Locus: Locating Bugs from Software Changes

Ming Wen, Rongxin Wu, Shing-Chi Cheung
Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
{mwenaa, wurongxin, scc}@cse.ust.hk

ABSTRACT

Various information retrieval (IR) based techniques have been proposed recently to locate bugs automatically at the file level. However, their usefulness is often compromised by the coarse granularity of files and the lack of contextual information. To address this, we propose to locate bugs using software changes, which offer finer granularity than files and provide important contextual clues for bug-fixing. We observe that bug inducing changes can facilitate the bug fixing process. For example, it helps triage the bug fixing task to the developers who committed the bug inducing changes or enables developers to fix bugs by reverting these changes. Our study further identifies that change logs and the naturally small granularity of changes can help boost the performance of IR-based bug localization. Motivated by these observations, we propose an IR-based approach Locus to locate bugs from software changes, and evaluate it on six large open source projects. The results show that Locus outperforms existing techniques at the source file level localization significantly. MAP and MRR in particular have been improved, on average, by 20.1% and 20.5%, respectively. Locus is also capable of locating the inducing changes within top 5 for 41.0% of the bugs. The results show that Locus can significantly reduce the number of lines needing to be scanned to locate the bug compared with existing techniques.

CCS Concepts

•Software and its engineering → Software testing and debugging; *Software evolution;* **•Information systems → Retrieval models and ranking;**

Keywords

bug localization, software changes, information retrieval, software analytics

1. INTRODUCTION

Bug localization using Information-Retrieval (IR) based techniques [26, 33, 42, 46] has been shown to be effective. It enjoys the advantage of not requiring program traces, which may not be obtainable in real-life systems. The intuition is that bug reports and the corresponding buggy source files share similar tokens, and thus bug locations can be retrieved from bug reports based on the token similarities calculated by various retrieval models [20, 31]. Although the general effectiveness of IR-based techniques has been demonstrated, recent studies [29, 36] have identified two major issues that can greatly hinder the practical usefulness of these techniques. Wang et al. [36] pointed out that existing works produce results at the source file level, which is coarse-grained and still leaves developers with a large amount of code to examine before they successfully find the buggy code. Parnin et al. [29] pointed out that locating buggy modules in isolation may not provide adequate information for developers to understand the bugs, and more contextual clues are desired.

To address these issues, we propose to locate bugs in terms of software changes rather than source files. Our idea is inspired by the observation that bug inducing changes (i.e., the changes which introduce a bug) are one of important clues for developers to understand and fix a bug. We find that bug inducing changes can bring about the following three benefits. First, since a bug inducing change records the developer who committed it, it can help activities like bug triaging. Our empirical study shows that around 70% to 80% of bugs are fixed by the developer who introduced the bug. Second, reverting a bug inducing change is a common way to fix bugs as pointed by existing work [21]. Third, it has been shown that the granularity of software changes is fine-grained [13], and debugging at the change level can significantly save efforts as compared with that at the source file level [13].

Since bug inducing changes are useful for debugging, we then investigated the feasibility of applying IR-based techniques to locate bugs at the change level. While locating bug inducing changes is challenging due to the massive number of changes and the limited information in bug reports, we observe that software changes can indeed facilitate IR-based bug localizations. First, change logs describing the intention or functionality of the changed code often contain substantial information. Common tokens are found between the log of a bug's inducing change and its bug report. These common tokens can be leveraged by IR-based techniques to locate bug inducing changes. Second, change hunks (i.e., a group of contiguous lines that are changed along with contextual unchanged lines) are intrinsically small in size. Bug related changes are associated with small size commits[6], which indicates that using change hunks can be an alternative way to segment source files [39]. This can mitigate the noise problem identified by existing studies [39, 42] in locating bugs using large files. Lastly, the performance of bug localization can be boosted by leveraging change histories. Bug fixing

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...
http://dx.doi.org/10.1145/2970276.2970359

histories inferred from the changes have been widely used for defect prediction [1, 18, 30] and bug localization [42].

Based on these observations, we propose an IR-based bug localization approach called LOCUS, which can locate bugs at both the change and source file levels. Unlike existing techniques, LOCUS retrieves information from software changes instead of source files. Besides extracting natural language tokens from software changes as current techniques [25, 33, 36] do, LOCUS also extracts the code entity names (i.e., package name, class name, method name). The reason is that these entities modified in the changes may alter the behaviors of the system, or even induce bugs. Therefore, two corpora based on natural language tokens and code entity names are created. LOCUS separately indexes these two corpora and generates two rankings by applying the Vector Space Model (VSM), which is explained in Section 3. Apart from these two corpora, LOCUS also leverages the information of change histories to generate an additional ranking as follows. For source file level localization, a suspicious value for each file to contain faults is computed based on its fixing histories. For change level localization, a boosting score for each change is calculated based on its committed time and the bug reporting time. LOCUS later combines these three rankings and generates the final localization results.

We evaluated LOCUS on six large open source projects. For bug localization at the source file level, the results show that it can successfully locate the buggy files and rank them within the top 5 for 68.5% of the bugs and within the top 10 for 74.6%. LOCUS outperforms the current state-of-the-art tools for all subjects. The MAP and MRR are respectively improved by 20.1% and 20.5% on average. To the best of our knowledge, LOCUS is the first IR-based technique to locate bug inducing changes based on bug reports without requiring any knowledge of the associated bug-fixing patches. Although there are some studies for collecting bug inducing changes [17, 34], all of them require the knowledge of *bug fixing patches*, which is unavailable before the bug is fixed. The experiments showed that LOCUS can locate the inducing changes and rank them within the top 5 for 41.0% of the bugs. Our results further show that the number of lines which need to be scanned to locate the bug has been reduced significantly compared with existing approaches. These results are promising and show the effectiveness of LOCUS. In summary, the main contributions of this paper are:

- We conducted an empirical study to show the usefulness of bug inducing changes and the benefits that IR-based bug localization techniques can get from software changes.
- We proposed LOCUS, a new IR-based bug localization method that can effectively locate bugs from software changes. To the best of our knowledge, we are the first to create a text corpus from software change hunks in IR-based bug localization techniques.
- We evaluated LOCUS on six open source projects. The results indicate that LOCUS can locate bugs with high accuracy and outperform the existing state-of-the-art approaches at the source file level. Besides, LOCUS can also get promising results in locating bug-inducing changes.

The rest of the paper is structured as follows. Section 2 introduces the background and motivation. We present our approach and its architecture in Section 3. The experiment setup follows in Section 4, which covers the datasets, evaluation metrics and the four proposed research questions. Section 5 presents the experimental results in detail and answers the four research questions. Threats to validity are covered in Section 6. Related work is introduced in Section 7 and Section 8 concludes this work.

2. BACKGROUND AND MOTIVATION

2.1 Change Hunks

When software evolves, changes are committed to fix bugs, introduce new features or refactor source codes. A committed change could modify multiple files. A file can be modified at one or more places, and each of them is called a **hunk** or a **delta**. A hunk is a group of contiguous lines that are changed along with contextual unchanged lines [6]. Figure 3 shows two hunk examples of source file `WsServerContainer.java`. One introduces a bug and the other fixes the bug. We call those modified lines the **changed lines**, which are highlighted in color in the example. The **contextual lines** are those unchanged lines which are displayed in black in the example.

