5-2021

Automatic solution summarization for crash bugs

Haoye WANG
Zhejiang University

Xin XIA
Monash University

David LO
Singapore Management University, davidlo@smu.edu.sg

John C. GRUNDY
Monash University

Xinyu WANG
Zhejiang University

Automatic Solution Summarization for Crash Bugs

Haoye Wang*, Xin Xia†‖ David Lo‡, John Grundy† Xinyu Wang*‖
*College of Computer Science and Technology, Zhejiang University
†Faculty of Information Technology, Monash University
‡School of Information Systems, Singapore Management University
{why_,wangxinyu}@zju.edu.cn, {xin.xia,john.grundy}@monash.edu, davidlo@smu.edu.sg

***Abstract*—The causes of software crashes can be hidden anywhere in the source code and development environment. When encountering software crashes, recurring bugs that are discussed on Q&A sites could provide developers with solutions to their crashing problems. However, it is difficult for developers to accurately search for relevant content on search engines, and developers have to spend a lot of manual effort to find the right solution from the returned results. In this paper, we present CRASOLVER, an approach that takes into account both the *structural information of crash traces* and the *knowledge of crash-causing bugs* to automatically summarize solutions from crash traces. Given a crash trace, CRASOLVER retrieves relevant questions from Q&A sites by combining a proposed position dependent similarity – based on the structural information of the crash trace – with an extra knowledge similarity, based on the knowledge from official documentation sites. After obtaining the answers to these questions from the Q&A site, CRASOLVER summarizes the final solution based on a multi-factor scoring mechanism. To evaluate our approach, we built two repositories of Java and Android exception-related questions from Stack Overflow with size of 69,478 and 33,566 questions respectively. Our user study results using 50 selected Java crash traces and 50 selected Android crash traces show that our approach significantly outperforms four baselines in terms of relevance, usefulness, and diversity. The evaluation also confirms the effectiveness of the relevant question retrieval component in our approach for crash traces.**

I. INTRODUCTION

Software crashes are a serious software defect problem and often requires developers to solve them with a high priority. However, as the complexity of a software system increases, the reasons for software crashes become ever more complicated. Fortunately, many bugs are recurring, and occur in different projects but are similar [1]. Previous research [2] [3] reported that there are about 17-45% of total bugs that can be considered as recurring. Therefore, the recurring bugs which have already been discussed on Q&A sites, such as Stack Overflow (SO), could potentially help developers solve their own software crashes.

Most mainstream programming languages have their own exception handling mechanism that produces *crash traces*, which can then be used for further investigation. Figure 1 shows an example of such a crash trace. We refer to red parts as *crash reasons* and blue parts as *stack frames*.

Typically, developers will organize their questions about a crash into a query to the search engines. However, for a large crash trace thrown by a program, it is often difficult for a developer to summarize this crash trace into a query. Due to limitations of input length, search engines like Google and Stack Overflow can not deal with the entire crash trace. In general, developers will try to directly use the *crash reasons*, as shown in Figure 1, as search engine input, then read the returned posts to find solutions to their software crash. However, this method will miss a lot of important information in *stack frames* and the result may not be accurate enough. To make matters worse, there is much noisy and redundant information in the returned posts, and developers need to spend a lot of time to digest and find the right solution.

Fig. 1. An example of a crash trace.

An automated approach considering both *crash reasons* and *stack frames* to provide solutions to a crash trace will help developers solve crash bugs more effectively. Recently, two approaches have been proposed to automatically generate solutions for programming questions: AnswerBot [4] and CROKAGE [5]. AnswerBot [4] generates a query-focused multi-answer-posts summary for a given technical question. It aims to help developers quickly capture the key points of several answer posts relevant to a technical question before they read the details of the posts. CROKAGE [5] takes the natural language description of a programming task and provides a comprehensive solution for the task. CROKAGE suggests programming solutions containing both code and explanations. However, both the approaches are not designed for crash trace analysis and require a natural language description of the programming problem as input, which is hard for developers. Several techniques use crash trace for bug localization [6]–

‖Corresponding author.

[11], automated program repair [1] [12], and duplicate bug report detection [13]–[16] (see Section VI for more detail). However, to the best of our knowledge, automatic generation of *explanations* that can guide a developer to fix the problem causing a given crash trace has not yet been investigated.

Unlike the question answering task [17]–[19], a crash trace may share no lexical units with an answer and the information they convey may be totally different. To address this, we utilize the fact that some Q&A posts discussing a crash bug usually have a crash trace attached, which contains certain information about the crash. We can retrieve related questions by measuring the similarities between crash traces. However, many modern programs depend on common packages and thus different crash traces may contain lots of similar tokens. Two different crash bugs may share many identical tokens in different positions of their crash traces if the programs rely on some common packages. Thus, traditional information retrieval methods, i.e., BM25 [20], TF-IDF based information retrieval [21], word-embedding based information retrieval [22], and document-to-vector based information retrieval [23], will bring bias and not work very well. The *crash reasons* and the structural information of *stack frames* may be the key points to improve our retrieval ability. Additionally, some crash traces on Q&A sites may be incomplete, as the questioner only pastes the *crash reason* and describes the crash in short. Only using crash traces will not be able to retrieve all related Q&A posts. If we can introduce some additional knowledge about the crash bugs to retrieve, it will greatly improve our approach. Unfortunately, it is easy for developers to have a concept what a crash trace is about but it is difficult for a program.

To address these we propose CRASOLVER (**Cra**sh **Solver**) that takes a crash trace as input and then returns a solution summary for the bug. Our approach takes into account both the structural information of the crash traces and the knowledge of crash bugs. First, we extract all crash traces from question bodies on Stack Overflow and use BM25 [20] to retrieve a set of relevant questions based on the *crash reasons* part of every crash trace. Then, we propose two metrics to evaluate the similarities between given crash trace and all posts in the question set: (i) a proposed position dependent similarity based on *stack frames* between a given crash trace and question's crash trace; (ii) an extra knowledge similarity measure, being the similarity between the knowledge from official documentation of a *crash reason* and the question's title. The sorted top-k question list and the answers belonging to these questions are returned. We then score these answer paragraphs according to several kinds of features, and we borrow ideas from the Maximal Marginal Relevance algorithm [24] to generate solution summaries. Finally, we filter some low quality sentences out and output the solution.

We built two repositories for Java questions and Android questions from Stack Overflow where there is crash trace in the question body, of size of 69,478 and 33,566 respectively. We randomly selected another 50 Java questions and 50 Android questions and extracted all crash traces in these question posts. We chose four approaches as our baselines – Google Search Engine, SO Search Engine, AnswerBot [4], and CROKAGE [5]. Our experiments demonstrate that CRASOLVER significantly outperforms all baselines in terms of relevance, usefulness and diversity. The evaluation of the retrieval component in our approach shows strong ability to search related questions on Q&A sites for a given crash trace.

The main contributions of this paper include:

- We propose a novel approach that automatically summarizes solutions for crash traces;
- We introduce two similarity metrics for retrieving relevant questions related to crash traces, position dependent similarity and extra knowledge similarity;
- We conduct extensive experiments to evaluate CRASOLVER, demonstrating its effectiveness compared to state of the art approaches.

The rest of this paper is organized as follows. Section II describes the details of proposed approach. Section III describes data preprocessing and the experiments. Section IV discusses the advantages and disadvantages of CRASOLVER. Section V elaborates the threats to validity and Section VI reviews the related work. Finally, Section VII concludes the paper and mentions future work.

