The following publication Yu, Le; Wang, Haoyu; Luo, Xiapu; Zhang, Tao; Liu, Kang; Chen, Jiachi; Zhou, Hao; Tang, Yutian; Xiao, Xusheng(2023). Towards Automatically Localizing Function Errors in Mobile Apps With User Reviews. IEEE Transactions on Software Engineering, 49(4), 1464-1486 is available at https://doi.org/10.1109/TSE.2022.3178096.

Towards Automatically Localizing Function Errors in Mobile Apps with User Reviews

Le Yu, Haoyu Wang, Xiapu Luo‡, Tao Zhang, Kang Liu,
Jiachi Chen, Hao Zhou, Yutian Tang, and Xusheng Xiao

Abstract—Removing all function errors is critical for making successful mobile apps. Since app testing may miss some function errors given limited time and resource, the user reviews of mobile apps are very important to developers for learning the uncaught errors. Unfortunately, manually handling each review is time-consuming and even error-prone. Existing studies on mobile apps' reviews could not help developers effectively locate the problematic code according to the reviews, because the majority of such research focus on review classification, requirements engineering, sentiment analysis, and summarization [1]. They do not localize the function errors described in user reviews in apps' code. Moreover, recent studies on mapping reviews to problematic source files look for the matching between the words in reviews and that in source code, bug reports, commit messages, and stack traces, thus may result in false positives and false negatives since they do not consider the semantic meaning and part of speech tag of each word. In this paper, we propose a novel approach to localize function errors in mobile apps by exploiting the context information in user reviews and correlating the reviews and bytecode through their semantic meanings. We realize our new approach as a tool named `ReviewSolver`, and carefully evaluate it with reviews of real apps. The experimental result shows that `ReviewSolver` has much better performance than the state-of-the-art tools (i.e., `ChangeAdvisor` and `Where2Change`).

Index Terms—Function error localization, User reviews, Mobile apps.

1 Introduction

With the rapid growth of mobile apps, removing function errors is critical for making successful mobile apps. Since app testing may not reveal all function errors given limited time and resource, the user reviews of mobile apps [2] are very important to developers for learning their apps' bugs [3], requested features [4], limitations [5], and strengths [6]. Unfortunately, manually handling each review is time-consuming and error-prone because apps may receive thousands of reviews, part of which may be useless and even incorrect. Moreover, if the person who processes the reviews is not familiar with the apps' code, it is difficult for him/her to determine whether a review is useful.

It is challenging to automatically map user reviews, especially function errors, to code, because reviews are written in natural languages by normal users and they are usually short and unstructured whereas the apps are developed in programming languages and compiled into bytecode or binary code, which are designed for the runtime instead of normal users. It is worth noting that the majority of existing studies on mobile apps' reviews just summarize and classify user reviews [7]–[12] without taking into account apps' code, and thus they cannot help developers locate the problematic code according to the reviews. To localize the function errors described in user reviews, a few studies extract words from user reviews and compare them with that in source code [13], bug reports [14], commit messages [15], stack traces [16]. However, none of them extracts the semantic information of function errors from bytecode and reviews, which may result in false positives and false negatives. For instance, one user review describes that the error has appeared "for the longest time". Although this error is not caused by the "time", these systems will still extract the word "time" and then localize the error in the *Clock* related class. In this case, one false positive is generated (See real examples in Section 2.3). Palomba et al. recently propose `ChangeAdvisor` [13] for mapping reviews to source code by first clustering similar reviews and then comparing the topic words identified from the clusters with the words extracted from the names of code components (e.g., methods, classes). Unfortunately, because `ChangeAdvisor` does not exploit the semantic information in reviews/bytecode and the names of code components contain limited information, its performance of localizing the function errors will be affected. Another system `Where2Change` [14] employs the bug reports to improve the performance of `ChangeAdvisor`. For each review cluster, `Where2Change` compares its topic words with the topic words of bug reports to determine if they are relevant or not. If true, `Where2Change` combines the topic words extracted from both the review cluster and the bug report to find the relevant source code. Different from `Where2Change`, the system `RISING` employs the commit messages and source code to find more mappings [15]. Apart from source code files, if the similarity between the review cluster and the commit message is high, `RISING` builds up mapping between the review cluster and the changed files of the commit message. Both `Where2Change` and `RISING` do not extract the semantic information of function errors from reviews and bytecode.

In this paper, to address this challenging problem, we propose a novel approach and develop a new tool named `ReviewSolver` to localize function errors in mobile apps by correlating their reviews and bytecode through their

‡*The corresponding author.*
Le Yu, Xiapu Luo, and Hao Zhou are with the Department of Computing in the Hong Kong Polytechnic University. Haoyu Wang is with the School of Cyber Science and Engineering in Huazhong University of Science and Technology. Tao Zhang is with the School of Computer Science and Engineering in Macau University of Science and Technology. Kang Liu is with the Institute of Automation in Chinese Academy of Sciences. Jiachi Chen is with the School of Software Engineering in Sun Yat-Sen University. Yutian Tang is with the School of Information Science and Technology in ShanghaiTech University. Xusheng Xiao is with the Department of Computer and Data Sciences in Case Western Reserve University.

semantic meanings with the hints of context information in user reviews. We aim at Android apps because Android has occupied 75% market share of mobile operating systems [17] and there are already 2.9 million apps in Google Play [18]. In particular, ReviewSolver exploits three new observations. First, as shown in Section 2.2, the user reviews related to function errors usually contain context information (e.g., API, GUI, etc.), which provides hints for inferring the source of errors. For example, one review of the app com.fsck.k9 is "*Reinstalled k9, reply button now doesn't show, can't find any solutions.*" This error is related to a button. To locate the corresponding code, we first analyze the structure of GUI and the components therein. After extracting the noun phrase "reply button" from the review, we search the word "reply" that modifies the "button" in the information related to each GUI component. Finally, we recommend the developer to check the activity *com.fsck.k9.activity.EditIdentity* since it contains a widget named "reply_to".

Second, due to the diverse expression and word ambiguity of user reviews, we need to conduct sentence-level analysis for squeezing useful information out of user reviews rather than relying on a few topic words from review clusters. The latter may miss much useful information. For example, a review of com.fsck.k9 is "*The latest upgrade just broke K9. Random certificate errors*". The user mentioned that this error was related to an certificate. When extracting topic words, ChangeAdvisor missed the word "certificate", and thus it cannot locate this error. After extracting the noun phrase "certificate error", we can locate APIs that contain "certificate" in their descriptions. Finally, we find the class *com.fsck.k9.view.ClientCertificateSpinner* since it calls the certificate related API *KeyChain.choosePrivateKeyAlias()*).

Third, the rich information distributed in various software artifacts related to apps should be leveraged to enhance the limited information in the names of code components. Moreover, instead of looking for exact words in user reviews and code, we should correlate them through their semantic meanings to avoid missing the mapping. For example, in the review "*When the picture is saved, it gets flipped upside my down*" of the app fr.xplod.focal, the "save picture" verb phrase can be mapped to the camera related APIs (e.g., *MediaRecorder.setVideoSource()*) since "picture" and "video" are similar nouns. Note that directly searching the phrase "save picture" in code files cannot find any related class.

New solutions. Therefore, to help developers automatically map function error reviews to code, ReviewSolver first identifies such kind of function error reviews through supervised machine learning algorithm and analyzes each sentence in such reviews to extract useful verb phrase and noun phrase through natural language processing (NLP) techniques (described in Section 3.2). Then, it conducts static bytecode analysis on apps to extract seven kinds of information (described in Section 3.3). Finally, ReviewSolver maps the reviews to the code according to their semantic similarity and recommends the most related code to developers (described in Section 4). We carefully evaluate the performance of ReviewSolver and compare it with ChangeAdvisor [13] and Where2Change [14], the state-of-the-art tool using real reviews of 18 open-source apps. It is worth noting that ReviewSolver handles apps' bytecode directly and we select open-source apps for the ease of evaluation and comparison, because ChangeAdvisor and Where2Change need apps' source code. The experimental results show that ReviewSolver can identify function error related reviews with at least 85.4% precision and 66.4% recall rate. For the same reviews that can be correlated to code files (by checking bug reports), ReviewSolver correctly locates 359 code files whereas ChangeAdvisor only correctly locates 102 code files and Where2Change only correctly locates 211 code files. For the same reviews that can be correlated to code files (by checking release notes), ReviewSolver correctly locates 84 code files whereas ChangeAdvisor only correctly locates 15 code files and Where2Change only correctly locates 25 code files. Moreover, ReviewSolver can map 57.9% of function error related reviews to code whereas ChangeAdvisor can only map 9.3% of such reviews and Where2Changecan only map 38.0% of such reviews.