Changes made by developers can induce new bugs [13, 16, 34, 43]. As a system evolves, the number of bugs induced by committed changes will outnumber those that are induced by the *initial codes* (code introduced in the first version of a source file). Figure 1 shows the number of bugs caused by the initial source files and that by the subsequent software changes of a popular Apache project Tomcat. We extracted all the changes starting from the release time of Tomcat which is Mar-23-2006 to Aug-27-2015. Fixing changes among all these changes were then identified by heuristic [9], and the SZZ algorithm was then applied to identify the inducing changes based on these fixes [34].

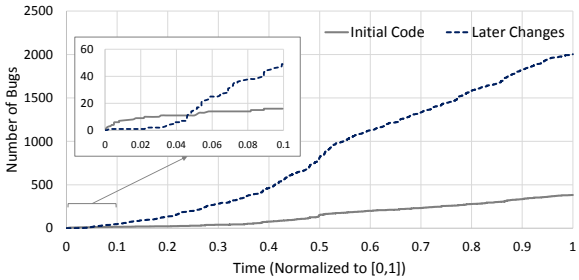

Figure 1: Comparison between the Number of Bugs Induced by Software Changes and the Initial Source Files of Tomcat

After obtaining the inducing changes, we distinguished whether the bug is introduced by the initial code or the subsequent changes. Figure 1 shows that the number of bugs caused by the initial code is larger than that induced by changes at the beginning after the release of Tomcat. However, more bugs are soon induced by the subsequent committed changes, and the gap between the number of the two types of bugs becomes larger and larger.

2.2 The Uses of Bug Inducing Changes

Locating buggy modules in isolation may not provide adequate information for developers to understand and fix a bug [29]. We found that the information of bug inducing changes has the potential to address this issue. First, most of the bugs are introduced by changes as discussed in Section 2.1. Second, bug inducing changes contain important information for developers to understand and fix a bug. We found evidence of this in 761 Apache bug reports and 1733 Eclipse bug

reports that contain discussions about bug inducing changes. The following are extracts of some of the discussions.

"This regression was caused by bugzilla 257440. That patch was reverted and this is now fixed. [1] *"*

"Ouch, that's my fault. ... We should revert revision 484642 ASAP. [2] *"*

To further understand developers' perspective on the usefulness of bug inducing change, we sent emails to developers from open source projects surveying whether bug inducing changes helped fix their bugs and what actions they would take if bug inducing information was available. The following shows some of the representative feedback.

"It can be a eureka moment for the developer, where they see the inducing change and say 'I know exactly why this is happening', thus resulting in a fix typically in a matter of days, even hours. It really is a critical piece of the puzzle."

"The action taken is usually getting the responsible developer to either A) back out the change or B) code and land a follow-up fix as soon as possible. "

Here, we summarize three aspects to show the benefits of using bug inducing change information for bug fixing.

First, since a bug inducing change records the developer who committed it, it facilitates triage the bug to the developer who is the most familiar with the buggy code. To confirm the benefit, we further investigated how many bugs are fixed by the developer who introduced the buggy code.

We studied the ratio of those bugs for three open source projects SWT, JDT and Tomcat. Table 1 shows the detailed results. On average, 77.86% of the bugs were fixed by the committer of the corresponding introducing changes. As bug inducing changes can facilitate triaging bugs automatically, this can save developers' manual efforts in this process. To quantify the efforts saved, we investigated how much time it requires to triage the bugs. We only investigated the JDT and SWT projects, since they recorded the bug triage activities in detail. We followed the existing approach [12] to measure the bug triage time. Figure 2 shows that the bug triage time is non-trivial. The median bug triage time in JDT and SWT is 15 and 42 days respectively. These manual efforts could be saved if bugs can be triaged automatically by leveraging the information of inducing changes.

Table 1: Ratio of Bug Reports Whose Fixing Developer is the Same as the Developer of the Bug Inducing Change

Subject	#Dev	#Bugs	#Same	Ratio (%)
SWT 3.1	86	97	65	67.0%
JDT 4.5	95	94	67	71.3%
Tomcat 8.0	29	193	167	86.5%

Figure 2: Bug Triage Time for JDT and SWT (days)

The second benefit is that reverting bug inducing changes is a common way to fix bugs as pointed out by the existing work [21] and this was also confirmed with developers' feedbacks in our study.

Another benefit is that debugging at the change level can significantly save effort when compared with coarser-grained debugging at source file level [13], since changes and hunks are small in size [6]. We compared the size of hunks, changes and source files in terms of lines code for project SWT, JDT and Tomcat. The median lines of code that a hunk contains is 8 for all three projects. For changes, the median numbers of lines are 27, 54, and 28 for SWT, JDT and Tomcat respectively. Source files contain more lines of code, with the median numbers 48, 136.5 and 125 accordingly. Among these three code elements, hunks offer the finest granularity while source files the coarsest ones.

2.3 Motivations of Using Software Changes in IR-Based Bug Localization

The above findings motivate us to locate the bugs at the change level. In this subsection, we investigate whether it is feasible to use software changes in IR-based bug localization. Case studies are then conducted to show that IR-based techniques can benefit from three aspects: informative change logs, change hunks' fine granularity, and the change histories.

2.3.1 Informative Change Logs

Change logs often contain substantial information that helps infer the intention or the functionality of the changed code. Such information can be leveraged by IR-based techniques. For example, Figure 3 shows the brief summary of bug `#56905` from project Tomcat. The bug reports a problem "unable to destroy `WebSocket` thread group when reloading webapp", and the corresponding buggy file is `WsServerContainer.java`. Commit `2653cea` modified `WsServerContainer.java` during the evolution of Tomcat. From the log message, we know that the intention of this change is to refactor and add new features to the `destroy()` method. It is this change that induced bug `#56905` and the bug was fixed in change `a027afd` as shown in Figure 3. The log of the bug inducing change `2653cea` shares many common tokens with the bug report `#56905` such as "destroy", "thread" and "group". These common tokens are essential to effective bug localization using IR-based techniques.

To understand the correlation between bug reports and the information contained in change logs, we computed the cosine similarities between bug reports and the corresponding buggy files as well as the change logs of the buggy files for all the bugs of the three subjects used in Section 2.2. For each bug, only those changes committed before the bug report was filed are considered. We selected the highest cosine similarity when a bug relates multiple source files or change logs. Figure 4 shows the results. The median values of commit logs are 0.300, 0.369 and 0.370 for SWT, JDT and Tomcat respectively, while the values are 0.191, 0.288 and 0.340 for source files. We conducted Mann-Whitney U-Test [22] to test if the cosine similarities computed from logs are significantly larger than that from source files. The results shows that the differences are significant for all three subjects ($p\text{-}value < 0.05$). These results suggest that change logs share substantial common tokens with bug reports.