II. PROPOSED APPROACH

We propose a new approach, CRASOLVER, to generate potential solution summaries for a given crash trace thrown by a program. Its framework is illustrated in Figure 2. CRASOLVER takes a crash trace as input and produces a solution summary for the crash trace. The details of its preprocessing and preparation phase, relevant question retrieval phase and solution generation phase are described below.

A. Preprocessing and Preparation

To find questions with a crash trace on Stack Overflow, we first manually checked a large number of exception-related questions on Stack Overflow. We observe that most crash traces are highlighted with the HTML tag $\langle code \rangle$ or $\langle blockquote \rangle$ and contain the string "Caused by" or "Exception in". We thus use Beautiful Soup [25] and regular expressions to extract all questions with crash traces based on these heuristics offline. In order to support the relevant question retrieval phase and the solution generation phase, we prepare the following materials:

Crash Trace Index: We use the NLTK package [26] to tokenize all the crash traces extracted from SO questions. All the pairs of question id and crash trace are collected to build the document corpus. The preprocessed document corpus is then utilized to build an index using Whoosh [27], similar to Lucene [28] but implemented in Python. CRASOLVER utilizes this index to initially filter out the relevant question set.

Exception Dictionary: To determine the concept of what a crash trace is about, we introduce knowledge from official documentation into our approach. We build a dictionary that stores the names of all exceptions and corresponding descriptions, crawled from official documentation sites.

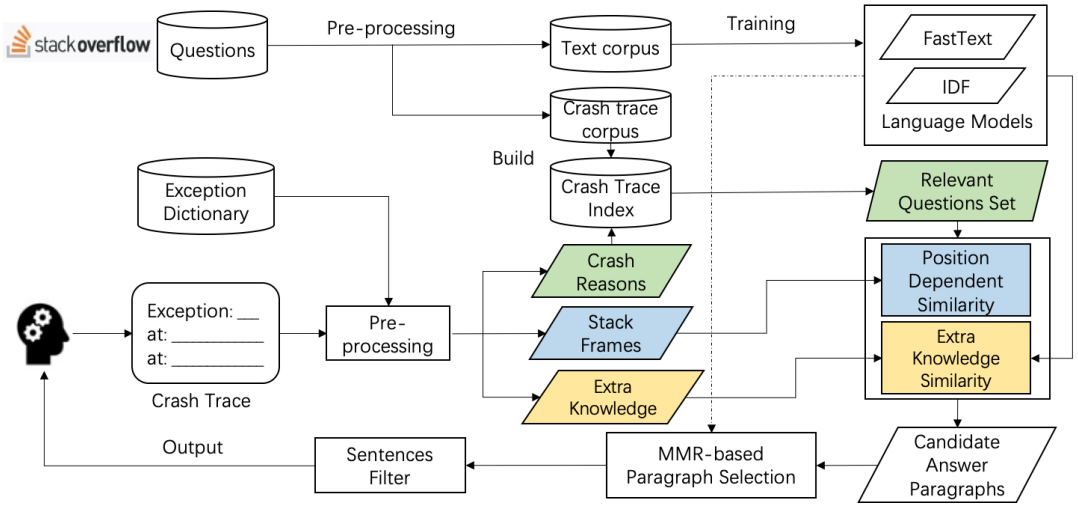

Fig. 2. The overall framework of our approach.

Language Models: To measure the similarity between a SO post and an exception description from official documentation, we build domain-specific language models. We collect all titles and body text of Stack Overflow posts and tokenize the extracted contents with white spaces and punctuation. We remove stop words and stem each word to its root form in the corpus using the NLTK package [26]. We then train a word embedding model FastText [29] to represent each word in the corpus as a fixed-length vector. We also compute the IDF metric (inverse document frequency) of each word in the vocabulary. The IDF metric represents the possibility that a word may carry important semantic information. We use the IDF-weighted word embedding vector as the representation of each word in the corpus. In this way, the semantic similarity is able to be computed in the question retrieval phase and the solution generation phase.

B. Relevant Question Retrieval

Given a crash trace CT, CRASOLVER first uses regular expressions to extract the *crash reasons* and *stack frames* respectively. CRASOLVER also checks every token contained in the trace *crash reasons*. If the token fully matches any names of exceptions in the Exception Dictionary (see Section II-A), the corresponding description and the exception name are added as extra knowledge for the crash trace.

In order to greatly narrow down the search space, CRASOLVER uses the Crash Trace Index to retrieve a relevant question set. CRASOLVER tokenizes the extracted *crash reasons* by NLTK [26] and loads the Crash Trace Index. CRASOLVER uses BM25 [20] to measure the lexical similarity of each crash trace in the index. The *crash reason* and the crash trace from the pre-loaded index are denoted as CR and P respectively. The similarity score is computed as follows:

$$Sim(P, CR) = \sum_{i=1}^{n} score_i \tag{1}$$

$$score_i = IDF(w_i) \times \frac{f(w_i, P) \times (k+1)}{f(w_i, P) + k \times (1 - b + b \times \frac{|P|}{avgdl})} \tag{2}$$

where $f(w_i, P)$ is the word w_i's term frequency in crash trace P, $|P|$ is the length of the crash trace, $IDF(w_i)$ is the inverse document frequency of word w_i, k and b are two free parameters. k is used to normalize the range of term frequencies, and b controls the influence of document length. We set k and b to be 1.2 and 0.75 by default. Prior works have demonstrated that this setting performs well for various corpora [30] [31]. In this paper, CRASOLVER only retrieves the top-50 similar questions to avoid introducing too much noise in later stages. CRASOLVER further filters out questions where the crash trace in question body shares no exception name with the query crash trace.

After getting the relevant question set based on the *crash reasons*, CRASOLVER further utilizes structural information contained in *stack frames* and extra knowledge from official documentation for a more precise result:

1) Position Dependent Similarity: We introduce our proposed position dependent similarity in this section, based on insights from Dang et al. [13]:

- the frame that causes the bug most likely occur near the top of the a *call stack*; and
- the alignment offset between two matched functions in two similar *call stacks* is likely to be small.

For the given a crash trace, CRASOLVER extracts all of its *call stacks* and corresponding *stack frames*. We denote the *call stacks'* position in the entire crash trace as s, and the distance of every function (token) to the top of its belonging *call stack* as d. CRASOLVER thus records the position information of each token w_i in the crash trace as a set:

$$position_{w_i} = \{(s_{i,1}, d_{i,1}), \cdots, (s_{i,k}, d_{i,k})\} \tag{3}$$

where k is the total number that w_i appears in the crash trace. For example, the position information of the token marked in blue in Figure 3 can be represented as $position_{w_{blue}} = \{(0, 2), (1, 4)\}$.

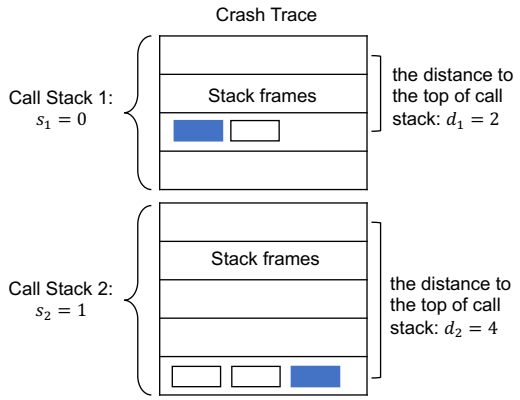

Fig. 3. Illustration of the Position information.

