash buckets. Different from change-based defect prediction work (Kamei et al. 2013; Kim et al. 2008) that cannot tell which bug a bug-prone change is responsible for, our technique can provide a suggested list of changes for a specific crash bucket.

10 Conclusion and Future Work

In this paper, we described an empirical study on crash-inducing changes. Based on the results of our empirical study, we propose ChangeLocator, an automatic tool for locating the crash-inducing changes. To evaluate the effectiveness of ChangeLocator, we conducted an experimental study on six release versions of NetBeans project. Our evaluation results show that, using ChangeLocator, we can locate 44.7%, 68.5%, and 74.5% of crash buckets by examining only top 1, 5 and 10 changes. We also compared our technique with the state-of-the-art bug localization technique Locus. The evaluation results showed that ChangeLocator outperforms the Locus significantly, with the improvement in MAP ranging from 58.8% to 219.7% and the improvement in MRR ranging from 54.3% to 227.6%.

In the future, we will evaluate ChangeLocator using more projects, including industrial projects. We will also conduct user study to evaluate the usefulness and effectiveness of ChangeLocator in practice. Moreover, we will consider to combine ChangeLocator with other bug localization techniques to locate bugs at a finer granularity. Also, applying ChangeLocator to automated program repair is an interesting research direction to explore.

Our tool and the experimental data used in this paper can be accessed via Git with the following url: https://bitbucket.org/rongxin/changelocator-dataset.git

Acknowledgments We thank anonymous reviewers for their insightful comments. This research is supported by Hong Kong SAR RGC/GRF grant 16202917, NSFC grant 61272089, and 2016 Microsoft Research Asia Collaborative Research Program.

References

Abreu R, Zoeteweij P, Van Gemund AJ (2007) On the accuracy of spectrum-based fault localization. In: Testing: academic and industrial conference practice and research techniques-MUTATION, 2007. TAICPART-MUTATION 2007. IEEE, Piscataway, pp 89–98

Al Shalabi L, Shaaban Z, Kasasbeh B (2006) Data mining: a preprocessing engine. J Comput Sci 2(9):735–739

An L, Khomh F (2015) An empirical study of crash-inducing commits in mozilla firefox. In: Proceedings of the 11th international conference on predictive models and data analytics in software engineering. ACM, New York, p 5

An L, Khomh F, Guéhéneuc Y-G (2017) An empirical study of crash-inducing commits in mozilla firefox. Softw Qual J, 1–32

Arcuri A, Yao X (2008) A novel co-evolutionary approach to automatic software bug fixing. In: 2008 IEEE Congress on evolutionary computation (IEEE world congress on computational intelligence). IEEE, Piscataway

Artzi S, Kim S, Ernst MD (2008) Recrash: making software failures reproducible by preserving object states. In: European conference on object-oriented programming, vol 8, pp 542–565

Batista GE, Prati RC, Monard MC (2004) A study of the behavior of several methods for balancing machine learning training data. ACM Sigkdd Explorations Newsletter 6(1):20–29

Bell J, Sarda N, Kaiser G (2013) Chronicler: lightweight recording to reproduce field failures. In: Proceedings of the 2013 international conference on software engineering. IEEE press, Piscataway, pp 362–371

Bug report list (2015) [online]. Available: https://bugzilla.mozilla.org/buglist.cgi?longdesc=regression%20range&longdesc_type=casesubstring&query_format=advanced&short_desc=crash&short_desc_type=allwordssubstr&order=bug_status%2cpriority%2cassigned_to%2cbug_id&limit=0

Cao Y, Zhang H, Ding S (2014) Symcrash: selective recording for reproducing crashes. In: Proceedings of the 29th ACM/IEEE international conference on automated software engineering. ACM, New York, pp 791–802

Mozilla crash reports (2015) [online]. Available: http://crashstats.mozilla.com

da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan A (2017) A framework for evaluating the results of the szz approach for identifying bug-introducing changes. IEEE Trans Softw Eng 43(7):641–657