2.3.2 Fine Granularity of Change Hunks

The effectiveness of IR-based techniques can be improved by using change hunks due to their granularity being finer than program source files. Recently, researches have found

[1] https://bugs.eclipse.org/bugs/show_bug.cgi?id=262975
[2] https://bz.apache.org/bugzilla/show_bug.cgi?id=41551

```
Bug #56905
Summary: Unable to destroy WebSocket thread group when
reloading webapp …… generally there might be threads that are
still running
```

```
Commit   #2653cea  (inducing change)
Author : Mark Emlyn David Thomas <markt@apache.org>
Date :   Tue Apr 22 08:31:56 2014 +0000
Log: Refactor server container shutdown into the destroy
method. Destroy the thread group on shutdown. Log a warning
if the thread group can't be destroyed
@@ -270,6 +275,21 @@ public void addEndpoint(Class<?> pojo)
+         shutdownExecutor();
+         super.destroy();
+         try {
+             threadGroup.destroy();
+         } catch (IllegalThreadStateException itse) {
...
    boolean areEndpointsRegistered() {
        return endpointsRegistered;
    }
```

```
Commit   #a027afd (fixing change)
Author : Mark Emlyn David Thomas <markt@apache.org>
Date :   Wed Sep 03 13:36:43 2014 +0000
Log: Fix https://issues.apache.org/bugzilla/show_bug.cgi?id=56905
@@ -273,14 +273,42 @@ public void addEndpoint(Class<?> pojo)
+         int threadCount = threadGroup.activeCount();
+         boolean success = false;
          try {
-             threadGroup.destroy();
-         } catch (IllegalThreadStateException itse) {

+             while (true) {
+                 int oldThreadCount = threadCount;
+                 synchronized (threadGroup) {
...
```

Figure 3: Bug Report #56905 of Tomcat and its Fixing Change and Inducing Change (The Bold Texts are the Common Tokens Shared between Bug Report, Commit Logs and Changes)

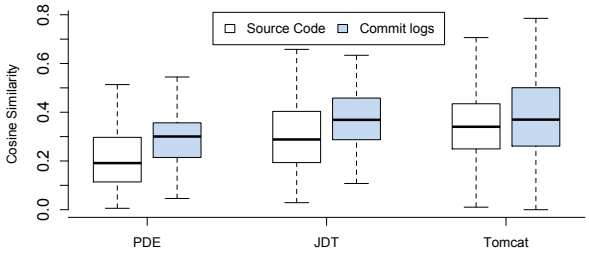

Figure 4: Text Similarities between Bug Reports and the Buggy files as well as the Change Logs

that bugs are usually located in a small portion of the code, and thus large source files are susceptible to noise due to the fuzziness arising from information retrieval [39, 42]. Take the case in Figure 3 as an example, the buggy file `WsServerContainer.java` contains 550 lines in total. However, only a small number of lines (from line 273 to 287) relate to the bug. Figure 5 shows the cosine similarities between the tokens extracted from the bug report and different lines of the source file `WsServerContainer.java`. The figure shows that the text similarities between the bug report and the source file diverge a lot at different lines. The lines relevant to the bug exhibit the highest similarity, while most of the lines that are irrelevant to the bug achieve lower similarities. Therefore, if we treat the whole source file as a unit for querying in the information retrieval model, the noise introduced by the irrelevant lines may degrade the performance of bug localization.

To handle the noise problem, different approaches have been proposed to use small pieces of codes to represent source files by segmentation. Wong et al. proposed dividing a source file into equally sized segments [39], and Ye et al. segmented source files into multiple methods [42]. Both of

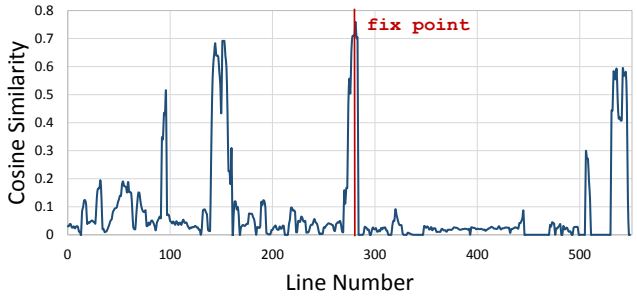

Figure 5: Text Similarities between Bug Report #56905 and Different Lines of its Buggy File WsServerContainer.java of Apache Tomcat

them use the most similar code segments compared with bug reports to represent the file. Their results have shown that the performance of bug localization can be boosted by segmentation. However, segmenting source files into equally sized segments could lose the characteristics of the source codes like code structures [39]. As for segmenting source files into methods, they could still be large in size. As a result, better ways to put highly related code portions into small segments are desired. Change hunks are intrinsically small pieces of code whose content are highly correlated [6] and most of the bugs are induced by them. Therefore, using change hunks instead of source files can mitigate the noise problem existing in large files for IR-based techniques. For instance, if we use the inducing change `2653cea`, we can obtain a high cosine similarity of 0.656 compared to the bug report while only 0.275 can be achieved by using the whole file `WsServerContainer.java` in the previous example.

2.3.3 *Change Histories*

Apart from the textual information, software changes capture the histories of source files such as the alternatives of authorship of a source file, the number of bugs a source file contained previously and the corresponding fixing histories. Such information can indicate the proneness for source files to contain faults [27]. Therefore, information extracted from change histories has been widely leveraged in bug prediction models [11, 18, 27, 30] as well as bug localization model [42] with promising performance. It has also been deployed in large systems in industry such as Google [1]. As a result, the performance of IR-based techniques can be boosted by leveraging change histories.

The empirical studies above showed the usefulness of bug inducing changes and the applicability of applying software changes in IR-based localization techniques. This motivates us to propose a new IR-based bug localization approach.

3. APPROACH

3.1 Vector Space Model

Like existing IR-based bug localization techniques [33, 42, 46], Locus adopts VSM to retrieve bug-related information. In VSM, both queries q and documents d are represented as vectors of weighted terms: $\mathbf{V} = \{w_t | t \in T\}$, T is the corpus of tokens created from all the queries and documents. The similarity between a query q and a document d is computed as the cosine similarity between their term vectors:

$$simi(q, d) = cosine(\mathbf{V_q}, \mathbf{V_d}) = \frac{\mathbf{V_q}^{\mathbf{T}} \mathbf{V_d}}{\|\mathbf{V_q}\| \, \|\mathbf{V_d}\|} \quad (1)$$

The weight w_t for each term is computed based on the classical weighting scheme *term frequency (tf)* and *inverse*

document frequency(idf). The intuition behind this is that the weight of a term normally increases with its appearance frequency in a document and normally decreases with its occurrence frequency in the other documents. Over the years, many approaches for calculating *tf* and *idf* have been proposed to improve the performance of VSM model. Locus follows the approach adopted by BugLocator[46], which was shown to offer better performance:

$$tf(t,d) = \log f_{td} + 1; \quad idf(t) = \log(\frac{N}{n_t})$$
$$w_{td} = (\log f_{td} + 1) \times \log(\frac{N}{n_t}) \tag{2}$$

In equation 2, f_{td} represents the number of appearance of term t in document d. N refers to the total number of documents in the corpus while n_t is the number of documents which contain term t. In existing models, each bug report is treated as a query q and a source file is regarded as a document. When a bug report is received, source files are ranked based on their similarities compared to the bug report calculated by the VSM model. However, as mentioned in Section 2, the effectiveness of the VSM model by regarding source files as single units can be easily affected by the noises contained in large files [39]. In our approach, we treat change hunks as single documents.

3.2 Architecture of Locus

In this study, we proposed a tool called Locus, which **LO**cats bugs from software **C**hange h**U**nk**S**. To create token corpora, existing approaches use a bug report and source files of the version where this bug occurred as the **input**, while Locus uses a bug report and all the changes committed before this bug was reported as the **input**. However, many change hunks extracted are unrelated to source files (e.g., modification in configuration files), have no semantic meaning (e.g. adding/deleting space or comments) or do not alter the behaviors of a system (e.g., code reformatting). A preprocessing step, as shown in Figure 6, is introduced to filter out these irrelevant hunks [14]. Since the entities modified by hunks are more likely to affect the behaviors of the software system or even induce bugs, the names of these changed entities are significant. Therefore, Locus extracts and indexes these code entity names separately besides the natural language tokens. As a result, from the selected hunks after filtering, Locus creates two corpora, which are **NL** (Natural Language) corpus and **CE** (Code Entity) corpus. Each hunk (including the content of **changed lines**, **contextual lines** and the corresponding **commit log**) are indexed as an independent document. Besides, Locus computes another **Boosting** score from the software change repository. At the source file level, Locus extracts the fixing history for each source file and uses it as a feature indicating the suspiciousness of source files to contain faults. At the change level, Locus leverages changes' committed time and favors those that were committed near the occurrence of the bug. Based on the **NL** corpus, the **CE** corpus and the boosting score, we design three ranking models, respectively, to rank suspicious faulty files or bug inducing changes.