New materials compared with the earlier version. Compared with our earlier version [19], this manuscript includes a significant amount of new materials. First, we enhance the capability of ReviewSolver. For the review analysis in Section 3.2, we propose to split each function error review into sentences and then perform sentiment analysis to identify the positive sentences. For the static analysis in Section 3.3, after downloading all versions of APK files, we enable ReviewSolver to generate summarization for each method in the APK file. It will also identify the abbreviations in the GUI and then replace them with the raw words. For the localizing function errors (in Section 4), we add four new methods to localize the function errors: 1) For the function error related to app specific task, we also leverage the generated summarization of each method to localize it (Section 4.1.1). 2) If the user implicitly describes the type of the issue, we search the type in GUI to localize it (Section 4.1.2). 3) For the function error related to the general task, we propose to localize it through using the Q&A of third party websites (Section 4.2.2). 4) For the function error review describing the type of exception, we localize it through checking the framework APIs and the methods defined by developers (Section 4.2.3). Second, we perform much more evaluations on ReviewSolver with a larger data set having 27000 user reviews and a new ground truth created by using the release notes (Section 5.1). Besides reconducting the experiments in earlier version [19], we add the following new evaluations, including 1) the performance of using machine learning algorithm to process the review dataset provided by Maalej et al. [20], [21], 2) comparing the performance of ReviewSolver with another state-of-the-art system Where2Change (Section 5.3), 3) measuring the percentage of function error reviews resolved by using the context information "General Task", "Exception", and the summarization of method (Section 5.4).

Main contributions. In summary, our major contributions include:

- By manually reading the function error reviews of mobile apps, we summarized the types of context information commonly used to describe the function errors. Based on this observation, we propose a novel approach to localize function errors in mobile apps by exploiting the context information in user reviews and correlating the reviews and bytecode through their semantic meanings.
- We realize the new approach in the tool ReviewSolver that leverages NLP and program

analysis techniques to automatically extract selected information from an app's APK file and its reviews, and then map the function error reviews to code.
- We evaluate ReviewSolver using real apps and their reviews, and compare it with ChangeAdvisor and Where2Change. The results show that ReviewSolver outperforms ChangeAdvisor and Where2Change in terms of correctly mapping more reviews to code.

The rest of this paper is organized as follows. Section 2 introduces the background and motivating examples. Section 3 and Section 4 detail the design of ReviewSolver. We present the experimental result in Section 5 and discuss the limitation of ReviewSolver in Section 6, respectively. After introducing the related work in Section 7, we conclude the paper in Section 8.

2 Background and Motivating Examples

2.1 Function Error Related Reviews

By manually analyzing 6,390 user reviews, Khalid et al. [22] summarized 12 types of user complaints in user reviews, and the top 3 most common complaints include function error (26.68%), feature request (15.13%), and app crashing (10.51%). Function error related reviews describe app specific problem found when using it. An example is "*Couldn't connect to server*". App crashing related reviews depict the event of app crashing. For instance, "*Crashes every time I use it*". Since both function errors and app crashing are critical problems, we consider them together under the same category (i.e., function errors) by mapping the user reviews to the corresponding code.

2.2 Context Information in Reviews

A key insight behind ReviewSolver is that users may describe the context under which an error occurred when writing reviews [23]. Such context information provides us useful hints to locate the problematic codes. To further illustrate it, we randomly select 250 function error reviews with at least 4 words from 18 open-source apps (Section 5.4, Table 11), manually read them, and summarize the context of the function errors. As shown in Table 1, 76.8% function error reviews contain more or less context information. We use examples to illustrate how to locate the problematic code by exploiting such context information in Section 2.3.

As shown in Table 1, most errors (30.4%) related to the functions specific to an app ("(1) App Specific Task" in Table 1). Since different apps have different specific functions, it is difficult to predefine some classes and group these functions into them. Hence, we look for the classes/methods that realize these functions. Since 8.8% errors appear after the app is updated, we will determine the code difference between the version reported by users and the previous version. Sometimes the users may describe the error message shown in app (10.8%), and hence we locate such error by checking the classes that display such error message. For function error reviews that report crashing right after the app is launched(3.2%), we will locate and check the starting activity. If the errors happen when registering account or during login (1.6%), we look for and examine the account registration and login related activities. 9.6% errors are related to the resource/information of the device.

TABLE 1
The context information in function error reviews.

Context	Description	Percentage	Example
(1) App Specific Task	Errors appear when performing app specific tasks	30.4% (76/250)	"Keeps crashing every time I open imgur links..."
(2) Updating App	Errors appear after updating the app	8.8% (22/250)	"App started crashing after recent update."
(3) GUI	Errors appear when interacting with GUI	6.0% (15/250)	"Note 4 does not have menu hard button."
(4) Error Message	Reviews contains the error messages from apps	10.8% (27/250)	"it just says "c:geo can't load data required to log visit""
(5) Opening App Activity	Errors appear when opening the app	3.2% (8/250)	"It crashed every time I opened it."
(6) Registering Account Interface	Errors appear when login-ing/registering account	1.6% (4/250)	"Cannot login to my gmail"
(7) API/URI/intent	Errors appear when accessing resource or information	9.6% (24/250)	"But I cannot save photos to sd card with it"
(8) General Task	General tasks implemented by many developers	5.6% (14/250)	"Too many errors that prevent file downloads from completing."
(9) Exception	An exception appeared when using the app	0.8% (2/250)	"You got a null pointer exception on the login screen"
(10) Other	User does not describe the context	23.2% (58/250)	"Sometimes not working."

Since such resource/information could be accessed by using APIs, URIs, or intents, we will locate the problematic code through the corresponding APIs, URIs, or intents. We also find 5.6% reviews describing the general tasks related to the bugs. Since these tasks have been implemented by many developers, we search the implementation of other developers by using Q&As of third party websites (e.g., Stack Overflow [24], CSDN [25]) and leverage them to find similar implementation in app code. Moreover, 0.8% reviews describe the exception message. We will search the classes throwing the corresponding exception.

We divide these types of context information in Table 1 into two categories and detail how to map them to code in Section 4.1 and 4.2 individually. One category includes app specific errors that are related to the functions implemented by developers (i.e., Table 1 case (1)-(6)). The other one includes the general errors commonly appeared in different apps (i.e., Table 1 case (7)-(9)).

Table 1 also shows that 23.2% function error reviews do not contain context information. They usually describe that the app does not work due to some bugs (e.g., "*Crash after crash. Uninstall very fast!*") or simply point out the device type (e.g., "*Please fix the bug. i'm using xiaomi mi4c*"). We discuss possible solutions to handle them in Section 6 and will investigate them in future work.

2.3 Motivating Examples

To clearly differentiate our approach (i.e., ReviewSolver) from the state-of-the-art method (i.e., ChangeAdvisor [26]), we use the motivating examples to demonstrate why ChangeAdvisor will lead to false positives and false negatives and how our approach can address the problems. Note that ChangeAdvisor [13] first clusters similar reviews and then looks up the topic words identified from the clusters in a set of words extracted from the names of source code

elements (e.g., fields, methods, and classes) to determine problematic source file.

Without considering the syntactic and semantic information in the sentence, `ChangeAdvisor` may include irrelevant words and cause *false positive* (i.e., the mapping from the review to the code is incorrect). **Example 1** illustrates this.

Example 1 `com.fsck.k9`: "*Unable to fetch mail on Samsung Note 4 for Nexus 7 for the longest time*".

`ChangeAdvisor` This review describes an error related to "fetch mail". `ChangeAdvisor` extracts the word "time" as topic words of the cluster and recommends the developer to check the class *com.fsck.k9.Clock* since the code file of class also contains the word "time". Unfortunately, this class is not related to this error.

`ReviewSolver` We first extract the verb phrase "fetch mail" from the syntactic tree of this review, and then compare the semantic similarity between this verb phrase and the verb phrases extracted from method names. If the similarity is higher than the threshold, `ReviewSolver` recommend the developer to check the corresponding method (i.e.,, "*com.fsck.k9.Account.getEmail()*" in this example).

Moreover, since `ChangeAdvisor` does not conduct static analysis on apps, it may lead to many false negatives (i.e., cannot map the errors to the code). By contrast, `ReviewSolver` can reveal them by leveraging the context information in user reviews and the information distributed in various software artifacts related to apps, as illustrated in the **Examples 2-5**.