Dang Y, Wu R, Zhang H, Zhang D, Nobel P (2012) Rebucket: a method for clustering duplicate crash reports based on call stack similarity. In: Proceedings of the 34th international conference on software engineering. IEEE press, Piscataway, pp 1084–1093

Dit B, Revelle M, Gethers M, Poshyvanyk D (2013) Feature location in source code: a taxonomy and survey. Journal of software: Evolution and Process 25(1):53–95

Glerum K, Kinshumann K, Greenberg S, Aul G, Orgovan V, Nichols G, Grant D, Loihle G, Hunt G (2009) Debugging in the (very) large: ten years of implementation and experience. In: Proceedings of the ACM SIGOPS 22nd symposium on operating systems principles. ACM, New York, pp 103–116

Jin W, Orso A (2012) Bugredux: reproducing field failures for in-house debugging. In: Proceedings of the 34th international conference on software engineering. IEEE, Piscataway, pp 474–484

Jones JA, Harrold MJ, Stasko J (2002) Visualization of test information to assist fault localization. In: Proceedings of the 24th international conference on software engineering. ACM, New York, pp 467–477

Kamei Y, Shihab E, Adams B, Hassan AE, Mockus A, Sinha A, Ubayashi N (2013) A large-scale empirical study of just-in-time quality assurance. IEEE Trans Softw Eng 39(6):757–773

Kim S, Pan K, Whitehead EJ Jr (2006) Micro pattern evolution. In: Proceedings of the 2006 international workshop on mining software repositories. ACM, New York, pp 40–46

Kim S, Zimmermann T, Pan K, James E Jr et al (2006) Automatic identification of bug-introducing changes. In: Proceedings of the 21st IEEE/ACM international conference on automated software engineering. IEEE, Piscataway, pp 81–90

Kim S, Zimmermann T, Whitehead EJ Jr, Zeller A (2007) Predicting faults from cached history. In: Proceedings of the 29th international conference on software engineering. IEEE computer society, Washington, pp 489–498

Kim S, Whitehead EJ Jr, Zhang Y (2008) Classifying software changes: clean or buggy? IEEE Trans Softw Eng 34(2):181–196

Kim D, Wang X, Kim S, Zeller A, Cheung S-C, Park S (2011) Which crashes should i fix first?: predicting top crashes at an early stage to prioritize debugging efforts. IEEE Trans Softw Eng 37(3):430–447

Kim S, Zhang H, Wu R, Gong L (2011) Dealing with noise in defect prediction. In: Proceedings of the 33rd international conference on software engineering. IEEE, Piscataway, pp 481–490

Kim S, Zimmermann T, Nagappan N (2011) Crash graphs: an aggregated view of multiple crashes to improve crash triage. In: 2011 IEEE/IFIP 41St international conference on dependable systems & networks. IEEE, Piscataway, pp 486–493

Kotsiantis S, Kanellopoulos D, Pintelas P (2006) Data preprocessing for supervised leaning. Int J Comput Sci 1(2):111–117

Le Goues C, Nguyen T, Forrest S, Weimer W (2012) Genprog: a generic method for automatic software repair. IEEE Trans Softw Eng 1:38

Liblit B, Aiken A, Zheng AX, Jordan MI (2003) Bug isolation via remote program sampling. In: Proceedings of the ACM SIGPLAN 2003 conference on programming language design and implementation. ACM, New York, pp 141–154

Liblit B, Naik M, Zheng AX, Aiken A, Jordan MI (2005) Scalable statistical bug isolation. In: ACM SIGPLAN Notices, vol 40, no 6. ACM, New York, pp 15–26

Mani I, Zhang I (2003) knn approach to unbalanced data distributions: a case study involving information extraction. In: Proceedings of workshop on learning from imbalanced datasets

Mann HB, Whitney DR (1947) On a test of whether one of two random variables is stochastically larger than the other, the annals of mathematical statistics

Manning CD, Raghavan P, Schütze H (2008) Introduction to information retrieval, vol 1. Cambridge University Press, Cambridge