When given a bug report, Locus constructs two queries, a *NL query* and a *CE query*. These two queries allow all hunks to be ranked based on the vector space model. The ranking results at the source file level or the change level can be later calculated based on the ranking of hunks and the boosting ranking model. The **output** of Locus is a rank list

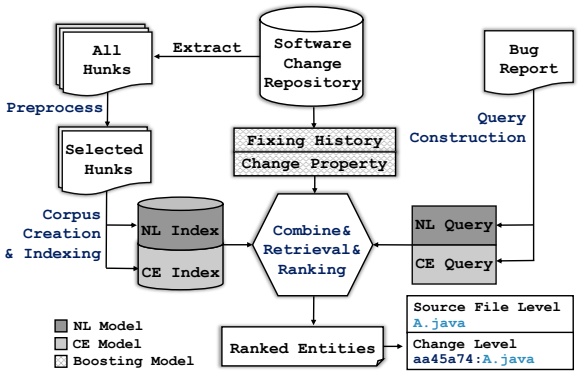

Figure 6: Architecture of Locus (NL Stands for Natural Language, CE for Code Entity, Fixing History is Used in Source File Level Prediction while Change Property is Used in Change Level Prediction)

of source files or changes. Note that, when locating bugs at change level, Locus narrows down the code to be examined for each suspicious change and only outputs the modification content of the most suspicious file in each suspicious change. For example, the modification contents of `A.java` on change `aa45a74` is presented in the suspicious rank.

3.3 Ranking Models

3.3.1 NL Model

The NL corpus is composed of natural language tokens extracted from the selected hunks after preprocessing. Given a bug report, we construct a query using the tokens extracted from the bug *summary* and *description*. Subsequent discussions from developers in the bug report which may include fixing hints (e.g., patched code) are not included as the input of our tool, since our tool is designed to provide debugging hints. This process is the same as general process of corpus creation and query construction. We perform lexical analysis for each hunk and create a vector of lexical tokens. English stop words (e.g. a, the, etc.) and programming language keywords (e.g. `if`, `for`, etc.) are removed. The identifiers which are composed of multiple tokens (e.g. `threadGroup`) are split into two individual tokens (e.g., `thread` and `group`). Finally, the Porter Stemming algorithm [3] is applied to stem a word to its root. After these steps, all the hunks as well as bug reports can be indexed and term weighted vectors can be computed. For a bug report b and a hunk h, the similarity is calculated as follows and *simi* is defined by Equation 1.

$$NL(b,h) = simi(b_{nl}, h_{nl}) \tag{3}$$

3.3.2 CE Model

Let us explain how to construct the CE corpus based on the code entity names appeared in the code repository. *Package names*, *class names* and *method names* are kept in our model, and they are treated as individual tokens without splitting. For example, a class `ThreadGroup` is treated as an individual token in the CE corpus. After creating the corpus, we extracted the code entity names from hunks and bug reports. However, these two artifacts are mainly described by natural languages, and thus extracting code entity names from them is different from that from source files. To do so, we first use the heuristics proposed by Meng et al. [24] to extract the *code like terms* from natural language. Then we compare each code like term with the tokens in the CE

[3]http://tartarus.org/martin/PorterStemmer/

corpus and only the matched ones are kept. We compute the term weighted vectors after indexing both the hunks and bug reports. Similar to that of the NL model, for a source file s and bug report b, the similarity is calculated as:

$$CE(b,h) = simi(b_{ce}, h_{ce}) \quad (4)$$

3.3.3 Boosting Model

Section 2 has shown the usefulness of change histories. At source file level localization, for the sake of simplicity and effectiveness, we adopt the same algorithm used by Google [1] to calculate the boosting score. For a source file s, its suspiciousness score of being buggy is computed as:

$$Fix(s) = \sum_{i=0}^{n} \frac{1}{1 + e^{-12t_i+12}} \quad (5)$$

where n is the number of bug fixing commits for s, and t_i is the timestamp of the ith bug fixing commit. The timestamp used in the algorithm is normalized between 0 and 1, where 0 is the release date of the code base and 1 is the time when predicting, which is the reporting time of the bug to be located in our case.

Existing works have pointed out that changes committed recently contribute the most to fault potential [11, 18]. Therefore, we favor those changes committed near the occurrence of the bug at change level localization. For each source file s, we extract all the changes c that modified s before the bug is reported. We then rank these changes based on their committed time from the oldest to the latest, and denoted it as $rank_{sc}$. For a change c, we compute the boosting score as follows, where $t(c)$ denotes the set of the source files that the change c has modified.

$$Time(c) = \max_{s \in t(c)} \frac{1}{rank_{sc} + 1} \quad (6)$$

3.4 Integrating Ranking Models

3.4.1 Combining Models

At both source file level and change level localization, Locus leverages three models, the NL and CE models serve as the basic units while the boosting model serves as a discriminative factor. At the source file level, the score for a source file s and a bug report b is determined by these three models integrated according to Equation 7. Source file s may be altered by multiple hunks $\{h_1, h_2, ..., h_n\}$. We follow the strategy adopted by existing approaches [39, 42] and select the hunk with the highest cosine similarity to represent the whole source file. Therefore, the final score is computed as follows:

$$score(b,s) = \max_{i=1}^{n}\{NL(b,h_i) + \alpha \times CE(b,h_i)\} + \beta_1 \times Fix(s) \quad (7)$$

Like source files, we choose the hunk with the highest cosine similarity to represent a change c if it comprises multiple hunks $\{h_1, h_2, ..., h_m\}$. As a result, the score between a change c and a bug report b is computed as follows:

$$score(b,c) = \max_{i=1}^{m}\{NL(b,h_i) + \alpha \times CE(b,h_i)\} + \beta_2 \times Time(c) \quad (8)$$

Here, we use the hunk h_p with the highest suspiciousness to represent the change. When we present the results at change level, we DO NOT present all the contents of source files modified by the change. Instead, we only present the modifications of the source file which h_p belongs to.

3.4.2 Determining Parameters

Three parameters, α, β_1, β_2, are involved when combining different models. The parameter α adjusts the weight of the VSM score calculated from the CE corpus, and thus the number of code entity names appeared in a bug report can be an important clue to determine α. Bug reports vary a lot in their quality [15]. The more code entity names a bug report contains, the more weight the CE model should deserve. Based on this intuition, we use the ratio of the number of code entity names compared to the number of split tokens to determine α individually for each bug report. However, the average ratio of code entity names appeared in bug reports is only 10.3% for the subjects we studied. Since the bug report quality is an important factor, we magnify the effect of code entity names by multiplying α with an amplification factor λ. For a bug report, if α is greater than 1 after amplification,we will set it to 1. Our experiments show that λ works well between 3 to 7, and we set it to 5 for all the subjects empirically for consistency.

$$\alpha = \lambda \times \frac{\#code\ entity\ names}{\#split\ tokens} \quad (9)$$

The parameter β captures the weight of the boosting score. Our experiments show that our proposed model performs the best when β_1 is between 0.05 and 0.15 at source file level, and β_2 between 0.15 and 0.25 at change level. β could also be set based on the target projects' characteristics, such as the number of changes, the number of history bugs and so on. We leave it to our future work. In this study, for the consistency and the general effectiveness, we set β_1 to 0.1 and β_2 to 0.2 empirically for all the subjects in our experiments.