Example 2 `org.thoughtcrime.securesms`: "*Unfortunately I can no longer send SMS to any non-signal user.*"

`ReviewSolver` Since some errors in reviews are related to Android framework APIs, we look for the classes that invoke the corresponding APIs. In particular, we extract the verb phrase "send SMS" from the review, and look for the APIs whose descriptions express the same meaning. Since the API *SmsManager.sendTextMessage()* fulfills the requirement, we recommend developer to check the class *org.thoughtcrime.securesms.jobs.SmsSendJob* since it calls this API.

Example 3 `org.thoughtcrime.securesms`: "*Signal crashed when i tried to find contact while writing sms ...*"

`ReviewSolver` Since some errors in reviews are related to the content providers, we locate the invocation of such content providers in apps. More precisely, after extracting the verb phrase "find contact" from the review, we conduct static analysis on code to find the classes that query content provider to get contact information. Eventually, we recommend developer to examine the method *ContactsDatabase.queryTextSecureContacts()* since it queries the content provider with URI `<android.provider.ContactsContract$Data: android.net.Uri CONTENT_URI>` to get contact.

Example 4 `org.mariotaku.twidere`: "*Update: uploading photos error.*"

`ReviewSolver` Since some errors in reviews involve sending/receiving intents, we find the classes that contain such intents. For example, after extracting camera related verb phrase "upload photo" from the review, we conduct static analysis to find the classes that send camera related intents. We recommend developer to investigate the method *MediaPickerActivity.openCamera()* because it will send an intent with action `android.media.action.VIDEO_CAPTURE` to other apps.

Example 5 `com.fsck.k9`: "*I like the app, but I receive an error message saying "Failed to send some messages" EVERY time I send an email.*"

`ReviewSolver` If the error reviews list the error messages from the apps, we can look for such messages in the app. For example, after determining the error message in the review, we locate the class that shows this message, and eventually recommend the developer to examine the class *com.fsck.k9.notification.SendFailedNotifications* since it raises this message.

Because `ChangeAdvisor` does not have the knowledge of implementing the general tasks with framework APIs, it cannot locate the function errors related to these general tasks. To overcome this limitation, `ReviewSolver` first uses the Q&As of third party websites to learn the knowledge of implementing these general tasks with the Android framework APIs. Then, it uses these Android framework APIs to help locate the corresponding bugs in the app code. We use **Example 6** to illustrate this procedure.

Example 6 `org.wordpress.android`: "*Won't connect. Get a 404 error when adding wordpress site*".

`ReviewSolver` By searching "404 error" in Stack Overflow [24], we find that this error usually happens when connecting server with the framework APIs (e.g., *WebView.loadUrl()*) [27], [28]. Then, we can locate the bug by finding the classes that call these APIs (e.g., *org.wordpress.android.ui.reader.ReaderPostPagerActivity*, *com.wordpress.rest.RestClient*).

Some users describe the type of exception thrown by the app in the review. After identifying the framework APIs that can cause the exception, we can locate it by checking the corresponding framework APIs. We illustrate this procedure with **Example 7**.

Example 7 `com.fsck.k9`: "*there's a socket exception when it polls*".

`ReviewSolver` This exception is related to "socket". According to the Android official document [29], the *SocketException* is thrown by the methods of the *java.net.Socket* class. By searching the classes that call *java.net.Socket* related APIs, we discover the *com.fsck.k9.mail.store.imap.imapconnection* class containing this exception.

The analysis of the above examples shows that, by extracting semantic information from both the reviews and bytecode, we can remove false positives and false negatives when localizing function errors in reviews.

3 System Design

3.1 System Overview

Fig. 1 shows the procedure of `ReviewSolver`. After crawling reviews from Google Play, the review analysis module identifies function error reviews (Section 3.2). After downloading different versions of all APK files and their corresponding release time, the static analysis module extracts useful information from each APK file (Section 3.3). By combining the information from reviews and APK files, `ReviewSolver` maps the function error reviews to the problematic code (Section 4).

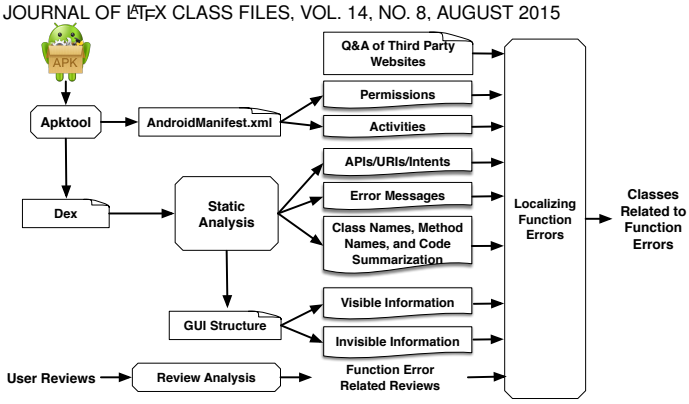


Fig. 1. Overview of `ReviewSolver`: Localizing Function Errors

3.2 Review Analysis

The review analysis module identifies the function error related reviews from those reviews. Then, this module performs sentiment analysis on the sentences of each function error related review and remove positive ones because they do not describe the errors. This module extracts the verb phrases and noun phrases from the natural and negative sentences. These phrases facilitate localizing function errors.

3.2.1 Pre-processing user reviews

We remove the non-ASCII characters and split the remaining content into distinct sentences by using NLTK [30]. To remove typos, we leverage the edit distance [31] to discover the correct word if the word is not found in the dictionary. Abbreviations are replaced with their original words (e.g., "pls" to "please", "pic" to "picture"). For each sentence in the review, we leverage Stanford Parser [32] to construct the parse tree and the typed dependency among words.

The parse tree contains the phrases of the sentence and the Part Of Speech (POS) tags of words. Each phrase occupies one line. For example, *NP* in Fig.2 means noun phrase and *VP* in Fig.2 refers to verb phrase. The typed dependency relation refers to the grammatical relation between two words [33]. For example, *dobj* in Fig.2 means direct object.

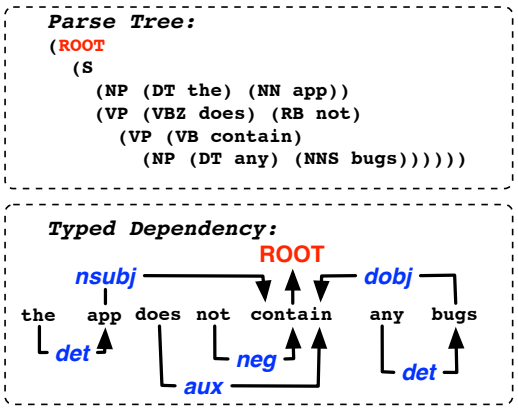


Fig. 2. Syntactic Analysis: Parse tree and typed dependency of the sentence: "the app does not contain any bugs".

3.2.2 Identifying function error reviews

We use supervised machine learning algorithms to identify the function error reviews described in Section 2. In particular, we use the TF-IDF values, N-Grams (N=2,3) as features, because these features are widely used in text classifications based on supervised machine learning models [26], [34]–[36]. TF-IDF (i.e., Term Frequency-Inverse Document Frequency) measures how important a word is to a review [37]. It is calculated by multiplying term frequency (TF) and inverse document frequency (IDF). TF measures how frequently a word occurs in a review while IDF measures how important a word is. The frequent words are less important (e.g., "an", "the").

$$TF(t) = \frac{Number\ of\ times\ word\ t\ appears\ in\ a\ review}{Total\ number\ of\ words\ in\ the\ review}$$

$$IDF(t) = \log \frac{Total\ number\ of\ reviews}{Number\ of\ review\ with\ word\ t\ in\ it}$$

N-Grams are a set of co-occurring words within a given window [38]. For example, given the sentence shown in Fig. 2, we extract "the app does", "app does not", "does not contain", "not contain any", and "contain any bugs" as N-Gram features (N=3). Note that without conducting syntactic analysis on each sentence, the TF-IDF (which only considers distinct words) and N-Gram features (which has fixed window size) cannot recognize the relation between negation words (e.g., "not") and error-related words (e.g., "bug"). Therefore, the classifier (e.g., those in [26], [34]) will regard the sentence of Fig. 2 as a function error review by mistake (i.e., a false positive). To address this issue, we analyze the typed dependency relations of the sentence. Since both "bug" and "not" are related to verb "contain", we regard "bug" as being related to "not", and thus remove the word "bug" related features before classification.

To train a classifier for identifying function error reviews, we create a training dataset with 700 positive reviews and 700 negative reviews. We test multiple algorithms (including, naive bayes, random forest, SVM, max entropy, boosted regression trees) by performing 10-fold cross-validation. Table 2 lists the precision, recall rate, and F1-score of each classifier. Finally, we adopt the boosted regression trees [26], [39] because it has the best performance (i.e., precision 91.4%, recall rate 92.0%, and F1-score 91.6%). The boosted regression trees aggregate the result from a sequence of decision trees. To train an expressive model, the algorithm iterates multiple times. During each iteration, this algorithm selects the feature that best partitions the data to create tree models. It will also adjust the weight of the samples classified incorrectly to enable the next tree to correctly classify them [40]. When performing the classification, we do not split reviews into sentences because considering individual sentences may miss the context information in other sentences.