Let W be the set of all the matched functions between the given crash trace CT_1 and a crash trace from the relevant question set CT_2. The position dependent weight of each word in W is computed as:

$$PDweight_{w_i} = 1/(e^{min(dis_1, dis_2)} \times (|dis_1 - dis_2| + 1.0)) \tag{4}$$

where dis is the average of d in the $position_{w_i}$ from their respective crash trace. Based on BM25 [20], CRASOLVER calculates the position dependent similarity between CT_1 and CT_2 as follows:

$$PDSim(CT_1, CT_2) = \sum_{i=1}^{n} score_i \times PDweight_{w_i} \tag{5}$$

where the $score_i$ is calculated by Equation 2. Even if two crash traces have many identical tokens, if the position distributions of these tokens are very different (meaning the two bugs are not similar), our algorithm can distinguish them and give a low score. In this way, the position dependent similarities between the given crash trace and each crash trace in the relevant question set can be calculated.

2) Extra Knowledge Similarity: To understand what the crash trace about, we introduce extra knowledge sourced from official documentation. Given a crash trace and the relevant question set, CRASOLVER transforms the corresponding description from official documentation of the crash trace and the title of a relevant question into two bag of words, denoted as D and T, respectively. Unlike Yang et al. [22], who average word embedding vectors of words in a document, CRASOLVER uses a IDF-weighted word embedding vector to represent a document.

CRASOLVER loads the prepared language model mentioned in Section II-A and computes the extra knowledge similarity as follows:

$$EKSim(D, T) = \frac{\overrightarrow{v}_D \cdot \overrightarrow{v}_T}{\|\overrightarrow{v}_D\| \|\overrightarrow{v}_T\|} \tag{6}$$

$$\overrightarrow{v} = \frac{\sum_{i=1}^{n} IDF(w_i) \times FastText(w_i)}{\sum_{i=1}^{n} IDF(w_i)} \tag{7}$$

where w_i is the word in the document, $FastText(w_i)$ is the word embedding vector of the w_i represented by pre-trained FastText model (see Section II-A).

Now we obtain extra knowledge similarities and position dependent similarities between the given crash trace and all questions in the relevant question set. We normalize both similarity scores to make them comparable. The final relevant score between the given crash trace (CT) and each question (Q) in the relevant question set is calculated as:

$$Rel(Q, CT) = (1 - \mu)PDSim + \mu EKSim \tag{8}$$

where μ is a hyper-parameter set to 0.25 by default. We give more weight to position dependent similarities because we need knowledge of official documentation to help retrieve, rather than lead, the search for relevant questions.

Based on the calculated final relevant scores, CRASOLVER returns the top-10 relevant questions along with their answers for the given crash trace.

C. Solution Generation

In order to avoid some redundant information and generate a comprehensive solution, we borrow an idea from Maximal Marginal Relevance algorithm [24] to generate the solution. Given the answers of the top-10 relevant questions, CRASOLVER uses the granularity of answer paragraphs to generate the final solution. We first score each answer paragraph, denoted as A, based on four features:

- the relevant score of the question to which the paragraph belongs, which is described in Section II-B;
- semantic similarity score with crash trace's description, calculated by Equation 6;
- vote on answer – we set the vote on the answer post where the paragraph comes from as the vote score; and
- whether the answer is accepted. If the answer to which the paragraph belongs is accepted, we set the score to 2, otherwise 1.

The final score for a answer paragraph $Score(A, CT)$ is computed by multiplying the above four scores and normalizing the result. We then apply the MMR algorithm to select a set of paragraphs as:

$$MMR \stackrel{\text{def}}{=} Arg \max_{A_i \in R \setminus S} [\lambda Score(A_i, CT) - (1 - \lambda) \max_{A_j \in S} Sim(A_i, A_j)] \tag{9}$$

where CT is the given crash trace, S is the set of paragraphs which have been selected, $R \setminus S$ is the paragraphs which have not been selected, Sim is the cosine similarity between the vector representations $\overrightarrow{v}$ of two answer paragraphs, and λ is a parameter to adjust the relevance and diversity of the results, which is set at 0.75 by default.

In this work, CRASOLVER selects 3 relevant paragraphs to prevent too long output. The short code fragments which are enclosed in HTML tag $\langle code \rangle$ in natural language paragraphs are preserved in the paragraphs. The long code snippets with no more than 20 lines are extracted along with the answer paragraph. CRASOLVER will append the corresponding long code snippets, if any, to the end of the solution.

Finally we follow the method from Wang et al. [32] that uses grammatical dependency analysis by Stanford CoreNLP [33] to identify the higher quality sentences from these answer paragraphs.

$$VP << (NP < /NN.?/) < /VB.?/$$
$$NP! < PRP\,[<< VP|\$VP] \quad (10)$$

We keep only sentences that have a verb phrase followed by a noun, and have a noun phrase followed by a verb phrase [32]. The other sentences are filtered out and then CRASOLVER outputs the final solution to the users.

III. EVALUATION

In this section, we describe the experimental setup that we follow to evaluate the performance of our CRASOLVER. Our experiments aim to answer the following research questions:

RQ1: How does CRASOLVER perform compared to other approaches?

RQ2: What impact does each of the proposed metrics have in the CRASOLVER approach?

RQ3: How effective is the CRASOLVER relevant question retrieval component for crash traces?

A. Data Collection and Tool Implementation

We downloaded the official data dump of Stack Overflow [34] published on March 2020. Considering that our approach is designed for crash traces based on the Java language, we extracted the questions on SO that are tagged with "java" or "android". To create our knowledge base of exception-related questions, we selected questions satisfying the following criteria: (i) there is at least one answer to the question; (ii) there is a crash trace in the question body. Note that questions where there is a crash trace in the body were automatically identified by the heuristics described in Section II-A. We collected 69,478 Java questions and 33,566 Android questions respectively.

We downloaded the Java SE 8 API documentation [35] to introduce additional crash solving knowledge. We parsed the html files using Beautiful Soup [25] and extracted all the class names along with their descriptions. We collected all classes and did not distinguish whether they are exceptions for the sake of simplicity. In total, we built a dictionary containing 4,216 class name-description pairs.

Based on these repositories, we built a text corpus using the title and body of each post. We used Gensim [36] to train the FastText model.

In order to evaluate CRASOLVER, we randomly selected 50 Java questions and 50 Android questions where there are crash traces attached and made sure there are no duplicate questions. We extracted all crash traces in these questions to build the experimental queries. Note that these 100 questions and their duplicate questions are removed from our exception-related question base. We refer the 50 crash traces extracted from Java questions as *java test crashes* and the 50 crash traces from Android questions as *android test crashes*.

TABLE I
TASK ALLOCATION

RQs	Java 1-25	Java 26-50	Android 1-25	Android 26-50
RQ1	P1,P2,P3	P4,P5,P6	P7,P8,P9	P10,P11,P12
RQ2	P1,P2,P3	P4,P5,P6	P7,P8,P9	P10,P11,P12
RQ3	P13,P14,P15	P13,P14,P15	P16,P17,P18	P16,P17,P18

B. Baselines

Since there is no previous work to generate solutions for crash traces that we could find, we choose two approaches for automatic solution generation for programming questions and two solution search methods commonly used by developers.

AnswerBot [4]: AnswerBot is a three-stage framework to achieve the goal of generating an answer summary for a non-factoid technical question. AnswerBot evaluation has shown that its generated answer summaries are relevant, useful and diverse. In our experiment, we use the crash trace as the input.

CROKAGE [5]: CROKAGE is a tool that takes the description of a programming task as a query and provides a comprehensive solution for the task. We conduct our experiment by using their online website [37] and take the *crash reasons* part of crash trace as input due to the limitation of input length.