4. EXPERIMENT SETUP

4.1 Subjects

To evaluate the performance of our proposed bug localization tool Locus, we selected six open source projects as shown in Table 2. All these projects have well maintained a bug tracking system and change histories. Half of them come from the benchmark datasets collected by Zhou et al.[46], which are ZXing, AspectJ and SWT 3.1. The code repository of the subject Eclipse in the benchmark [46] was previously maintained via CVS and the CVS repository is deprecated nowadays. The repository of Eclipse is now maintained by its sub-project teams separately via Git, and thus we collected two projects from Eclipse: Eclipse JDT Core 4.5 and Eclipse PDE UI 4.4. We also collected another Apache project: Tomcat 8.0. All of these three subjects are popular and large open source projects. We adopt the traditional heuristics [9] to build the links between bug reports and bug fixes. JDT Core 4.5 contains 111 fixed bugs [2] and 94 of them are linked to source codes. PDE 4.4 contains 87 fixed bugs [3] in total and 60 of them are linked. Tomcat 8.0 contains 303 fixed bugs [4] and 193 of them are linked to source files. We manually checked the links between bugs and the corresponding source files later and found out that some of the links may not be applicable to our evaluation oracle. For example, for the bug `70619` of subject AspectJ, one of its linked source file is `modules/tests/bugs/bug70619/Precedence.java`, it is obvious that this source file is designed to test bug `70619` and is not the root cause for this bug. This will inevitably cause bias to the evaluation results. As such, we removed those links from the six subjects whose linked source files are designed as test cases. Due to the removal of these links, the number of bugs in the benchmark subject AspectJ [46] has been reduced to 244.

Table 2: Basic Information of Evaluation Subjects

Subject	#Bugs	#Files	K Loc	K Changes
ZXing	20	391	49.6	3.14
SWT 3.1	98	484	141.9	11.9
AspectJ	244	6,485	511.9	7.7
PDE 4.4	60	5,273	565.2	11.3
JDT 4.5	94	6,775	1,675.3	21.7
Tomcat 8.0	193	2,042	485.7	16.1

4.2 Evaluation Metrics

In order to evaluate the effectiveness of our proposed bug localization model, we adopt the following three metrics, which are widely used to evaluate the performance of bug localization techniques [33, 40, 42, 46].

Top@N: This metric reports the percentage of bugs, whose buggy entities (source files, inducing changes) can be discovered by examining the top $N(N{=}1,2,3,...)$ of the returned suspicious list of code entities. The higher the value, the less efforts required for developers to locate the bug, and thus the better performance.

MRR: *Mean Reciprocal Rank* [35] is the average of the reciprocal ranks of a set of queries. The reciprocal rank of a query is the multiplicative inverse of the rank of the first relevant answer found.

This metric is used to evaluate the ability to locate the first relevant code element for a bug.

MAP: *Mean Average Prevision* [23] is by far the most commonly used traditional IR metric. It takes all the relevant answers into consideration with their ranks for a single query.

This metric is used to evaluate the ability of approaches to locate all the buggy entities of a bug.

4.3 Research Questions

Our experiments are designed to address the following four research questions:

RQ1: Are the tokens extracted from software changes better than those from source files in improving the performance of IR-based bug localization?

Section 2 shows that software changes can benefit IR-based techniques in several aspects. However, can the text tokens extracted from software changes effectively locate bugs? How is the effectiveness compared with the tokens extracted from source files? To answer these questions, we compare the localization results using the text tokens extracted from software changes with those from source files. Note that, given a bug report, the bug localization is conducted on all the changes committed before this bug was reported, as well as the source files (without segmentation) in the version where this bug occurred. We conducted the experiments under three different settings: keeping only the split and stemmed natural language (NL) tokens, keeping only the code entity names (CE) and using both of them (NL+CE), the heuristics to combine NL and CE together is the same as described in Section 3.4. We performed the experiment for all six subjects and compared the results of MAP and MRR.

RQ2: How effective is LOCUS? Does it outperform other bug localization tools?

To answer this question, we apply our approach to locate bugs for the collected six subjects, and then use all the metrics defined above to characterize the effectiveness of the results. BugLocator proposed by Zhou et al. [46] is one of the representative IR-based bug localization techniques, which can locate bugs at source file level. BRTracer is built on top of BugLocator and can achieve better performance by segmenting large files into segments and leveraging stack traces [39]. Saha et al. proposed another technique called BLUiR [33] which outperforms BugLocator by using structured information retrieval. The fixing information of similar bugs has been shown to be useful in bug localization [46]. The underlying intuition is that similar bugs tend to fix similar source files. Therefore, AmaLgam [37] combined this information with the structure information together to improve the performance. We compare Locus with these three state-of-the-art IR-based approaches: BRTracer, BLUiR and AmaLgam. As BLUiR is publicly available, we directly use it in all the evaluation subjects. For BRTracer, we carefully implemented it on top of BugLocator as described in the paper. For AmaLgam, two of its key components, similar bug component and structure components, are provided by BugLocator [46] and BLUiR [33], which are both publicly available. We implemented AmaLgam by combining the components using the parameter as described in their papers.

RQ3: What is the performance of LOCUS to locate bugs at the change level?

Locus is capable of locating bugs at change level. To the best of our knowledge, Locus is the first approach that targets at locating the inducing changes based on the descriptions of a bug report before it is fixed. This research question is designed to evaluate the effectiveness of Locus to locate bugs at this finer granularity. In order for the evaluation, the changes which induced the bugs need to be extracted. SZZ algorithm proposed by Sliwerski et al. [34] can identify the bug inducing changes based on the fix locations, and we adopt this algorithm to extract the oracles for our evaluation. Since multiple files can be modified in a change, we also keep the information of which source file to be blamed for the bug in this change. A successful localization for Locus requires to locate not only the correct change, but also the buggy source file in the change.

Besides the evaluation metrics described in Section 4.2, we further estimate the debugging effort required to locate bugs at the change level compared with that at the file level. We referred to the existing effort-based evaluation method [5] that measures the number of blocks to be inspected until the buggy block is located with a given rank of blocks. Similarly, we measured the number of lines to be inspected until the buggy statement is located of the results generated by Locus, and compared with the state-of-the-art approaches.

RQ4: What is the contribution of each model?

In our approach, we leveraged three different information extracted from change hunks and packaged them into three different models: NL model, CE model and Boosting model. Can all these models contribute to our final performance? In this research question, we aim to evaluate the effectiveness of each model. To evaluate the contributions of these three factors, we conducted three experiments on the six subjects. First, we only used the NL model in Locus to locate all the bugs, and then we added the CE model and combined it with the NL model. Finally, we added boosting model into Locus. Experiments are performed under these three settings, and our purpose is to investigate if the performance can be improved by adding a new model.

5. EXPERIMENT RESULTS

In this section, we answer the four proposed research questions through analyzing the experimental results.

RQ1: Are the tokens extracted from software changes better than those from source files in improving the performance of IR-based bug localization?