TABLE 2
Classifier selection: 10-fold cross-validation result

Classifier	Precision	Recall	F1-Score
Naive bayes	56.8%	99.6%	72.2%
Random forest	85.3%	87.8%	86.5%
SVM	87.5%	84.2%	85.7%
Max entropy	55.6%	99.7%	71.3%
Boosted regression trees	**91.4%**	**92.0%**	**91.6%**

3.2.3 Removing positive sentences

Since some positive sentences of function error reviews do not describe errors, we identify them through performing sentiment analysis [41], [42]. These positive sentences are not used when localizing the errors because they will generate false positives. In previous version [19], we directly used

the scores of reviews to filter positive reviews (i.e., Rated 4 or 5 stars out of five stars). We do not employ this method because many function error reviews have high scores. For example, the function error review "*...Crashes in the middle of writing a post and there is no backup...*" has a high score (i.e., Rated 4 stars out of five stars). To measure how many reviews with high scores (i.e., Rated 4 or 5 stars out of five stars) describe the function errors, we randomly selected 900 user reviews of 18 apps. Then, we manually read these user reviews and determine the number of function error reviews for each score. The result is shown in Table 3. In total, we discovered 333 function error reviews, and 24.6% of them (i.e., (64+18)/333) are reviews with high scores (i.e., Rated 4 or 5 stars out of five stars). This result shows that if we use the scores of reviews to filter positive reviews, we will lose about a quarter of function error reviews.

TABLE 3
Number of user reviews with different scores and the number of function error reviews contained in them.

Review Score	Number of Review	Number of Error Review
1	150	112
2	97	64
3	118	75
4	155	64
5	380	18
Total	900	333

One function error review contains multiple sentences, which are obtained through NLTK (Section 3.2.1). Then, we employ the sentiment analysis tool to process them. If the result of one sentence is positive, the sentence will be discarded since these positive sentences usually do not describe the errors (e.g., "*love u first of all for making this app...*"). Otherwise, if the result is negative or neutral, the sentence will be kept since it may describe the function error. Jongeling et al. [43] found four sentiment analysis tools commonly used in previous software engineering studies (i.e., SentiStrength [44], [45], Alchemy [46], Stanford NLP sentiment analyser [47], and NLTK [48]). Since Alchemy was retired in March 2018, we conducted an additional experiment to choose the best sentiment analysis tool from SentiStrength, Stanford NLP sentiment analyser, and NLTK. In this experiment, we compare the ability of identifying the negative reviews by using these three sentiment analysis tools. For the 900 randomly selected user reviews (same as Table 3), we manually read each review. If it contains at least one negative sentence, we add one "#NEG" label for it (e.g., "*A bad app, often crash*"). Then, we use these three sentiment analysis tools to process these reviews with the same procedure. If the tool finds one negative sentence in the review, it also adds one "#NEG" label for it. Finally, we compare the number of negative reviews discovered by these three tools with that discovered through manual annotation. The result is shown in Table 4. For the 428 negative reviews discovered through manual annotation, the number of negative reviews also discovered by SentiStrength (i.e., 207) is much higher than that of NLTK (i.e., 51) and Stanford NLP sentiment analyser (i.e., 56). Since the result of SentiStrength is most similar to the result manual annotation, we think that SentiStrength achieves the best performance. We select SentiStrength to perform sentiment analysis on each sentence of one review.

Especially, if the sentence contains adversative coordinating conjunctions (e.g., "but", "whereas", "nevertheless"), this sentence expresses or indicates contrast or opposite meaning between two statements [49], [50]. The negative or neutral part of the sentence may describe the errors while the positive part of the sentence can be discarded. Thus, we leverage the parse tree of each sentence to identify the adversative coordinating conjunctions by using the "CC" label, which means "coordination". We combine the words before or after the adversative coordinating conjunctions to construct one distinct sentence (similar to RISING [15]). Then, we also employ CoreNLP to obtain the sentiment analysis result of the constructed sentence. For example, the review "*It's a great app BUT since the last update, my stats page doesnt work properly.*" will be transformed into two sentences: (1)"*It's a great app*", the positive part of the sentence. (2)"*since the last update, my stats page doesnt work properly.*", the negative part of the sentence. We discard (1) and keep (2) for further analysis.

TABLE 4
Number of reviews for each score, number of negative reviews discovered through manual annotation, and number of negative reviews also discovered through three sentiment analysis tools (i.e., SentiStrength, NLTK, Stanford NLP sentiment analyser).

Score	Num of Review	#Neg Manual	#Neg SentiStrength	#Neg NLTK	#Neg Stanford
1	150	144	77	26	25
2	97	90	48	7	13
3	118	92	40	12	12
4	155	72	29	5	4
5	380	30	13	1	2
Total	900	428	207	51	56

3.2.4 Extracting verb phrase and noun phrase

To capture the semantic information of function error reviews, we extract the verb phrase and noun phrase by using the parse tree and typed dependency relations. The verb phrase contains a verb and its object (e.g., "import contact"). The noun phrase contains a word or group of words containing a noun (e.g., "the last phone call"). We do not employ the bag-of-words model to represent the semantic information of review because the word frequency cannot capture the part-of-speech (POS) tags of words. For example, although both "contact me if you like" and "import contact" contain the word "contact", the former is a verb (cannot be mapped to the behavior of the app) and the latter is a noun (can be mapped to the access of contact list through content provider in the app). Since the verb/noun phrases retain the part-of-speech tags of words, we can remove the false mappings from reviews to code(Section 4).

The verb phrase is extracted from typed dependency. For the sentence shown in Fig.2, as the verb is "contain" and the object is "any bugs", we acquire the verb phrase (i.e.,"contain any bugs") by checking the typed dependency relation (i.e., *dobj*, *nsubjpass*) between words. The noun phrase is obtained through parse tree. For each line of the parse tree, if the line starts with NP (i.e., noun phrase), the phrase of the line will be extracted as noun phrase. For example, for the sentence of Fig.2, we extract two noun phrases (i.e., "the app" and "any bugs") from the parse tree.

When extracting the verb phrases and noun phrases from function error related reviews, we found that some reviews contain more than one sentence. Some of the sentences describe the context of function errors while others

do not. For these sentences unrelated to function errors, if we extract verb phrases and noun phrases to localize the function errors, we will obtain many false mappings between the function error reviews and code. Previous researchers [51] found that, when writing reviews, some users will describe the features that can be added to improve/enhance the apps' functionalities (i.e., "feature request"). Moreover, some other users will use reviews to obtain some information from developers (i.e., "information seeking"). Finally, some users will provide some information for the developers through reviews (i.e., "information giving").

Although the sentences related to "feature request", "information seeking", and "information giving" are important for developers, they are not related to the function errors of the apps. Thus, they should be ignored when extracting the verb phrases and noun phrases to localize the function errors to avoid generating false mappings. To achieve this goal, we employ the state-of-the-art system developed by Panichella et al. [51] to identify and filter the "feature request", "information seeking", and "information giving" related sentences. This system combines NLP, text analysis, and sentiment analysis techniques to classify sentences of user reviews into these three categories automatically. We also tested the effectiveness of this system [51] by using 1,500 randomly selected function error reviews. The result shows that it successfully filtered 146 sentences (e.g., "*I use Nougat (7.0) android version*").

3.3 Static Analysis

The static analysis module is intended to discover the behaviors contained in the APK files. Since the developers keep changing the app code according to user reviews and releasing new versions of APK files on app market, we first download all versions of APK files and their corresponding release time in order to avoid incorrect mapping between function error reviews and code (Section 3.3.1). Given an APK file, we analyze its `AndroidManifest.xml` file and `Dex` file to extract seven kinds of information to facilitate mapping function error reviews to code (Section 3.3.2).

3.3.1 Downloading APK Files

For each app, we download all versions of APK files and their release times to avoid mapping the function error review to non-existent classes or methods. The principal reason is that the developers may modify the app code (e.g., add/change/remove some classes and methods) when releasing new versions of APKs. To correctly locate the function error described in the review, we check the publication time of the review. Then, we identify the APK file released before the publication time to localize the error. Otherwise, incorrect mappings will be generated (i.e., mapping the review to the classes or methods created after the publication of the review). For example, we compare two versions of APK files of the app `org.thoughtcrime.securesms`. The version released on January 24, 2018 contains 1,850 classes (ignoring third party libraries). Another version released on December 14, 2018 contains 2,446 classes (ignoring third party libraries). Compared with the version released on January 2018, 113 classes have been removed (113/1850, 6.1%), and 709 classes are new created ones (709/1850, 38.3%). For one function error review published on January 2018, if we map it to these new created classes, an incorrect mapping is generated (i.e., these classes do not exist when users finding the error).