Google Search Engine: Google search engine is the most common way for developers to search for solutions. However, the length of input to google search engine is limited to 32 words. We take the *crash reasons* part of test crash trace as input in our experiment, which is also the general practice of most developers. For the sake of fairness, we add "site:stackoverflow.com" to the input of Google search engine so that it searches only posts on Stack Overflow.

Stack Overflow Search Engine: Stack Overflow is an important body of knowledge for solving developers' technical questions. Developers formulate their question as a query to Stack Overflow, and the search engine will return a list of relevant questions. We also use the *crash reasons* as query due to its limitation of query input.

For the baselines based on a search engine, we use the answer that is accepted or has the highest votes if there is no accepted answer from the first ranked Stack Overflow question returned by a search engine. Due to the different implementation of their search engines, even if we input the same thing into Google and Stack Overflow, the order of SO pages returned by both search engines is often different.

C. Participant Selection and Task Allocation

We invited 18 students who major in Computer Science to participate in our study, including 14 Ph.D. students and 4 graduate students. As shown in Table I, they will be assigned to the user studies for RQ1, RQ2 and RQ3, respectively. All of them are not co-authors and are indexed from P1 to P18. All study participants have industrial experience in Java programming ranging from 4 to 8 years, and participants evaluating the Android crash traces have at least 3 years' Android development experience. We have a tutorial for participants and they are allowed to conduct the study in their own lab.

TABLE II
COMPARISON RESULTS WITH BASELINES ON DIFFERENT LANGUAGES

Languages	Approaches	Relevance	Usefulness	Diversity
Java	CraSolver	**3.433**	**2.900**	**2.247**
	Google	2.740***	2.307***	1.540***
	Stack Overflow	2.500***	2.093***	1.333***
	AnswerBot	2.160***	1.70***	1.320***
	CROKAGE	2.410***	1.893***	1.353***
Android	CraSolver	**3.540**	**3.047**	**2.640**
	Google	2.833***	2.313***	1.560***
	Stack Overflow	2.313***	2.020***	1.480***
	AnswerBot	2.253***	1.753***	1.520***
	CROKAGE	2.317***	1.847***	1.373***

***p-value<0.001

Table I presents the task allocation. The 100 crash traces (50 from Java questions and 50 from Android questions) are divided into four groups, e.g., Java 1-25, Java 26-50, Android 1-25 and Android 26-50, to reduce overall task size for participants. RQ1 and RQ2 both evaluate the overall performance of CRASOLVER, and RQ3 evaluates the effectiveness of the retrieval component. To avoid biasing results, we have participants analyzing the same crash traces in RQ1 and RQ2, and arranged another group of participants to conduct RQ3 (e.g. P1, P2 and P3 evaluate Java 1-25 in RQ1 and RQ2 but Java 1-25 are evaluated by P13, P14 and P15 in RQ3).

D. Experimental Results

RQ1: How does CRASOLVER perform compared to other approaches?

Motivation. CRASOLVER aims to automatically provide a solution summarization for a given crash trace. We choose two approaches for automatic solution generation for programming questions, and two solution search methods commonly used by developers as baselines. Compared to these baselines, we wanted to evaluate the overall performance of our CRASOLVER approach in terms of the relevance, usefulness and diversity.

Approach. We compare CRASOLVER against four baselines described in Section III-B. In this user study, participants will first read the given crash trace, followed by five solutions from CRASOLVER and the four baseline approaches. Note that participants do not know which approach each solution comes from and the order is randomized.

Similar with Xu et al. [4], participants are asked to score the five solutions from three aspects, i.e., relevance, usefulness, and diversity. *Relevance* refers to how relevant the solution is to the crash trace. *Usefulness* refers to how useful the solution is for solving the crash bug. If a solution contains a variety of methods to solve the crash bug, as long as one of them can solve the problem well, we give it a high usefulness score. *Diversity* refers to whether the generated solution contains multiple relevant methods for the crash trace. Note that if some contents are not related to the crash, the diversity score cannot be improved because of this irrelevant content in the generated solution. The score ranges from 1 to 5. Score 1 means "irrelevant/useless/identical" and 5 means "highly relevant/useful/diverse".

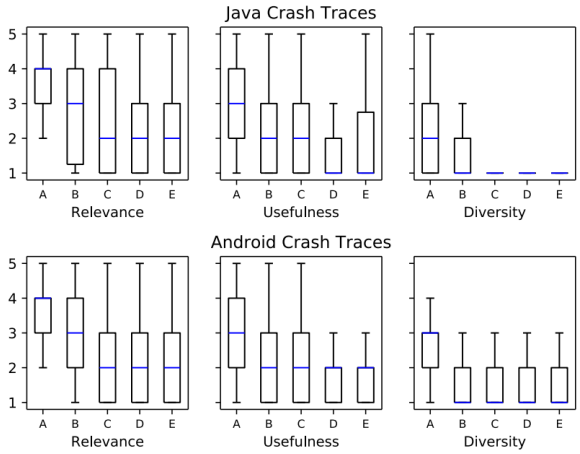

Fig. 4. Box plots of the relevance, usefulness and diversity of different approaches, e.g., (A) CraSolver, (B) Google, (C) Stack Overflow, (D) AnswerBot and (E) CROKAGE.

Results. Table II presents the mean of relevance, usefulness, and diversity scores and Figure 4 presents in box plots these scores of each solution from different approaches. The baseline built on Google is the best in all three aspects, besides our CRASOLVER method. Although Xu et al. [4] have reported that AnswerBot diversity is much higher than that of Google, it has not performed well because their approach is not aimed at crash traces.

However, **our CRASOLVER approach performs far better than the best baseline built on Google**. This phenomenon demonstrates that directly using the *crash reason* of a crash trace as input to Google has a limited effectiveness on identifying solutions to the crash bug. For the *java test crashes*, the relative improvements of CRASOLVER are 25.3%, 25.7% and 45.9%, w.r.t., relevance, usefulness, and diversity, respectively. For the *android test crashes*, CRASOLVER outperforms Google solution search by 25.0%, 31.7% and 69.2% in terms of relevance, usefulness and diversity, respectively. On average, CRASOLVER outperforms the baseline built on Google by 25.4%, 28.7% and 57.6% in terms of relevance, usefulness and diversity, respectively.

We conducted a Wilcoxon signed-rank test [38] with a Bonferroni correction [39] to evaluate whether the differences between CRASOLVER and these baselines are statistically significant. The improvements of our CRASOLVER approach over these baselines are statistically significant on three aspects at the confidence level of 99.9%. This suggests that the solution for crash trace solution generation by our CRASOLVER approach is better than other baselines in terms of relevance, usefulness, and especially diversity.

In summary, CRASOLVER **significantly outperforms other approaches for both Java and Android crash traces.**

RQ2: What impact does each of the proposed metrics have in our CRASOLVER approach?

TABLE III
RESULTS OF THE ABLATION STUDY

Languages	Approaches	Relevance	Usefulness	Diversity
Java	CraSolver^{-EKSim}	3.007***	2.507***	2.007**
	CraSolver^{-PDSim}	2.753***	2.147***	1.833***
	CraSolver	**3.433**	**2.900**	**2.247**
Android	CraSolver^{-EKSim}	3.027***	2.547***	2.200**
	CraSolver^{-PDSim}	2.780***	2.320***	1.867***
	CraSolver	**3.540**	**3.047**	**2.640**

***p-value<0.001, **p-value<0.01

Motivation. In order to improve the quality of the final generated solutions, we introduce position dependent similarity and extra knowledge similarity to the relevant question retrieval component, described in Section II-B. However, CRASOLVER can still work with only relying on one similarity metric. Thus, we need to conduct an ablation study to investigate the influence of these two similarity metrics.