Table 3 shows the results of MAP and MRR by using source files or change hunks as the source of text tokens. The results show that no matter using the split natural language tokens, the code entity names or using both, adopting the text tokens extracted from change hunks in IR-based models achieves better results than that from source files for all six subjects. For example, for SWT, the improvements by using only NL are 78.7% and 74.9% for MAP and MRR. If only using CE, the improvements are 44.5% and 55.6%. The improvements are 65.0% and 64.0% for MAP and MRR respectively by combing NL and CE. We later conducted one-tailed statistics test of the results to see if the improvements are significant. The Mann-Whitney U-Test [22] shows that for most of the subjects (the bold text in Table 3), the performance of hunks are significantly better than that of source files ($p\text{-}value < 0.05$). These results prove that the advantages of the software changes discussed in Section 2.3, including the information in change logs and the small granularity, do help locate bugs for IR-based techniques.

Table 3: Comparisons of the Results between Using Source Files and Changes Hunks (The Bold Text Means the Outperformance is Significant)

Subjects		NL		CE		NL+CE	
Text		Source	Hunk	Source	Hunk	Source	Hunk
ZXing	MAP	0.380	0.401	0.258	0.268	0.443	0.458
	MRR	0.429	0.464	0.257	0.268	0.491	0.532
SWT	MAP	0.280	**0.500**	0.314	**0.453**	0.382	**0.630**
	MRR	0.322	**0.563**	0.361	**0.562**	0.430	**0.705**
AspectJ	MAP	0.145	**0.270**	0.140	**0.207**	0.182	**0.303**
	MRR	0.174	**0.323**	0.165	**0.245**	0.218	**0.360**
PDE	MAP	0.207	0.256	0.209	0.349	0.257	**0.398**
	MRR	0.245	0.288	0.255	**0.431**	0.287	**0.472**
JDT	MAP	0.161	**0.252**	0.163	**0.248**	0.206	**0.329**
	MRR	0.211	**0.329**	0.228	**0.321**	0.268	**0.417**
Tomcat	MAP	0.233	**0.441**	0.237	**0.475**	0.299	**0.547**
	MRR	0.276	**0.484**	0.296	**0.544**	0.360	**0.606**

RQ2: How effective is LOCUS? Does it outperform other bug localization tools?

Table 4 shows the results of our approach for all evaluated subjects. It shows that our approach can locate the buggy files and rank them as top 1 among all the source files for 45.0%, 64.3%, 25.0%, 41.7%, 30.9% and 53.9% of the bugs for ZXing, SWT, AspectJ, PDE, JDT and Tomcat respectively. On average, Locus locates the buggy files and ranks them at the top 1 for 41.0% of all the 709 bug reports, and 68.5% within top 5. For 74.6% of the bug reports, Locus locates the correct buggy files within the top 10. The results also indicate the Locus can outperform the three existing works BRTracer, BLUiR and AmaLgam for five subjects SWT, AspectJ, PDE, JDT and Tomcat under all the metrics. For ZXing, Locus achieves the same result as BRTracer under metric Top@1, for other metrics, Locus outperforms all the other approaches.

Specifically, compared with BRTracer, the improvement of MAP varies from 12.8% to 54.6%, and the weighted average improvement is 31.9%. The average improvement of MRR is 32.4%, varing from 6.0% to 46.9% for different subjects. Compared to BLUiR, the improvement of MAP varies from 14.3% to 32.1%, and the weighted average improvement is 22.4%. The average improvement of MRR is 25.3%, which varies from 11.5% to 35.5% for different subjects. Compared to AmaLgam, MAP and MRR are improved by 20.1% and 20.5% respectively on average.

Table 4: Comparisons of the Results at the Source File Level with the State-of-the-Art Approaches

Subjects	Methods	MAP	MRR	Top@1	Top@5	Top@10
ZXing	Locus	**0.502**	**0.563**	**0.450**	**0.700**	**0.800**
	BRTracer	0.445	0.537	**0.450**	0.600	0.700
	BLUiR	0.380	0.490	0.400	0.650	0.700
	AmaLgam	0.410	0.510	0.400	0.650	0.700
SWT	Locus	**0.640**	**0.725**	**0.643**	**0.847**	**0.918**
	BRTracer	0.467	0.512	0.357	0.734	0.857
	BLUiR	0.560	0.650	0.551	0.765	0.867
	AmaLgam	0.578	0.649	0.551	0.776	0.847
AspectJ	Locus	**0.320**	**0.381**	**0.250**	**0.566**	**0.639**
	BRTracer	0.264	0.305	0.225	0.402	0.467
	BLUiR	0.263	0.321	0.213	0.447	0.525
	AmaLgam	0.271	0.335	0.234	0.471	0.545
PDE	Locus	**0.422**	**0.533**	**0.417**	**0.717**	**0.733**
	BRTracer	0.367	0.448	0.367	0.567	0.650
	BLUiR	0.349	0.408	0.300	0.550	0.650
	AmaLgam	0.322	0.392	0.250	0.567	0.683
JDT	Locus	**0.359**	**0.436**	**0.309**	**0.617**	**0.691**
	BRTracer	0.232	0.297	0.191	0.426	0.521
	BLUiR	0.277	0.324	0.191	0.521	0.628
	AmaLgam	0.282	0.334	0.202	0.543	0.606
Tomcat	Locus	**0.566**	**0.638**	**0.539**	**0.777**	**0.819**
	BRTracer	0.408	0.465	0.332	0.648	0.736
	BLUiR	0.459	0.471	0.332	0.668	0.736
	AmaLgam	0.472	0.523	0.399	0.720	0.782

To combine the CE model with the NL model, we set λ to 5 in the above experiments. Figure 7 shows the performance (measured in terms of MAP and MRR) of the six subjects with different λ values. We find that the bug localization performance increases with the increasing of λ when λ is small. This shows that magnifying the effect of the CE model helps improve the performance of bug localization. Locus achieves the optimum performance for the six subjects when λ is between 3 to 7. The improvement slows down or is even decreased when λ is set to 7 or larger. However, the decrease is subtle since we set an upper bound 1 for α and it guarantees that the CE model shares at most the same weight with the NL model. These results are consistent with our intuition that code entities names are important as discussed in Section 3.

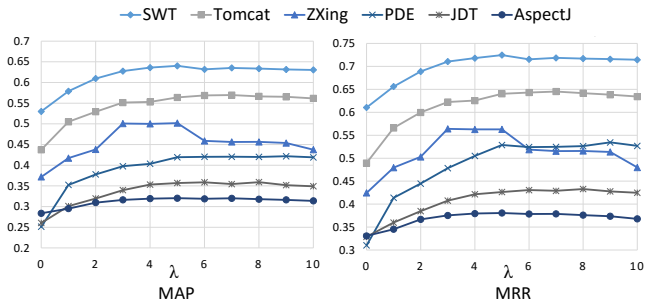

Figure 7: The Effect of Parameter λ

In summary, these results show that Locus is effective in locating bugs at the file level. Locus can outperform the existing state-of-the-art IR-based bug localization approaches.

RQ3: What is the performance of LOCUS to locate bugs at the change level?

To the best of our knowledge, we are the first one to locate bugs at the change level using IR-based techniques. Table 5 shows the results of MAP, MRR and Top@N for

all the subjects. The MAP and MRR are 0.205 and 0.256 respectively on average. Besides, Locus can locate the inducing changes and rank them within the top 5 for 41.0% of bugs. For 51.4% of the bugs, the inducing changes can be ranked within the top 10.