Various websites can be used to download all versions of APK files. For one open-source app, the websites (e.g., F-droid [52] and Github [53]) provide the service of downloading different versions of APK files, their source code, and their release time. For one closed-source app, some third party websites (e.g., ApkMonk [54], APKPure [55]) allow downloading different versions of APK files and their release time.

3.3.2 Extracting Behaviors from APK File

Given an APK file, we first extract the `AndroidManifest.xml` file and `Dex` file from it. Then, we use Vulhunter [56] to process the `Dex` file and create android property graph (APG) of the app. APG combines abstract syntax tree (AST), method call graph (MCG), and data dependency graph (DDG). When building the DDG, we leverage the IccTA [57] to identify the target component of intent.

Extracting permissions and activities We parse the `AndroidManifest.xml` file to extract the permissions and activities. The starting activity is identified through the action "android.intent.action.MAIN" and the category "android.intent.category.LAUNCHER" in the intent filter.

Extracting APIs/URIs/intents, error message, class/method names, and method summarization We analyze the APG to identify three kinds of information (i.e., APIs/URIs/intents, class/method names, error messages). Because the raw method name may fail to reflect its function (i.e., the method name is incorrect or it has been replaced with meaningless characters [58]), apart from method names set by developers, we also employ the deep learning method Code2vec [59] to generate the code summarization of each method based on the statements included in it.

To identify APIs, we check all the *assign* statements and *invoke* statements contained in AST. If the invoked method name is a framework API, we record it so that they can be used to localize API related function errors.

To identify URIs, through which apps can get information (e.g., contacts), we first determine the content provider operations (e.g.,*ContentResolver.query()*), and then conduct backward taint analysis by traversing the DDG [60]. In particular, the traversal starts from the statements related to content provider operations and ends at the statements that define local variables. All URI used in code are recorded. PScout [61] uses static analysis to obtain the mapping between the permissions and their related APIs/URIs. After discovering the APIs/URIs used in code, we leverage the mappings proposed by PScout to find out the permissions used in code.

By sending the intents to other apps, an app can call other apps to perform specific tasks. For example (Fig. 3), the app `com.fs.catw` sends out an intent (i.e., type is `android.media.action.IMAGE_CAPTURE`) to the the camera app for capturing an image and obtaining it. To identify the intents sent by the app, we first collect all intent related statements (e.g, *Activity.startActivityForResult()*), and then perform backward taint analysis on it. The sources of this taint analysis are the statements that call APIs to send out intent. The sinks are the statements that create new variables (i.e., statements that do not contain any outgoing data dependency relation). All string

parameters appeared in the path will be recorded (e.g., `android.media.action.IMAGE_CAPTURE` in Fig. 3).

Fig. 3. Code Example: Send intent to take picture

If error occurs, an app may notify users the details [62] by using *AlertDialog*, *TextView*, or *Toast*. To identify the error message pop-up in each class, after determining the statements that invoke error message related APIs (e.g., *AlertDialog.setTitle()*, *AlertDialog.setMessage()*, *TextView.setError()*, and *Toasts.makeText()*), we conduct backward taint analysis. The sources of this taint analysis are the statements that call the APIs to pop-up error message. The sinks are also the statements that create new variables. All the string parameters appeared on the path are recorded.

After building the AST, we record all class names and method names. Since class and method names may provide information about the corresponding classes and methods [63], we extract them and use them to locate app specific task errors described in user reviews (Section 4.1). If the extracted method names do not correctly describe the functions of their included statements, they cannot be used to localize the app specific task related errors (Table 1 case (1)). Such issue exists if the developers do not carefully design the method names or if the developers employ obfuscation technique to hide the method names [58], [64]. To tackle this problem, we employ the state-of-the-art code summarization system Code2vec [59] to summarize the function of each method. In Section 4.1.1, we use the summarization of each method (i.e., a list of words) to assist localizing function errors. Code2vec learns a neural model from the AST paths of a dataset of methods. This model represents the code snippets as distributed vectors and then it can predict semantic properties of new code snippet. The reason is, when training the model, the input to the Code2vec model is a code snippet (i.e., C) and a corresponding tag/label L (e.g., method name). Through training, a tag vocabulary is learned:

$$tags\_vocab \in R^{|Y| \times d}$$

where Y is the set of tag values found in the training corpus. Each row of $tags\_vocab$ shows the embedding of one tag (e.g., the method names "contains", "canHandle"). Then, the predicted distribution of the model is computed as the normalized dot product between the code vector v and each tag embedding. In other words, given code snippet C, the probability that a specific tag (i.e., method name) y_i is the normalized dot product between the vector of y_i and the code vector v. We downloaded 1300 open-source Android projects from F-droid [52] to train a new neural model for Android apps. We do not use the existing model provided by Code2vec (trained by using 14 millions Java methods [65]) since it cannot handle Android specific life-cycle methods (e.g., *onCreate()*) and UI callbacks (e.g., *onClick()*). To measure the performance of the trained neural model, we conduct an experiment by using the source code of 22 open-source apps downloaded from F-Droid. For each app, we first compile the source code to get an obfuscated APK file by using ProGuard [58]. The class names and method names are hidden (i.e., replaced by "a", 'b"). Then, we use the trained neural model to generate code summarization for each method. Finally, we use the words extracted from the raw method names of source code as the ground truth and measure how many words contained in the raw method names are included in the code summarization. The result shows that the method names of these apps contain 10,688 words, and the trained neural model can predict 3,674 of them (i.e., 3674/10688=34.4%). This result shows that, even if the method names are removed through obfuscation, the code summarization can also discover about 35% of words contained in the raw method names.

Extracting visible/invisible label information from GUI We first recover the structure of each activity, and then extract the visible and invisible label information from it. The former includes the texts shown in GUI. If the user review mentions such information, we look for the UI component that contains the corresponding text for localizing problematic code. The invisible label information refers to the ids of widgets/UI components in the GUI. Since developers may include the purpose of the widget when setting the id (e.g., `quoted_text_edit`), we can use them to understand the function of each widget (e.g., "edit text").

We use `GATOR` (Version 3.6 [66]) to recover the GUI structure of all activities [67]. GATOR first parses the manifest file (to identify the activities), the layout file (to get the parent-child relationship between widgets), and resource id file (to obtain the mapping between id names and values). Then, it inspects each method and conducts reference analysis to construct the constraint graph of GUI related objects. Finally, GATOR combines the information obtained from the layout file and the dynamically generated widgets inferred from the constraint graph to reconstruct the GUI structure.

After obtaining the GUI structure of each activity, to identify the text displayed by the app, we extract the values of the `android:hint` and `android:text` attributes. If the value of one attribute is an id of String resource, we inspect the `res/values/strings.xml` file to get the corresponding String (e.g., "@string/account_setup_hint" in Fig. 4).

To extract the invisible information from the ids of the widgets of the GUI, for the id of each widget, we split it into a series of words. For example, in Fig. 4, the id `show_password` is transformed to "show" "password". Since some developers use abbreviations when setting the ids (e.g., use "btn" to represent "button"), for each extracted word, we design an "abbreviation matching method" to identify the abbreviations and replace them with the raw words. In detail, through checking the UI related abbreviations summarized in [68], [69], we obtained 39 UI related nouns and their corresponding abbreviations (e.g., "rb" means "radio button"). For each word extracted from the id, we compare it with the obtained abbreviation list to determine if it is an abbreviation or not (e.g., "btn"). If so, we replace the abbreviation with the corresponding raw word (e.g., "button").

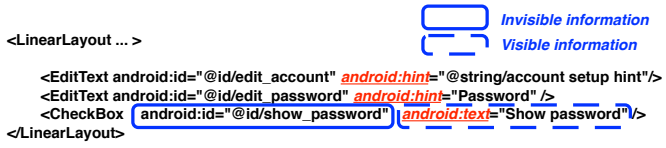


Fig. 4. Snippet of a layout file

4 LOCALIZING FUNCTION ERRORS

By correlating the information extracted from user reviews and APK files, `ReviewSolver` first localizes app specific errors (Section 4.1) and general errors (Section 4.2), and then ranks the selected classes before recommending them to the developer (Section 4.3). Note that, for each function error review, we check its publication time and then identify the last version of APK file released before this time to locate the error in the app code.