Approach. To answer this research question, we analyze the performance gain achieved due to various components of our approach by performing an ablation study. We first remove the extra knowledge from official documentation and set all the extra knowledge similarity scores to 0 (i.e., $EKSim$) in CRASOLVER. We refer to this reduced approach as CraSolver^{-EKSim}. The second variant, CraSolver^{-PDSim}, disables the position dependent similarity, setting all the $PDweight$ to be 1. Finally we compare the performance of solutions generated by the above two methods with CRASOLVER using both similarity metrics. Participants are asked to score each solution from these approaches as in RQ1. The tasks are assigned according to Table I.

Results. Table III presents the results of our ablation study. We observe that CRASOLVER **performs significantly better than using each similarity metric individually**, on both *java test crashes* and *android test crashes*. On average, CRASOLVER outperforms CraSolver^{-PDSim} by 26.0%, 33.2% and 32.0% in terms of relevance, usefulness and diversity; CRASOLVER outperforms CraSolver^{-EKSim} by 15.6%, 17.7% and 16.0% respectively. Comparing the improvement ratio over CraSolver^{-PDSim} and CraSolver^{-EKSim}, we can see that the proposed position dependent similarity greatly improves the performance of our approach. The introduction of extra knowledge further improves the effectiveness of CRASOLVER.

As for RQ1, we conduct a Wilcoxon signed-rank test [38] with a Bonferroni correction [39], which has been marked in the table. The improvement of our approach over the variant CraSolver^{-PDSim} on all three aspects is statistically significant at the confidence level of 99.9%. Compared with CraSolver^{-EKSim}, the improvement of CRASOLVER on diversity is statistically significant at the confidence level of 99%, the improvement on relevance and usefulness is statistically significant at the confidence level of 99.9%. In summary, **the combination of the two similarity metrics makes our** CRASOLVER **approach achieve its best performance.**

RQ3: How effective is CRASOLVER's relevant question retrieval component for crash traces?

Motivation. In order to generate high-quality solutions, CRASOLVER must first narrow down the search scope to find the Stack Overflow questions related to the given crash trace. Whether the retrieval questions are relevant to the given crash trace will greatly affect the quality of the final solution. Here we investigate the effectiveness of our relevant question retrieval component (see Section II-B) in CRASOLVER, compared with several traditional information retrieval methods.

Approach. We choose four most commonly used information retrieval methods as our baselines: BM25 [20], TF-IDF based information retrieval [21], word embedding based information retrieval [22], and document-to-vector based information retrieval [23]. BM25 [20] is a ranking function used by search engines to estimate the relevance of documents to a given search query. TF-IDF [21] is also a traditional IR metric which has been widely used in software engineering [40]–[42]. Yang et al. [22] use the mean of word embedding vectors to represent the document and calculate the similarity. an unsupervised framework that learns continuous distributed vector representations for pieces of texts Document-to-vector [23] is an unsupervised framework that learns continuous distributed vector representations for pieces of texts.

For each test crash trace, we collect the top-10 relevant questions returned by our relevant question retrieval component or one of the baselines. We put all the top-10 results together from 5 approaches and there are many overlapping questions. Then, according to the task allocation (Table I), we ask 6 participants (P13, P14 and P15 for *java test crashes*, P16, P17 and P18 for *android test crashes*) to identify the relevant questions within the results. It is easy for participants because they only need to give yes/no response. Each participant had 5 days to complete the task, and they spent an average of 3.5 hours on it. We further use Fleiss Kappa [43] to measure the agreement between the three participants. The kappa values are 0.86 and 0.83 for *java test crashes* and *android test crashes* respectively, which indicate that participants have a high degree of consistency. If there are inconsistent labels among the participants, we take the label by majority rule as the final result.

The evaluation metrics we used to evaluate our retrieval component and other baselines are Top-K Accuracy (Top@K), Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP), which are widely used in previous software engineering studies [4], [40], [44]–[46]. Top-K Accuracy is the percentage of test crash traces where there is at least one relevant question within the top-K returned questions. In this paper, we set the K to be 1, 5 and 10. Mean Reciprocal Rank is the multiplicative inverse of the rank of the first relevant questions. Mean Average Precision is the average of precision for each test crash trace, where the precision is the percentage of relevant questions within the top-10 returned results.

Results. Table IV presents the performance of different approaches in terms of Top@K Accuracy, MRR and MAP. The

TABLE IV
PERFOMANCE OF CRASOLVER AND OTHER BASELINES IN TERMS OF TOP@K ACCURACY, MEAN RECIPROCAL RANK (MRR) AND MEAN AVERAGE PRECISION (MAP)

Languages	Approaches	Top@1	Top@5	Top@10	MRR	MAP
Java	TF-IDF	0.68	0.84	0.90	0.76	0.58
	BM25	0.78	0.86	0.94	0.82	0.68
	Word Embedding	0.70	0.92	0.94	0.80	0.60
	Doc2Vec	0.34	0.60	0.70	0.46	0.24
	CraSolver	**0.94**	**1.00**	**1.00**	**0.96**	**0.86**
Android	TF-IDF	0.62	0.88	0.92	0.73	0.56
	BM25	0.74	0.94	0.94	0.81	0.73
	Word Embedding	0.72	0.80	0.90	0.76	0.52
	Doc2Vec	0.50	0.66	0.80	0.58	0.28
	CraSolver	**0.92**	**0.96**	**0.98**	**0.94**	**0.83**