Table 5: Results of MAP, MRR and Top@N at the Change Level

Subject	MAP	MRR	Top@1	Top@5	Top@10	Top@20
ZXing	0.262	0.333	0.200	0.400	0.500	0.900
SWT	0.140	0.224	0.141	0.308	0.436	0.551
AspectJ	0.217	0.315	0.228	0.406	0.506	0.628
PDE	0.219	0.330	0.208	0.479	0.604	0.667
JDT	0.103	0.223	0.162	0.275	0.385	0.474
Tomcat	0.268	0.390	0.276	0.511	0.598	0.701

Figure 8 shows the results produced by Locus at the change level and results by AmaLgam at source file level in terms of the lines of codes need to be inspected (in natural logarithm). Note that, AmaLgam performs better than other approaches at the file level, so we only show the comparison between Locus and AmaLgam. The results indicate that debugging efforts can be saved significantly by using Locus. When using Locus, the median number of lines needing to be inspected has been reduced by an order of magnitude, compared with that using AmaLgam. For example, for all the bugs in Tomcat, the median number of lines is 193 using Locus , while the median number is over 3000 using AmaLgam. To further understand the differences between the results generated by Locus and AmaLgam, we take bug `56199` in Tomcat 8.0 as an example. Both Locus and AmaLgam will rank the relevant element in the top 1. However, the result generated by Locus is `JspCServletContext.java` in the change `05c84ff`, which includes only 32 lines of code. The result generated by Amalgam is `JspCServletContext.java`, which includes 864 lines of code. However, this is only an estimation of the efforts required to locate bugs. Developers may not investigate suspicious source files line by line when debugging in practice, and debugging habits are developer-specific. Therefore, to evaluate the efforts saved precisely requires user studies involving real developers, and we leave it as our future work.

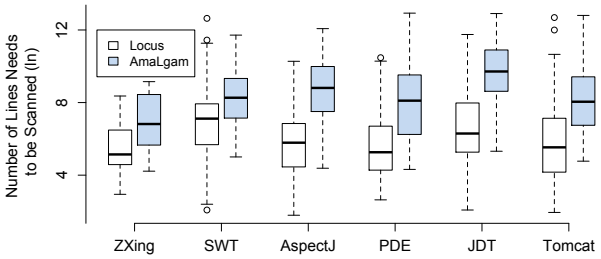

Figure 8: Comparison between Locus and Amalgam, in terms of the Effort-Based Evaluation.

Locating bugs at the change level is worth exploring and the results shown in Table 5 and Figure 8 are promising overall. However, the massive number of change hunks and the prevalent similar changes [38] make it a challenge task to locate the exact inducing change hunks. We plan to leverage more properties of software changes such as change patterns [10], authorship and their proneness to contain faults [13, 16]. This information can be integrated into our localization model to improve the results. Besides, we can excavate deep relations between software changes and bug reports by leveraging the information of stack traces if found in bug reports. We leave the study of such additional software change properties to our future work.

RQ4: What is the contribution of each model?

Figure 9 shows the results under the three settings as described in the experiment setup. Using only the natural language information at the source file level, the average MAP and MRR are 0.348 and 0.402, respectively. The results can be improved by 23.6% and 23.7% on average when incorporating our proposed CE model. This demonstrates the usefulness of the CE model. By adding the information of fixing history, the results of MAP and MRR can be further improved by 5.1% and 5.7%, respectively, on average. At the change level, the average MAP and MRR are, respectively, 0.160 and 0.259 when using only the NL model. The results can be improved by 24.5% and 18.8% after combining the CE model. The information of change time can improve the results by 3.8% and 2.8% on average in a further step. In summary, the results show that each model has contributed to the performance of bug localization.

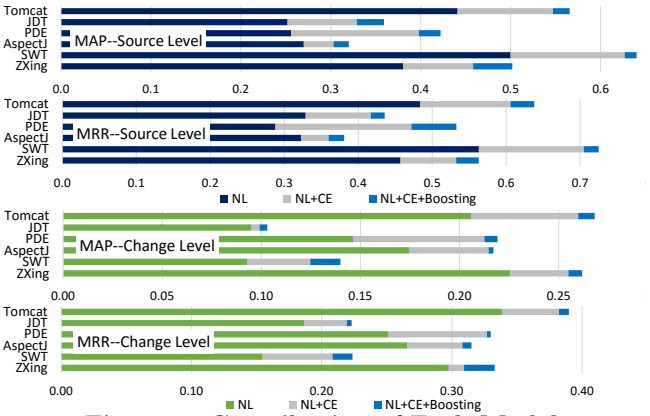

Figure 9: Contribution of Each Model

6. THREATS TO VALIDITY

Our experiments are subject to several threats to validity.

Subject Selection Bias: Six popular open source projects are used in our study and evaluation. Similar results were found among these projects, suggesting that our findings are generalizable. However, Locus performs the worst on subject JDT and AspectJ (MAP<0.4) as shown by Table 4 while these subjects contain the largest number of source files. This raises the concern that our findings may not generalize to very large subjects. Therefore, evaluation on more subjects are desired to increase the confidence for our findings. Besides, the nature of open source projects may differ from that of commercial projects. To validate if our findings are generalizable to commercial projects, experiments on these projects are required. We leave this as our future work.

Data Quality: The quality of bug reports collected may vary a lot for different subjects, and some subjects may contain more reports submitted by developers than end users. The effect of bug reports' quality on our final results should be investigated in a further step. However, it's hard to differentiate between bug reports submitted by developers and that by users in practice. Another threat is that the links between bug reports and software artifacts in the datasets may not be well maintained [41]. To mitigate this threat, we adopted the benchmark dataset including the bug reports and the linked source files from BugLocator [46] for three of our studied subjects. The dataset is widely evaluated

by existing studies [33, 39]. For the other three subjects collected by ourselves, the heuristics adopted to recover the links are widely used by existing works [33, 39, 42, 46].

Empirical Evaluation: We designed several experiments in this study to evaluate the effectiveness of LOCUS. However, the practical usefulness of LOCUS should be validated by developers through their debugging tasks. The carrying out of a user study is left as an important future work.

7. RELATED WORK

7.1 IR-Based Bug Localization

Lots of approaches have been proposed to locate bugs automatically using IR-based techniques [15, 20, 26, 28, 33, 37, 39, 46]. Zhou et al. proposed BugLocator that combines similar bugs with a revised VSM model, which considers the length of source files, to rank relevant source files for a bug report [46]. Wong et al. found that large source files may contain noises, and thus they segmented source files into equal-sized segments and chose the segments with the highest similarity to represent the whole source file [39]. They also considered the information of stack traces which may be found in bug reports. Moreno et al. also found that using the information of stack traces can boost the performance of IR-based techniques [25]. Saha et al. proposed a tool named BLUiR which considers the structure information of source files [33]. Similar to our approach, they considered code entity names in their ranking model. However, our approach is different from them in the following aspects: first, they only extract the identifiers from the stack traces or code snippets in bug reports while we adopted the heuristics [24] that can extract code entity names from general artifacts written in natural languages like bug reports or commit logs. Second, they indexed full identifiers as well as split tokens together while we treated split natural language tokens and code entity names as two corpora and indexed them individually, the VSM scores were calculated separately and were then combined together. The approaches discussed above are based on VSM model. There are other IR-based techniques using different models. Lukins et al. found that Latent Dirichlet Allocation can successfully be applied to source code retrieval for the purpose of bug localization [20]. LDA was also extended in BugScout [28] to narrow down the search space of buggy files given a bug report.

Information retrieval models have also been combined with other techniques to improve the performance of bug localization. The combination with learning-to-rank was proposed by Ye et al [42]. They extracted six features from the given bug reports and source files with domain knowledge such as lexical similarity, bug fixing recency and so on. A learning model was trained on historical fixed bugs and was then used to locate relevant files for newly received bug reports. The combination with deep learning was proposed by Lam et al.[7]. They leveraged deep neural network to relate the terms in bug reports to potentially different code tokens and terms in source files. These approaches require training a model from historical data while LOCUS does not. The combination with spectrum-based localization was proposed by Le et al.[19]. They found that by considering program spectra which are traces of program elements executed under different test cases, the performance of bug localization can be improved. However, this technique requires the availability of test cases and execution traces of subjects.