4.1 Localizing App Specific Errors

For the context information defined in Tab. 1, six of them (i.e., app specific task, updating app, GUI, error message, opening app Activity, registering account interface) contained in user reviews are app specific. We describe how to use them to localize the function errors in this section.

4.1.1 Using Class/Method Name and Code Summarization

If the function error appears when performing app specific tasks, for each verb phrase extracted from function error review, we check whether it is similar to the raw method name of each method or the summarization generated by Code2vec (Section 3.3.2). If so, we recommend the developer to check the corresponding method. Since the raw method name is a String that cannot be used when calculating the semantic similarity, we leverage the camel case to convert the method name to verb phrase by referring the method described by McBurney et al. [63]. For example, we transform *getEmail()* to "get Email". If the method name only contains a verb, we use the words extracted from the class names as the object of the verb phrase (e.g., we transform *MessageListFragment.move()* to "move Message List Fragment"). Since the life-cycle methods in Android apps (e.g., *onCreate()*) may have the same method names, to correctly describe their functions, we remove their stopwords (e.g., "on") and combine the remaining verbs with component names to create verb phrase.

To determine whether two phrases are similar or not, we leverage Word2Vec [70] to calculate the semantic similarity between two phrases, because representing the word with a series of words can capture syntactic and semantic regularities between words [71], [72]. More precisely, by using the model trained on Google News dataset (contains 300-dimensional vectors for 3 million words and phrases) [73], we transform each word ($word_i$, $i = 1, .., n$) of the $phrase$ into a 300-dimensional vector. We combine them to get the vector of the phrase.

$$Vector(phrase) = \frac{1}{n}\sum_{i=1}^{n} Vector(word_i)$$

Then we calculate the cosine similarity between two phrase vectors. If the similarity is higher than the threshold value (0.68 by referring [74]), we regard them as similar ones.

$$CosineSimilarity = \frac{Vector(phrase1) \bullet Vector(phrase2)}{\|Vector(phrase1)\|\|Vector(phrase2)\|}$$

4.1.2 Using Visible/Invisible Label Information

To localize the errors related to GUI, we compare the verb/noun phrase extracted from review with the visible and invisible label information extracted from code. For the former, we check two kinds of noun phrase extracted from review: (1) If the user explicitly points out the widget (e.g., "reply button"), we regard the phrase as GUI related phrase. In this case, we extract the word modifying the widget (e.g., "reply") and look it up on the visible label information; (2) If the user implicitly describes the type of the issue (e.g., "Certificate issues"), this phrase may be relevant to the GUI. The reason is that users interact with the GUI of the app to perform some task (e.g., "uploading photo"). If errors appear during this procedure, they will describe the task (e.g., "uploading photo error") in the corresponding review. In this case, we extract the word modifying the issue related noun (e.g., "Certificate") and search it on the visible label information. For the latter, we check the verb phrase extracted from the user review by comparing its semantic meaning with the verb phrase transformed from the invisible label information.

When reading the function error review, users can also vaguely describe the error (i.e., some functions cannot work). Manually summarizing the semantic patterns from these sentences is time-consuming and error-prone. Thus, we select the state-of-the-art pattern extraction tool NEON [75] to extract semantic patterns. After parsing the semantic graphs of sentences, NEON identifies the recurrent grammatical structures (i.e., syntactical rules) appearing in them and then regards them as the patterns. After selecting 100 such sentences and using them as the input of NEON, we obtained four semantic patterns (shown in Table 5). $[function]$ means the problem function. NEG means negation related words (e.g., "cannot") and phrases (e.g., "does not"). To localize such errors, we first extract the $function$ word of these patterns, and then look them up on the GUI's visible label information. The activities that contain these words will be recommended to the developer. For example, for P2, we recommend the developer to check the activity that contains the verb "register".

TABLE 5
Semantic patterns of vaguely describing the error.

#	Semantic Pattern	Example
P1	$[function]$ NEG $work$	*"sync does not work"*
P2	$[subject]$ NEG $[function]$	*"I cannot register"*
P3	$[function] fail$	*"Login always fails"*
P4	$[function] stopped$	*"Update button has stopped"*

4.1.3 Localizing Errors Related to Error Message

Users may describe the error message precisely. For example, given the review *"I receive an error message saying "Failed to send some messages""*, we extract the error message and compare it with the error messages extracted from the app's APK file.

Sometimes, the user may simply point out the type of the error, and hence we first check whether the noun phrases contain error related words (e.g., "error", "bug", "fault"). If so, we extract the word that modifies these error related words. Then, we check all the APIs invoked in code. If the API's description mentions this word, we recommend the

corresponding class to the developer. For example, in the review "*a **connection error message** at the bottom*", since the user mentions that the error is related to "connection", we recommend the developer to check the classes that call the API *HttpURLConnection.getInputStream()*.

4.1.4 Localizing Errors Related to Opening App Activity

If the function error review contains verb phrases such as "open app", "launch app", or "start app", the error may appear when the app is launched. Since the *onCreate()*, *onStart()*, and *onResume()* methods of the starting activity are called sequentially when an app is launched, for this kind of error, we recommend the developer to check these three methods of the starting activity.

4.1.5 Localizing Errors Related to Account Registration

If the function error review contains verb phrases such as "register account", "sign in", "login in" or if the review contains noun phrase such as "registration", the error may appear when registering account. For this kind of error, we recommend the developer to check the activity related to registering account. We search the text content of each activity and report the activity that contains phrases related to account registration (e.g., "sign in", "login").

4.1.6 Localizing Errors Related to App Updating

If the function error review contains updating related phrases (e.g., "update app", "latest update", "new update", "recent update"), this error may be caused by the app update. For such kind of error, we first check other verb/noun phrases of the review. If they can be mapped to the app specific error or general error, we extract the corresponding classes and recommend them to developers. Otherwise, we recommend the developer to check the code difference between the latest two versions.

4.2 Localizing General Errors

For the context information defined in Tab. 1, three of them (i.e., API/URI/intent, general task, exception) are generally contained in user reviews of different apps. We describe how to use them to localize the function errors in this section.

4.2.1 Localizing Errors Related to APIs/URIs/Intents

If the errors described in the reviews are related to the APIs, URIs, or intents (i.e., Table 12 case (7)), we propose Algorithm 1 to locate them. For API, we compare the verb phrase extracted from review with the verb phrases related to the API (line 3-5 in Algorithm 1). For URI, we compare the object of the verb phrase extracted from review with the noun phrases related to the URI (line 11-13 in Algorithm 1). For intent, we compare the object of the verb phrase extracted from review with the noun phrases related to the intent (line 19-21 in Algorithm 1). If they are similar, we recommend the developer to check the API/URI/intent and corresponding class.

We extract the verb phrase related to API from API signature, description, and permission. The signature of an API contains its class, return value, method name and parameters (e.g., <*android.location.Address: double getLatitude()*>). We convert the API signature into verb phrase by using the method described in Section 4.1. We also extract verb phrases from its official description by using the typed dependency [33]. For example, we extract verb phrases such as "open communication link", "establish connection" from the description of the API *URLConnection.connect()*.

ALGORITHM 1: Find the classes related to the API/URI/intent.

Input: $VerbPhrase$: Verb phrase extracted from the review; $ApiSet$: APIs provided by Android document; $UriSet$: URIs provided by PScout; $IntentSet$: intent provided by Android document.
Output: $ClassList$: the classes related to API/URI/intent.

```
1  Function LocateApiUriIntent(VerbPhrase, ApiSet, UriSet, IntentSet):
2      ClassList=[]
3      foreach API in ApiSet do
4          ApiPhraseList = getAPIRelatedPhrases(API)
5          foreach ApiPhrase in ApiPhraseList do
6              if Similar(VerbPhrase, ApiPhrase) then
7                  ClassList.add(getCaller(API))
8              end
9          end
10     end
11     foreach URI in UriSet do
12         UriNounList = getURIRelatedNouns(URI)
13         foreach UriNoun in UriNounList do
14             if Similar(getObj(VerbPhrase), UriNoun) then
15                 ClassList.add(getCaller(URI))
16             end
17         end
18     end
19     foreach Intent in IntentSet do
20         IntentNounList = getIntentRelatedNouns(Intent)
21         foreach IntentNoun in IntentNounList do
22             if Similar(getObj(VerbPhrase), IntentNoun) then
23                 ClassList.add(getCaller(Intent))
24             end
25         end
26     end
27     return ClassList;
```

For each verb phrase extracted from review, we use the Word2Vec [70] to calculate the semantic similarity and determine if it is similar to any verb phrases extracted from the method name or description of the API or not. If true, we also suggest developers check the API. If the verb of the verb phrase extracted from review is related to information collection (e.g., "gather"), access (e.g., "read"), or utilization (e.g., "use") related verbs [76] and its object is similar to the personal information protected by permission, we also recommend the developer to check this permission related API and corresponding class.