TABLE V
SOME EXAMPLES OF GENERATED SOLUTIONS

Crash Trace	Generated solutions
Process: social.com.networking.social.media.app, PID: 28258 java.lang.NoClassDefFoundError: Failed resolution of: Lcom/squareup/okhttp/OkHttpClient; at com.squareup.picasso.OkHttpDownloader.defaultOkHttpClient(OkHttpDownloader.java:31) at com.squareup.picasso.OkHttpDownloader.<init>(OkHttpDownloader.java:76) at com.squareup.picasso.OkHttpDownloader.<init>(OkHttpDownloader.java:65) at social.com.networkingsocialmediaapp.Global.onCreate(Global.java:17)... Caused by: java.lang.ClassNotFoundException: Didn't find class "com.squareup.okhttp.OkHttpClient" on path: DexPathList [[zip file "/data/app/social.com.networking.social.media.app-2/base.apk"], nativeLibraryDirectories=[/data/app/social.com.networking.social. media.app-2/lib/arm64, /vendor/lib64, /system/lib64]] at dalvik.system.BaseDexClassLoader.findClass(BaseDexClassLoader.java:56) at java.lang.ClassLoader.loadClass(ClassLoader.java:511) at java.lang.ClassLoader.loadClass(ClassLoader.java:469) at com.squareup.picasso.OkHttpDownloader.defaultOkHttpClient(OkHttpDownloader.java:31) at com.squareup.picasso.OkHttpDownloader.<init>(OkHttpDownloader.java:76)...	**Crasolver:** Well This is not an appropriate solution but it seems that there is some bug in okhttp library.. To solve this error i just replaced my gradle with the older version of okhttp. compile 'com.squareup.okhttp:okhttp:2.5.0'. compile 'com.squareup.okhttp3:okhttp:3.6.0'. And it works like a charm.. Thankyou. Those are the steps I followed to do the signup / login / linking of users with ParseUsers according to Parse documentation and some tips I found on the web:. First, I went to this website and created a new application.. I downloaded the Facebook SDK, I did the steps 4 and 5 and I added my package name and the default class name. I added the key hashes:... **Baseline Google:** Got the solution ,it is happening because of instant run, Just disable it Goto : Android Studio –>File –>Setting –>Build, execution, deploy –>Instant run.
Exception in thread "main" java.io.IOException: Not in GZIP format at java.util.zip.GZIPInputStream.readHeader(GZIPInputStream.java:141) at java.util.zip.GZIPInputStream.<init>(GZIPInputStream.java:56) at java.util.zip.GZIPInputStream.<init>(GZIPInputStream.java:65) at postgresjdbc.PostgresJDBC.gunzip(PostgresJDBC.java:237) at postgresjdbc.PostgresJDBC.readBytes(PostgresJDBC.java:230) at postgresjdbc.PostgresJDBC.main(PostgresJDBC.java:208) Java Result: 1	**Crasolver:** Since fcontents contains your gzipped data it should be a byte[] and not a String?. I recommend using IOUtils for reading the file into a byte array as reading it into a string will most likely corrupt your data... **Baseline Google:** String outStr = out.toString("UTF-8"); This "out" is ziped byte stream,encode it to String then decode it from String will be lose some bytes.This maybe a bug of java. To Resolve it,you can encode bytes to String in compress() to return, such as : String infoBase64Encode = new String(Base64.encodeBase64(out.toByteArray()))...
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space at java.util.HashMap.createEntry(HashMap.java:869) at java.util.HashMap.addEntry(HashMap.java:856) at java.util.HashMap.put(HashMap.java:484) at com.Kaggle.baseline.BaselineNew.createSimMap(BaselineNew.java:70) at com.Kaggle.baseline.BaselineNew.<init>(BaselineNew.java:25) at com.Kaggle.baseline.BaselineNew.main(BaselineNew.java:315)	**Crasolver:** The problem is that Eclipse Memory Analyser does not have enough heap space to open the Heap dump file.You can solve the problem as follows:open the MemoryAnalyzer.ini file. change the default -Xmx1024m to a larger size Note that on OS X, to increase the memory allocated to MAT, you need to right-click Memory Analyzer.app and show the package contents. The MemoryAnalyzer.ini file is under /Contents/MacOS/. I was facing the same issue but with the eclipse plugin and I did not have any Memory Analyzer App in Applications Folder. The solution which worked for me was:. Right Click on Eclipse icon and select Show Package Content. Go to Contents>Eclipse. Open Eclipse.ini. Restart Eclipse. **Baseline Google:** Well, it's fairly self-explanatory: you've run out of memory. You may want to try starting it with more memory, using the -Xmx flag, e.g., java -Xmx2048m [whatever you'd have written before] This will use up to 2 gigs of memory. See the non-standard options list for more details.

results show that **the relevant question retrieval component of** CRASOLVER **significantly outperforms other baselines in terms of Top@1, MRR and MAP for both test datasets.** Compared to the best baseline approach BM25, our approach outperforms BM25 by 21%, 17% and 26% in terms of Top@1, MRR and MAP on *java test crashes*. As for the *android test crashes*, the relative improvements of CRASOLVER are 24%, 16% and 14% in terms of Top@1, MRR and MAP respectively.

Another phenomenon is **that almost all the metrics of our method are close to 1.0, indicating its powerful performance in finding relevant SO questions for crash traces.** Our approach decomposes a crash trace into *crash reasons* and *stack frames*, and introduces the position dependent similarity and extra knowledge similarity for retrieval component. These enable CRASOLVER to achieve the best performance. This also ensures that the resulting solution is mostly composed of answer paragraphs to the relevant questions. The results indicate the effectiveness of the relevant question retrieval component, and the superior performance of this component further improves the quality of our final generated solutions.

IV. DISCUSSION

In order to help developers resolve software crash bugs more efficiently, we propose CRASOLVER to automatically generate solutions for thrown crash traces. CRASOLVER decomposes

every crash trace into *crash reasons* and *stack frames*, and utilizes the information of these two parts for better retrieval of similar crash traces. In particular, we designed two similarity scoring metrics for crash trace, e.g., position dependent similarity and extra knowledge similarity. The experiments described in Section III have shown the effectiveness of our CRASOLVER approach. In this section, we qualitatively analyze the advantages and disadvantages of CRASOLVER.

Until now, there have been no approaches designed for automatic crash solution generation. According to the experimental results from Section III, the baseline built on Google search engine performs the best among these other four baselines. We manually compare solutions from the baseline built on Google search engine and the solutions generated by CRASOLVER.

Table V presents three examples. In the first example, we can see from the crash trace that the cause of this crash bug may be a problem related to the "OkHttp" package or project configuration. The solution generated by CRASOLVER first points out that there may be some bugs in the okhttp library and then gives a feasible solution *"replace my gradle with the older version of okhttp"*. However, the second paragraph of the solution describes the "Facebook SDK" settings which is not very relevant to this crash trace. We check the answer post corresponding to this paragraph and find that it comes from a question that throws the same *crash reason* as this crash trace report, but involves different packages. We introduce the MMR algorithm (see Section II-C) to generate diversified and comprehensive solution suggestions, which may also introduce some noise into the final output. This leads to CRASOLVER suggested solutions occasionally containing some irrelevant content. The solution from the baseline built on Google provides a common method to solve the problem of project package configuration. From this case, we can find that the solutions given by CRASOLVER are more targeted, while those from baseline built on Google are more general.

In some cases like the second crash trace in Table V, the proportion of keywords (GZIP) in the *crash reason* part is relatively low. If we use *crash reason* to search for relevant answers in search engines, as most developers do, the answers returned by search engines may contain a lot about "IOException". This will cost developers a lot of time to find the right solution. In contrast, CRASOLVER first utilizes the *crash reason* to narrow down the search space. Among the relevant found questions, CRASOLVER then introduces a position dependent similarity and an extra knowledge similarity to retrieve more relevant questions. The tokens (e.g., zip and GZIPInputStream) in the *stack frames* of the crash trace help CRASOLVER to better locate relevant questions. The experimental results of RQ3 in Section III-D also show the effectiveness of the relevant question retrieval component in our approach. We can see from the solution generated by CRASOLVER it first explains the cause of the crash bug, and then gives the targeted solution. This also shows the disadvantage of using only the *crash reason* part of crash trace to search. Our approach utilizes both the *crash reason* and *stack frames* contained in the crash trace, which ensures better quality of the final generated solutions.

The third case is an "OutOfMemoryError", which is a crash bug that we often encounter in our daily development. The solution from baseline built on Google suggests using the -Xmx flag to start the program with more memory. In contrast, CRASOLVER provides not only the reason for the crash bug, but also several alternative methods to solve it. In this example, CRASOLVER provides solutions for different situations with or without MAT (Memory Analyzer Tool). Even better, CRASOLVER specially suggests a way to solve this crash bug on the OS X system. The solutions returned by the baseline built on Google usually focus on a specific aspect of the solution to the crash bug. Our generated solutions are derived from answers posted in several relevant questions. Therefore, the solutions generated by CRASOLVER can provide solutions with more alternative methods to solve crash bugs and more useful information for developers.