7.2 Debugging at Software Change Level

Our empirical study shows that many bugs are induced by software changes, and thus assuring the quality of software changes is important. Change impact analysis has been well studied [8, 32], aiming to select a subset of regression test suite that might be affected given a change and then identify program edits that induced the test failures. Delta debugging was proposed [44] to locate a subset of the history that may contributes to a test failure. Combining test spectra (passed or failed test traces) and change impact analysis, FAULTTRACER was able to locate the failure-inducing program edits [45]. Different from these works, LOCUS does not require any regression test suites or test spectra, it is designed to locate the inducing changes based on the descriptions of bug reports and change properties. Thomas et al. studied the properties of bug inducing changes in large open source projects including Mozilla and Eclipse [34]. They found that bug-fixing changes as well as those changes committed on Fridays have higher chances to induce new bugs. Kim et al. proposed an approach to automatically predict whether a change is buggy or clean using machine learning classification algorithms[16]. They extracted features from source codes, change logs and change metadata and trained learning models using historical data. Newly committed changes can be classified into either clean or buggy by the model. Kamei et al. conducted an empirical study of just-in-time quality assurance [13], which concerns defect prediction at the change level. They pointed out that the fine granularity of prediction at the change level can save large efforts over coarser grained predictions at the source file level.

8. CONCLUSION AND FUTURE WORK

The practical usefulness of existing IR-based bug localization techniques is greatly compromised by the coarse granularity of files and the lack of contextual information. We observed that bug inducing changes can help developers in debugging, and software changes can benefit IR-based bug localization techniques. Inspired by our observation, we proposed an approach LOCUS, which locates bugs in terms of software changes instead of source files. It creates two individual corpora composed of natural language tokens and code entity tokens, respectively. It leverages the information of change histories. Experimental evaluation on six popular open source projects shows that LOCUS can locate the relevant files within top 5 for 68.5% of the bugs and 74.6% within top 10 on average. Our approach outperforms three state-of-the-art approaches. LOCUS can also locate the bug inducing changes within top 5 for 41.0% of the bugs, which is very promising.

In the future, we plan to investigate if software changes can also improve the performance of bug localization based on other IR models besides VSM. We also plan to incorporate more properties of software changes such as change patterns [10], authorship and their proneness to contain faults [16] into LOCUS to filter out uninteresting changes, and hence to more accurately locate bug inducing changes.

9. ACKNOWLEDGMENTS

The research was supported by RGC/GRF Grant 611813 of Hong Kong and 2016 Microsoft Research Asia Collaborative Research Grant. We would like to thank the anonymous reviewers for their valuable comments.

10. REFERENCES

[1] http://google-engtools.blogspot.hk/2011/12/bug-prediction-at-google.html. Accessed: 2015-03-22.

[2] https://bugs.eclipse.org/bugs/buglist.cgi?classification=Eclipse&component=Core&list_id=11582065&product=JDT&query_format=advanced&resolution=FIXED&version=4.5. Accessed: 2015-03-22.

[3] https://bugs.eclipse.org/bugs/buglist.cgi?classification=Eclipse&component=UI&list_id=11582038&product=PDE&query_format=advanced&resolution=FIXED&version=4.4. Accessed: 2015-03-22.

[4] https://bz.apache.org/bugzilla/buglist.cgi?product=Tomcat%208&query_format=advanced&resolution=FIXED. Accessed: 2015-03-22.

[5] R. Abreu, P. Zoeteweij, and A. J. Van Gemund. On the accuracy of spectrum-based fault localization. In *TAIC-PART'07*, pages 89–98, 2007.

[6] A. Alali, H. Kagdi, J. Maletic, et al. What's a typical commit? a characterization of open source software repositories. In *ICPC'08*, pages 182–191. IEEE, 2008.

[7] H. A. N. An Ngoc Lam, Anh Tuan Nguyen and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports. In *ASE'15*, pages 151–160. IEEE, 2015.

[8] S. A. Bohner. Software change impact analysis. 1996.

[9] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *ASE'07*, pages 433–436. ACM, 2007.

[10] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on Software Engineering*, 26(7):653–661, 2000.

[12] G. Jeong, S. Kim, and T. Zimmermann. Improving bug triage with bug tossing graphs. In *FSE'09*, pages 111–120. ACM, 2009.

[13] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[14] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *ICSE'11*, pages 351–360. ACM, 2011.

[15] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610, 2013.

[16] S. Kim, E. J. Whitehead Jr, and Y. Zhang. Classifying software changes: Clean or buggy? *IEEE Transactions on Software Engineering*, 34(2):181–196, 2008.

[17] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr. Automatic identification of bug-introducing changes. In *ASE'06*, pages 81–90. IEEE, 2006.

[18] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *ICSE'07*, pages 489–498. IEEE Computer Society, 2007.

[19] T.-D. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: better together. In *FSE'15*, pages 579–590. ACM, 2015.

[20] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.

[21] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *NSDI'13*, pages 57–70, 2013.

[22] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[23] C. D. Manning and H. Schütze. *Foundations of statistical natural language processing*, volume 999. MIT Press, 1999.

[24] S. Meng, X. Wang, L. Zhang, and H. Mei. A history-based matching approach to identification of framework evolution. In *ICSE'12*, pages 353–363. IEEE, 2012.

[25] L. Moreno, W. Bandara, S. Haiduc, and A. Marcus. On the relationship between the vocabulary of bug reports and source code. In *ICSE'13*, pages 452–455. IEEE, 2013.

[26] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen. On the use of stack traces to improve text retrieval-based bug localization. In *ICSME'14*, pages 151–160. IEEE, 2014.

[27] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *ICSE'08*, pages 181–190. IEEE, 2008.

[28] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *ASE'11*, pages 263–272. IEEE, 2011.

[29] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ISSTA'11*, pages 199–209. ACM, 2011.

[30] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu. Bugcache for inspections: hit or miss? In *FSE'11*, pages 322–331. ACM, 2011.

[31] S. Rao and A. Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *MSR'11*, pages 43–52. ACM, 2011.

[32] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.

[33] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE'2013*, pages 345–355. IEEE, 2013.

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

[35] E. M. Voorhees et al. The trec-8 question answering track report. In *Trec*, volume 99, pages 77–82, 1999.

[36] Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of ir-based fault localization techniques. In *ISSTA'15*, pages 1–11. ACM, 2015.

[37] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *ICPC'14*, pages 53–63. ACM, 2014.

[38] S. Wang, D. Lo, and X. Jiang. Understanding widespread changes: A taxonomic study. In *CSMR'13*, pages 5–14. IEEE, 2013.

[39] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *ICSME'14*, pages 181–190. IEEE, 2014.

[40] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim. Crashlocator: locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 204–214, 2014.

[41] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *FSE'11*, pages 15–25. ACM, 2011.

[42] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *FSE'14*, pages 689–699. ACM, 2014.

[43] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *FSE'11*, pages 26–36. ACM, 2011.

[44] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[45] L. Zhang, M. Kim, and S. Khurshid. Localizing failure-inducing program edits based on spectrum information. In *ICSM'11*, pages 23–32. IEEE, 2011.

[46] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE'12*, pages 14–24. IEEE, 2012.