Since there is no official description of URI, we cannot extract verb phrases related to URI. To map the function error review to URI, we compare the noun phrases described in review with the noun phrases related to the URI. To obtain the latter, we first leverage PScout [61] to get the permission related to the URI. Then, we regard the noun phrase extracted from the permission description [77] as the noun phrase related to URI. For example, the URI "`content://call_log`" is protected by the `READ_CALL_LOG` permission. We extract "call log" from the permission description (i.e., "*Allows an application to read the user's call log.*").

Moreover, we manually define the noun phrase of each intent by referring the Android official document. The Android official document [78] provides 11 kinds of common intents. For example, "camera" is related to the intent with the action `android.media.action.IMAGE_CAPTURE`.

4.2.2 Localizing Errors Related to General Tasks With Q&As of Third Party Websites

If the error described in the user review appears when performing general tasks (e.g., download files), we search the Q&As of third party websites to obtain their implementation. We first download the questions and answers from the website (**Step 1**). After identifying the framework APIs commonly used in the implementation (**Step 2**), we use these framework APIs to locate the error in app code (**Step 3**).

Step 1:To obtain the implementation (i.e., code snippets) of general tasks, we select the Q&As in Stack Overflow [24], which is the largest community for developers [79]. We download 1,272,968 Android related questions and their corresponding 1,054,122 answers [80]. Each question has a short title that summarizes the encountered problem and a long text that describes the details of the problem. One question may have one or more answers.

Step 2: Both the question description and the answers may have code snippets, which can be extracted through the `<code>` tag. Thus, we can build up mapping between the question title and the framework APIs contained in the corresponding code snippets. To identify the framework APIs related to each question, we design a parser to extract the called framework APIs from the code snippets. For each line of the code snippet, we check if it defines a new object or not. If so, we extract the class name and the name of the object. If one line of the code snippet does not define a new object, we employ the javalang [81] to extract the called method name. Once the name of the called method is extracted, we combine the class name of the object with the extracted method name. Then, we determine if they are consistent with that of one framework API or not. If true, we record this framework API since it is related to the question. Finally, we obtained 1,079,053 code snippets. Each code snippet contains at least one framework API identified by javalang. 553,681 of them are from Android related questions and 525,372 of them are from Android related answers.

Step 3: Based on the framework APIs used to implement the general tasks, we design the algorithm 2 to locate the corresponding errors. For each verb phrase of the function error review, we first identify the questions whose titles contain the same verb phrase because these questions may contain the code implementation (i.e., code snippets) (Line 4-5). For each matched question, we extract the invoked framework APIs from their code snippets (Line 6). For the same task, developers' code implementation are significantly different but the invoked framework APIs are similar. We count the frequency of each framework API (Line 8). The top k most frequent framework APIs (Line 12) and the corresponding classes that invoke these framework APIs are identified (Line 14-15). Currently, we set k as 5.

4.2.3 Localizing Errors Related to Exception

Since the exception discovered by users are generated when invoking framework APIs or methods defined by developers, to locate these errors in app code, we first identify the exceptions thrown by the framework APIs or the methods defined by developers by using the Android document and the AST (**Step 1**). Then, we map the user review to these exceptions (**Step 2**).

ALGORITHM 2: Find the classes related to general tasks.

Input: $VerbPhrase$: Verb phrase extracted from the review;
$QuestionSet$: Set of Android related questions in Stack Overflow;
Output: $ClassList$: the classes related to this verb phrase.

```
1  Function LocateGeneralTasks(VerbPhrase, QuestionSet):
2      ClassList=[]
3      APICount={}
4      foreach Question in QuestionSet do
5          if containInTitle(VerbPhrase, Question) then
6              APIs = getRelatedAPIs(Question);
7              foreach API in APIs do
8                  APICount[API] = APICount[API] + 1;
9              end
10         end
11     end
12     topAPIs = getMostFrequentAPIs(APICount);
13     foreach API in topAPIs do
14         relatedClasses = getClassesCallAPI(API);
15         ClassList.addAll(relatedClasses);
16     end
17     return ClassList;
```

Step 1: For the framework APIs, the exceptions that they can throw are described in the official document. After parsing the Android official document, we identify 5,808 framework APIs distributed in 1,172 classes that throw 195 types of exceptions. For example, the framework API *android.location.LocationManager.requestLocationUpdates()* throws two kinds of exceptions (i.e., *IllegalArgumentException*, *SecurityException*). For the methods defined by developers, due to the absence of the software documents, we check the statements contained in each method to determine the types of exceptions it can catch.

Step 2: For each noun phrase extracted from the review, if it contains the String "exception" or "Exception", we extract the words before it as the type of the exception. Then, based on the result of **Step 1**, we identify the framework APIs or methods defined by developers throwing this type of exception. Finally, we output the classes that call these framework APIs or the methods defined by developers because these classes can throw the exception mentioned in the review.

4.3 Ranking the Classes

Since one function error review may contain multiple types of context information, we employ multiple approaches to map function error reviews to code. The verb/noun phrases of one review may be mapped to multiple methods of multiple classes. For example, the review "*I get an out of memory error message and can't take pictures*" contains two types of context information. One is an error message (i.e., "out of memory"), and the other is API (i.e., "take picture"). To avoid generating too many code snippets for developers, we combine the methods included in the same class together and output the most important classes for users. In detail, after computing the importance of these classes, we recommend the top N most related ones to developers (Currently, N is 15). Assume that we find n mappings between verb/noun phrases and classes (i.e., $m_1, m_2, ..., m_n$), $m_i =< phrase_j, class_k >$, by using the approaches proposed in Section 4.1 and Section 4.2. For each class, we calculate the importance by counting the number of mappings between different phrases and the target class. For example, if we find one mapping $< phraseA, classA >$,

the importance of $classA$ will be increased by one. The selected classes are ranked according to their importance. If many classes share the same importance, we analyze the class dependency relations of these classes and select the classes that are dependent on the largest number of classes. The reason is, if one class is implemented by using many other classes, it has a higher probability to implement the core function of the app. Thus, it has a higher probability to trigger the function errors. A class dependency between classes $Class_A$ and $Class_B$ indicates that one method is defined in $Class_A$ and this method is invoked by the methods of $Class_B$ [82].

5 Experimental Result

After describing the dataset used in experiments (Section 5.1), we conduct extensive experiments to answer the following research questions:

RQ1: Can `ReviewSolver` correctly identify reviews related to function errors (Section 5.2)?

RQ2: How is the performance of `ReviewSolver` compared with the state-of-the-art system `ChangeAdvisor` [13] and `Where2Change` [14] (Section 5.3)?

RQ3: How many function error related reviews can be addressed by `ReviewSolver` (Section 5.4)?

5.1 Dataset

To measure how many function error related reviews can be solved, we select 18 apps that can be downloaded from Google Play and provide source code in F-Droid or Github. We first downloaded all versions of APK files and their release time. Then, we collect their user reviews from Google Play. Tab. 6 shows the APK id, name, number of APK files of each app. For each app, we download the latest 1,500 reviews to analyze (i.e., the dataset contains 27,000 user reviews in total).

TABLE 6
The APK id, name, number of APK versions of each app

APK Id	APK Name	#APK
org.mariotaku.twidere	Twidere	12
com.zegoggles.smssync	SMS Backup+	44
org.thoughtcrime.securesms	Signal	47
com.totsp.crossword.shortyz	Shortyz Crosswords	9
com.fsck.k9	K-9 Mail	80
com.andrewshu.android.reddit	rif is fun for Reddit	59
fr.xplod.focal	Focal	1
org.geometerplus.zlibrary.ui.android	FBReader	35
com.battlelancer.seriesguide	SeriesGuide	109
org.wordpress.android	WordPress	205
com.kmagic.solitaire	Solitaire	1
org.coolreader	Cool Reader	7
cgeo.geocaching	Cgeo	93
com.joulespersecond.seattlebusbot	OneBusAway	66
com.achep.acdisplay	AcDisplay	31
de.danoeh.antennapod	AntennaPod	11
com.frostwire.android	FrostWire	271
com.ichi2.anki	AnkiDroid	551

To answer RQ2, we first employ `ReviewSolver` to identify the function error related reviews, and then apply `ReviewSolver`, `ChangeAdvisor` [13], and `Where2Change` [14] to mapping such reviews to code. `ChangeAdvisor` and `Where2Change` are implemented by the authors of corresponding papers. We downloaded them from [83] and [84], respectively. We invite three PhD students to construct the ground truth of the mappings from reviews to code by exploiting two kinds of documents (i.e., bug reports and release notes). Each student has three years of experience in developing Android apps, and their research direction is identifying the security/function issues of Android system and apps. These students work together, and the final mappings must be agreed by at least two students.