In addition to the advantages and disadvantages of our approach discussed above, CRASOLVER is also limited to situations that developers can find relevant solutions for the thrown crash traces by referring to the relevant content on Q&A sites. If there is no relevant content on Q&A sites, CRASOLVER is not able to generate useful and comprehensive solutions to the given crash traces. In contrast, the Google search engine can search for relevant solutions from a much wider range of website resources, and its results returned may be more useful. However, many programs depend on a certain framework, thus many crash bugs are recurring. The scale of questions on Q&A sites is also growing with the development of community. Our CRASOLVER approach has the potential to generate solutions for more crash traces in the future.

Another limitation is that our implementation of CRASOLVER can only deal with crash traces based on Java language at present. At present, we collect Stack Overflow questions tagged with Java or Android to build our repositories, and the official documentation we use is Java SE 8 API documentation [35]. However, the crash logs in other languages, e.g., python and c#, also have a structure similar to the Java call stack, which makes it not difficult for our approach to support other language crash traces. Based on the idea proposed in this paper, we plan to design algorithms that can analyze crash traces based on other programing languages to further enhance the generality of CRASOLVER. In addition, we will collect more official documentations from various languages and frameworks to extend the extra knowledge of our approach in the future.

V. THREATS TO VALIDITY

Threats to internal validity are related to the implementation of different approaches and the design of our user study. We have double checked our code to make sure that the questions used in test datasets are excluded from the question base when conducting experiments. We directly use the published tool of AnswerBot [47] and the online tool website of CROKAGE [37]. As for the input to the search engines,

we choose the *crash reasons* as long as possible in crash traces as the input, which is also one of the most commonly ways for developers to search for solutions. For the user study, each participant's development experience and understanding of the given crash trace may affect the results. We mitigate this threat by asking three participants with at least 3 years of programming experience to score for each solution. In addition, we recruited participants who show interest in our research and give them enough time to do the user study to reduce any threats caused by a participant's impatience.

Threats to external validity relate to the generalizability of the our experimental results. There are crash traces from various programming languages on Stack Overflow and each program language has its own crash trace structure. At present, CRASOLVER only supports crash traces based on Java language. But it is not difficult to support other languages based on the approach we propose. For this reason, we only use the Java and Android questions in this work. As the experimental part needs a lot of human participation, we only collected 50 example Java and 50 example Android crash traces for evaluation. In future, we will collect crash traces containing more types of crash bugs to mitigate these threats.

Threats to construct validity relate to suitability of our evaluation metrics. We use relevance, usefulness and diversity to evaluate the quality of generated solutions. These metrics are widely used to evaluate summarization tasks in software engineering [4], [48]–[51]. To investigate the effectiveness of retrieval component in our approach, we use Top@k accuracy, MRR and MAP, which are classical evaluation metrics for information retrieval [4], [40], [44]–[46].

VI. RELATED WORK

Text Summarization. Although there is to our knowledge no previous work that automatically generates solutions for given crash traces, this task is similar to other summarization tasks in software engineering, e.g., bug reports summarization [49], [52]–[54] and comment summarization [32] [50] [55]). Rastkar et al. [52] [53] used a bug report corpus to train a classifier and is based on conversational data for automatic bug report summarization. Mani et al. [49] proposed an unsupervised approach based on noise reducer to improve the precision of bug report summarization. Lotufo et al. [54] pose three hypotheses on what makes a sentence relevant and used heuristic rules to perform bug report summarization.

For comment summarization, Wong et al. [32] use natural language processing to filter relevant sentences to compose the descriptions for a code. Our approach borrows two patterns they used to identify important sentences. SURF [50] classifies each user review sentence to one of the user-intention categories and generates user review summary based on a scoring mechanism. Gias et al. [55] proposed two algorithms (statistical and aspect-based) to summarize opinions about APIs. Different from the above studies, our work aims at providing useful solution generation for a given crash trace.

Mining Developer Forums. There are abundant resources for researchers to explore in developer Q&A forums. Treude et al. [56] analyzed how programmers ask and answer questions on the web. They find that Q&A websites are effective for code reviews and conceptual questions. ACE [57] is a novel traceability recovery approach to extract the code elements contained in informal documentation like Stack Overflow. SeaHawk [58] and Prompter [59] retrieve API names and keywords from Stack Overflow by the formulated queries based on the code context. However, their approach is not applicable to the crash trace solution recommendation problem. Huang et al. [40] proposed an approach named BIKER to recommend APIs for the programming task via analyzing the posts on Stack Overflow. AnswerBot [4] bridges the lexical gap and provides relevant, useful and diverse answer to the natural language queries. CROKAGE [5] provides comprehensive solution containing not only relevant code examples but also their succinct explanations. Different from our approach, the inputs to both AnswerBot and CROKAGE are natural language queries, which are difficult for developers to use to describe a complex crash bug in natural language. In addition, a lot of information contained in the crash trace will be lost.

Analysing Crash Traces. The crash trace has been used in many software engineering studies, such as bug localization [6]–[11], automated program repair [1], [12], [60]–[63] and duplicate bug report detection [13]–[16].

CrashLocator [6] locates faulty functions by expanding function call sequences in a static call graph and then ranking suspicious functions, according to the suspiciousness of each function. Wang et al. [7] proposed three rules to identify correlated crash types and an algorithm to locate and rank buggy files using crash correlation groups. Moreno et al. [8] and Wong et al. [9] proposed to use code segments and stack traces to retrieve code elements relevant to bug reports. Mohammad et al. [10] transform stack traces into a trace graph and reformulate queries to improve IR-based bug localization. STMLocator+ [11] automatically locates the relevant buggy source files for a given bug report.

For automated program repair, Gao et al. [1] proposed an approach to fix recurring crash bugs by generating edit scripts for source code via analyzing Q&A sites. However, their approach only uses the *crash reason* of crash traces to retrieve relevant questions by a search engine. Droix [12] uses user event sequences (e.g., clicks and touches) as input to repair buggy apps and utilizes the crash traces to locate the bug. Some other existing techniques in automated program repair typically rely on unit tests [60] or test scripts [61]–[63] to guide repair process. Different from these techniques that aim at generating patches, our approach provides *explanations* for crash traces. Thus, they are complementary and can work hand-in-hand to help developers resolve crashes.

Kim et al. [14] proposed crash graphs which provide an aggregated view of multiple crashes in the same bucket to detect duplicate bug report. Dang et al. [13] proposed a method for clustering crash reports based on call stack matching. Similar to our position dependent similarity, they also considered the distance to the top frame and alignment offset. However, they

used frame as the matching granularity and their similarity metric is formulated in a different way. Johannes et al. [15] only used stack traces and their structure as input for detecting bug-report duplicates. DURFEX [16] is a feature extraction technique that extracts features from bug reports with a focus on crash traces for detection of duplicate bug reports. However, our approach not only analyses the structure of crash traces but also introduces extra knowledge for retrieving potential solution posts. Our experimental results in Section III-D have shown its effectiveness.

VII. Conclusion

In this paper, we propose CraSolver to automatically generate possible solution summaries from a given crash trace. CraSolver parses a crash trace into *crash reasons* and *stack frames*, and introduces position dependent similarity and extra knowledge similarity measures to retrieve relevant questions on Stack Overflow. After obtaining the answers to these questions, CraSolver summarizes a solution based on several features. We leverage both information contained in crash traces and also the knowledge in official documentation to improve the effectiveness of CraSolver. Our evaluation using Java and Android crash traces demonstrates the relevance, usefulness and diversity of our generated solutions. We believe the CraSolver approach will help developers solve crash bugs more efficiently and accurately in practice. The code for our approach and the data of experiments are available at http://tiny.cc/qzhqsz.