Moran K, Linares-Vásquez M, Bernal-Cárdenas C, Vendome C, Poshyvanyk D (2016) Automatically discovering, reporting and reproducing android application crashes. In: 2016 IEEE international conference on software testing, verification and validation. IEEE, Piscataway, pp 33–44

Moreno L, Treadway JJ, Marcus A, Shen W (2014) On the use of stack traces to improve text retrieval-based bug localization. In: 2014 IEEE International conference on software maintenance and evolution. IEEE, Piscataway, pp 151–160

Moser R, Pedrycz W, Succi G (2008) A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In: Proceedings of the 30th international conference on software engineering. ACM, New York, pp 181–190

Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on software engineering. IEEE, Piscataway, pp 284–292

Nallapati R (2004) Discriminative models for information retrieval. In: Proceedings of the 27th annual international ACM SIGIR conference on research and development in information retrieval. ACM, New York, pp 64–71

Netbeans bugzilla (2015) [online]. Available: https://netbeans.org/bugzilla

Netbeans exception reports (2015) [online]. Available: http://statistics.netbeans.org/analytics/list.do?query

Netbeans report exception faqs (2015) [online]. Available: http://wiki.netbeans.org/usecases

Netbeans source code repository (2015) [online]. Available: http://hg.netbeans.org

Technical note tn2123: Crashreporter (2015) [online]. Available: developer.apple.com/library/mac/#technotes/tn2004/tn2123.html

Parnin C, Orso A (2011) Are automated debugging techniques actually helping programmers? In: Proceedings of the 2011 international symposium on software testing and analysis. ACM, New York, pp 199–209

Prati RC, Batista GE, Monard MC (2004) Class imbalances versus class overlapping: an analysis of a learning system behavior. In: MICAI 2004: advances In artificial intelligence, vol 4, pp 312–321

Regression range (2015) [online]. Available: https://wiki.mozilla.org/firefox_OS/performance/bisecting_regressions

Rao S, Kak A (2011) Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In: Proceedings of the 8th working conference on mining software repositories. ACM, New York, pp 43–52

Robertson SE, Jones KS (1976) Relevance weighting of search terms. Journal of the Association for Information Science and Technology 27(3):129–146

Saha RK, Lease M, Khurshid S, Perry DE (2013) Improving bug localization using structured information retrieval. In: 2013 IEEE/ACM 28Th international conference on automated software engineering. IEEE, Piscataway, pp 345–355

Schroter A, Schröter A, Bettenburg N, Premraj R (2010) Do stack traces help developers fix bugs? In: 2010 7Th IEEE working conference on mining software repositories. IEEE, Piscataway, pp 118–121

Seo H, Kim S (2012) Predicting recurring crash stacks. In: Proceedings of the 27th IEEE/ACM international conference on automated software engineering. ACM, New York, pp 180–189

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? ACM sigsoft software engineering notes 30(4):1–5

Turpin A, Scholer F (2006) User performance versus precision measures for simple search tasks. In: Proceedings of the 29th annual international ACM SIGIR conference on research and development in information retrieval. ACM, New York, pp 11–18

Venkatesh GA (1991) The semantic approach to program slicing. In: ACM SIGPLAN Notices, vol 26, no 6. ACM, New York, pp 107–119

Wang S, Lo D (2014) Version history, similar report, and structure: Putting them together for improved bug localization. In: Proceedings of the 22nd international conference on program comprehension. ACM, New York, pp 53–63

Wang Q, Parnin C, Orso A (2015) Evaluating the usefulness of ir-based fault localization techniques. In: Proceedings of the 2015 international symposium on software testing and analysis. ACM, New York, pp 1–11

Wang S, Khomh F, Zou Y (2016) Improving bug management using correlations in crash reports. Empir Softw Eng 21(2):337–367

Weimer W, Forrest S, Le Goues C, Nguyen T (2010) Automatic program repair with evolutionary computation. Commun ACM 53(5):109–116

Weka (2016) [online]. Available: http://www.cs.waikato.ac.nz/ml/weka

Wen M, Wu R, Cheung S-C (2016) Locus: locating bugs from software changes. In: Proceedings of the 31st IEEE/ACM international conference on automated software engineering. ACM, New York

White M, Linares-Vásquez M, Johnson P, Bernal-Cárdenas C, Poshyvanyk D (2015) Generating reproducible and replayable bug reports from android application crashes. In: 2015 IEEE 23Rd international conference on program comprehension. IEEE, Piscataway, pp 48–59

Witten IH, Frank E, Hall MA, Pal CJ (2016) Data mining: practical machine learning tools and techniques. Morgan Kaufmann, Burlington

Wong C-P, Xiong Y, Zhang H, Hao D, Zhang L, Mei H (2014) Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In: 2014 IEEE international conference on software maintenance and evolution. IEEE, Piscataway, pp 181–190

Wu R, Zhang H, Kim S, Cheung S-C (2011) Relink: recovering links between bugs and changes. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th european conference on foundations of software engineering. ACM, New York, pp 15–25

Wu R, Zhang H, Cheung S-C, Kim S (2014) Crashlocator: locating crashing faults based on crash stacks. In: Proceedings of the 2014 international symposium on software testing and analysis. ACM, New York, pp 204–214

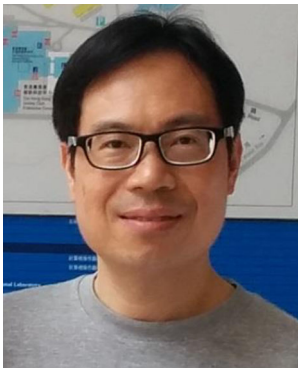
Ye X, Bunescu R, Liu C (2014) Learning to rank relevant files for bug reports using domain knowledge. In: Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering. ACM, New York, pp 689–699

Zeller A (1999) Yesterday, my program worked. today, it does not. why? In: ACM SIGSOFT Software engineering notes, vol 24, no 6. Springer, Berlin, pp 253–267

Zhou J, Zhang H, Lo D (2012) Where should the bugs be fixed?-more accurate information retrieval-based bug localization based on bug reports. In: Proceedings of the 34th international conference on software engineering. IEEE, Piscataway, pp 14–24

Rongxin Wu received the doctoral degree in computer science and engineering from The Hong Kong University of Science and Technology in 2017. Now he is a post-doctoral fellow in The Hong Kong University of Science and Technology. His research interests include program analysis, software crash analysis, and mining software repository. He has served as a reviewer in reputable international journals and a program committee member in several international conferences. He has ever received ACM SIGSOFT Distinguished Paper award. More information about him can be found at: http://home.cse.ust.hk/~wurongxin/.

Ming Wen received his BSc degree from the college of computer science and technology of Zhejiang University (ZJU) in 2014. He is currently a PhD student in the department of computer science and engineering at the Hong Kong University of Science and Technology (HKUST). His research interests include program analysis, mining software repositories, fault localization and repair.



Shing-Chi Cheung received his doctoral degree in Computing from the Imperial College London. He joined the Hong Kong University of Science and Technology (HKUST) where he is a professor of Computer Science and Engineering in 1994. He founded the CASTLE research group at HKUST and co-founded in 2006 the International Workshop on Automation of Software Testing (AST). He was the General Chair of the 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014). He was an editorial board member of the IEEE Transactions on Software Engineering (TSE, 2006-9). His research interests focus on the quality enhancement of software for mobile, web, deep learning, opensource and end-user applications. He is an ACM Distinguished Scientist. More information about his CASTLE research group can be found at http://sccpu2.cse.ust.hk/castle/people.html.

Hongyu Zhang is an Associate Professor with The University of Newcastle, Australia. Previously, he was a Lead Researcher at Microsoft Research Asia and an Associate Professor at Tsinghua University, China. He received his PhD degree from National University of Singapore in 2003. His research is in the area of Software Engineering, in particular, software analytics, testing, maintenance, metrics, and reuse. He has published more than 100 research papers in reputable international journals and conferences. He received two ACM Distinguished Paper awards. He has also served as a program committee member for many software engineering conferences. More information about him can be found at: https://sites.google.com/site/hongyujohn/.