In the future, we plan to integrate CraSolver into an IDE and provide timely solutions when developers encounter crash bugs during development. Furthermore, we will extend CraSolver to support crash traces based on more programming languages and introduce more official documentation into its knowledge base.

Acknowledgment

This research was partially supported by the National Key R&D Program of China (No. 2019YFB1600700), Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE200100021), ARC Laureate Fellowship funding scheme (FL190100035), ARC Discovery grant (DP200100020), Key Research and Development Program of Zhejiang Province (No.2021C01014), and the National Research Foundation, Singapore under its Industry Alignment Fund – Prepositioning (IAF-PP) Funding Initiative. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei, "Fixing recurring crash bugs via analyzing q&a sites (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 307–318.

[2] S. Kim, K. Pan, and E. J. Whitehead Jr, "Memories of bug fixes," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 35–45.

[3] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 315–324.

[4] B. Xu, Z. Xing, X. Xia, and D. Lo, "Answerbot: Automated generation of answer summary to developers' technical questions," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 706–716.

[5] R. Silva, C. Roy, M. Rahman, K. Schneider, K. Paixao, and M. Maia, "Recommending comprehensive solutions for programming tasks by mining crowd knowledge," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 358–368.

[6] 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*, 2014, pp. 204–214.

[7] S. Wang, F. Khomh, and Y. Zou, "Improving bug localization using correlations in crash reports," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 247–256.

[8] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the use of stack traces to improve text retrieval-based bug localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 151–160.

[9] 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 *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 181–190.

[10] M. M. Rahman and C. K. Roy, "Improving ir-based bug localization with context-aware query reformulation," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 621–632.

[11] Y. Wang, Y. Yao, H. Tong, X. Huo, M. Li, F. Xu, and J. Lu, "Enhancing supervised bug localization with metadata and stack-trace," *Knowledge and Information Systems*, pp. 1–24, 2020.

[12] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 187–198.

[13] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 1084–1093.

[14] S. Kim, T. Zimmermann, and N. Nagappan, "Crash graphs: An aggregated view of multiple crashes to improve crash triage," in *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. IEEE, 2011, pp. 486–493.

[15] J. Lerch and M. Mezini, "Finding duplicates of your yet unwritten bug report," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 69–78.

[16] K. K. Sabor, A. Hamou-Lhadj, and A. Larsson, "Durfex: a feature extraction technique for efficient detection of duplicate bug reports," in *2017 IEEE international conference on software quality, reliability and security (QRS)*. IEEE, 2017, pp. 240–250.

[17] M. Asaduzzaman, A. S. Mashiyat, C. K. Roy, and K. A. Schneider, "Answering questions about unanswered questions of stack overflow," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 97–100.

[18] M. Iyyer, J. Boyd-Graber, L. Claudino, R. Socher, and H. Daumé III, "A neural network for factoid question answering over paragraphs," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 633–644.

[19] D. Chen, A. Fisch, J. Weston, and A. Bordes, "Reading wikipedia to answer open-domain questions," *arXiv preprint arXiv:1704.00051*, 2017.

[20] S. E. Robertson and S. Walker, "Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval," in *SIGIR'94*. Springer, 1994, pp. 232–241.

[21] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 842–851.

[22] X. Yang, D. Lo, X. Xia, L. Bao, and J. Sun, "Combining word embedding with information retrieval to recommend similar bug reports," in *2016 IEEE 27Th international symposium on software reliability engineering (ISSRE)*. IEEE, 2016, pp. 127–137.

[23] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014, pp. 1188–1196.

[24] J. Carbonell and J. Goldstein, "The use of mmr, diversity-based reranking for reordering documents and producing summaries," in *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, 1998, pp. 335–336.

[25] "Beautiful soup," https://www.crummy.com/software/BeautifulSoup/, 2020.

[26] E. Loper and S. Bird, "Nltk: the natural language toolkit," *arXiv preprint cs/0205028*, 2002.

[27] "Whoosh," https://whoosh.readthedocs.io/en/latest/, 2012.

[28] "Apache, lucene," http://lucene.apache.org/, 2020.

[29] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[30] Y. Lv and C. Zhai, "Adaptive term frequency normalization for bm25," in *Proceedings of the 20th ACM international conference on Information and knowledge management*, 2011, pp. 1985–1988.

[31] H. Fang, T. Tao, and C. Zhai, "A formal study of information retrieval heuristics," in *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, 2004, pp. 49–56.

[32] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 562–567.

[33] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, 2014, pp. 55–60.

[34] "Stack overflow data dump," https://archive.org/download/stackexchange, 2020.

[35] "Java se 8 api documentation," https://www.oracle.com/java/technologies/javase-jdk8-doc-downloads.html, 2020.

[36] R. Rehurek and P. Sojka, "Software framework for topic modelling with large corpora," in *In Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Citeseer, 2010.

[37] "Crokage tool," http://isel.ufu.br:9000/, 2019.

[38] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*. Springer, 1992, pp. 196–202.

[39] H. Abdi, "Bonferroni and šidák corrections for multiple comparisons," *Encyclopedia of measurement and statistics*, vol. 3, pp. 103–107, 2007.

[40] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "Api method recommendation without worrying about the task-api knowledge gap," in *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2018, pp. 293–304.

[41] D. Shepherd, K. Damevski, B. Ropski, and T. Fritz, "Sando: an extensible local code search framework," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–2.

[42] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 964–974.

[43] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[44] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 476–481.

[45] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 262–273.

[46] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong, "Mining likely analogical apis across third-party libraries via large-scale unsupervised api semantics embedding," *IEEE Transactions on Software Engineering*, 2019.

[47] L. Cai, H. Wang, B. Xu, Q. Huang, X. Xia, D. Lo, and Z. Xing, "Answerbot: an answer summary generation tool based on stack overflow," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1134–1138.

[48] D. Radev and W. Fan, "Automatic summarization of search engine hit lists," in *ACL-2000 Workshop on Recent Advances in Natural Language Processing and Information Retrieval*, 2000, pp. 99–109.

[49] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: approach for unsupervised bug report summarization," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[50] A. Di Sorbo, S. Panichella, C. V. Alexandru, J. Shimagaki, C. A. Visaggio, G. Canfora, and H. C. Gall, "What would users change in my app? summarizing app reviews for recommending software changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 499–510.

[51] R. Hao, Y. Feng, J. A. Jones, Y. Li, and Z. Chen, "Ctras: Crowdsourced test report aggregation and summarization," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 900–911.

[52] S. Rastkar, G. C. Murphy, and G. Murray, "Summarizing software artifacts: a case study of bug reports," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 1. IEEE, 2010, pp. 505–514.

[53] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 366–380, 2014.

[54] R. Lotufo, Z. Malik, and K. Czarnecki, "Modelling the 'hurried'bug report reading process to summarize bug reports," *Empirical Software Engineering*, vol. 20, no. 2, pp. 516–548, 2015.

[55] G. Uddin and F. Khomh, "Automatic summarization of api reviews," in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 159–170.

[56] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?(nier track)," in *Proceedings of the 33rd international conference on software engineering*, 2011, pp. 804–807.

[57] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 832–841.

[58] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Leveraging crowd knowledge for software comprehension and development," in *2013 17th European Conference on Software Maintenance and Reengineering*. IEEE, 2013, pp. 57–66.

[59] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining stackoverflow to turn the ide into a self-confident programming prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, 2014, pp. 102–111.

[60] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[61] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[62] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th international conference on software engineering*, 2016, pp. 691–701.

[63] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.