• Fig. 5 shows the procedure of leveraging bug reports to correlate reviews and code. After reading a function error related review, the student identifies the bug described in it and then looks for the bug in the existing bug reports. If we found that the bug has been fixed, the corresponding code files modified by the developers are regarded as the code related to the review.

• Fig. 6 demonstrates the procedure of using release notes of open-source apps to map reviews to code. When publishing a new version of app, the developers can write the release note to tell users the changes developers have made to the new version of app, including the newly added features, fixed bugs, and patched vulnerabilities [85]. After reading a function error related review, the student identifies the bug described in it and then searches the release notes describing that this bug has been fixed. If one release note fixing this bug is found, the student compares the code of this version with the previous version to locate the changed files and then regards them as the code related to the function error review.

Finally, the ground truth constructed by using bug reports contains 8 apps and the ground truth constructed by using release notes contains 6 apps. Other apps that do not contain bug reports or release notes are not included. We do not use the datasets provided by `ChangeAdvisor` [83] and `Where2Change` [84] as the ground truth. The reason is, although these two datasets contain the apps' source code and the corresponding user reviews used in the evaluation, they do not provide the mapping between each function error review and its related source code, which have been manually verified to evaluate their system performance. Thus, we need to manually create the ground truth (i.e., mapping between function error reviews and code) by downloading and checking the bug reports and release notes.

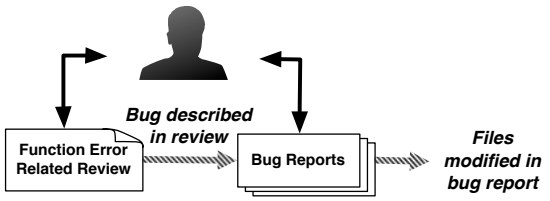


Fig. 5. Procedure of building ground truth with bug reports

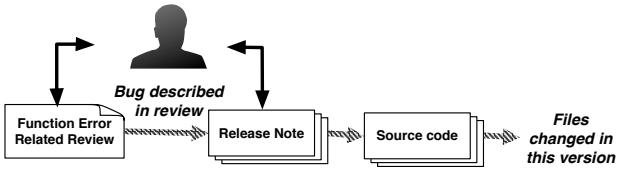


Fig. 6. Procedure of building ground truth with release notes

5.2 Review Identification Performance

To evaluate the performance of classifying function error related reviews, we adopt the two datasets and process them

with ReviewSolver. The dataset provided by Ciurumelea et al. [26] contains 199 reviews (87 of them are function error related ones). Another dataset provided by Maalej et al. [20], [21] contains 747 reviews (369 of them are function error related ones). The authors of these two datasets have added the labels for the user reviews that are (not) function error reviews. We directly use them as the ground truth of identifying function error related reviews. Moreover, we do not use them when building up the mapping between user reviews and code as ground truth since: (1) The reviews provided by Ciurumelea et al. [26] are derived from 39 apps. However, the authors do not provide the APK id of these reviews. Thus, we cannot use them to build up the mapping between user reviews and code. (2)The reviews provided by Maalej et al. [20], [21] are obtained from 4 iOS apps and 4 Android apps. However, all the 4 Android apps (i.e., 80 apps, PicsArt, Pinterest, Whatsapp) are closed-source apps. Thus, we cannot obtain their bug reports or release notes to build up the mapping between function error reviews and code.

Table 7 shows the result of ReviewSolver. For the dataset provided by Ciurumelea et al. [26], our system achieves 84.6% precision and 88.5% recall rate for detecting function error related reviews. For the dataset provided by Maalej et al. [20], [21], our system achieves 88.4% precision and 66.4% recall rate for detecting function error related reviews. We manually analyze the cause of false positives/negatives.

TABLE 7
Result of Classifying Function Error related Reviews

Dataset	Precision	Recall	F-1
Ciurumelea el al. [26]	85.4%	87.4%	86.4%
Maalej et al. [20], [21]	88.3%	66.4%	75.8%

False positives. The major cause of false positive is that although some reviews contain function error related words (e.g., "bug", "problem"), the objects that user really wanted to describe are some fixed bugs, small limitations, or bugs of other apps. For example, "*Amazing This app helped me a lot. Allowed me to see why my apps crashed so I could fix the bugs*". To remove such false positives, we could analyze the tense of the review to identify the fixed bugs (e.g., "... has been fixed") and check the subject related to the bug (e.g., "my apps").

False negatives. The major cause of false negative is that users may describe function errors implicitly. For example, the review "*Slow on tablets In need of a major update. Images not as crisp or bright as on jjComic Viewer or Perfect Viewer.*" does not contain any function error related words (e.g., "bug", "error"), and the user only described that the error makes the tablet "Slow", thus ReviewSolver cannot recognize it. We can add the function error related reviews that describe the error implicitly into the training set to remove such false negatives.

Answer to RQ1: The experimental result shows that: For the review dataset provided by Ciurumelea et al. [26], ReviewSolver can achieve 85.4% precision, 87.4% recall rate for identifying function error related reviews. For another review dataset provided by Maalej et al. [20], [21], ReviewSolver can achieve 88.4% precision, 66.4% recall rate for identifying function error related reviews.

5.3 Performance of ReviewSolver

We use the mapping from user reviews to code with ground truth (described in Section 5.1) to evaluate ReviewSolver and compare it with ChangeAdvisor [13] and Where2Change [14].

Comparison ReviewSolver with ChangeAdvisor and Where2Change by using bug reports. The "#Total Map" column of Table 8 shows the total number of mappings from reviews to code files with ground truth of bug reports. From the "#RS Map" and "#CA Map" columns of the Table 8, we can see that ReviewSolver can identify more mappings than the state-of-the-art system (i.e., ChangeAdvisor). In total, the number of mapping identified by ReviewSolver (i.e., 324) is much more than the number of the mappings found by ChangeAdvisor (i.e., 102) and Where2Change (i.e., 211). For example, for the Cgeo app, ReviewSolver identifies 56 mappings whereas ChangeAdvisor only finds 13 mappings. For the WordPress app, ReviewSolver discovers 74 mappings while ChangeAdvisor only identifies 24 mappings and Where2Change finds 43 mappings. This performance of Where2Change is better than ChangeAdvisor since Where2Change extracts topic words from bug reports and then combines them with the topic words of the review cluster to localize the function error.

TABLE 8
The number of mappings that can be identified by ReviewSolver and ChangeAdvisor. The meaning of each column (from 2-6): total number of manually analyzed function error reviews, the number of mappings identified by using bug reports (column "#Total Map"), the number of mappings identified by ReviewSolver (column "#RS Map"), ChangeAdvisor (column "#CA Map"), and Where2Change (column "#W2C Map")

APK Name	#Error Reviews	#Total Map	#RS Map	#CA Map	#W2C Map
Twidere	247	2874	44	1	3
Signal	204	1387	73	15	65
K-9 Mail	159	591	37	20	18
SeriesGuide	221	1545	19	13	56
WordPress	298	3146	74	24	43
Cgeo	179	1147	56	13	12
OneBusAway	146	428	16	12	4
AntennaPod	82	422	5	4	10
Total	1536	11450	324	102	211

Comparison ReviewSolver with ChangeAdvisor and Where2Change by using release notes. In Table 9, the "#Total Map" column shows the total number of mappings from reviews to code files with ground truth of release notes. From the "#RS Map" and "#CA Map" columns of the Table 9, we can see that ReviewSolver can identify more mappings than the state-of-the-art system (i.e., ChangeAdvisor). In total, the number of mapping identified by ReviewSolver (i.e., 65) is four times as many as the number of the mappings discovered by ChangeAdvisor (i.e., 15). For example, for the K-9 Mail app, ReviewSolver discovers 11 mappings whereas ChangeAdvisor only identifies 5 mappings. For another app WordPress, ReviewSolver finds 15 mappings while ChangeAdvisor only identifies 3 mappings. By comparing the "#RS Map" and "#W2C Map" columns, we also find ReviewSolver discover more mappings (i.e., 65) than Where2Change (i.e., 25).

To determine if the results of ChangeAdvisor and Where2Change can complement the results of the

TABLE 9
The number of mappings that can be identified by ReviewSolver and ChangeAdvisor. The meaning of each column (from 2-6): total number of manually analyzed function error reviews, the number of mappings identified by using release notes (column "#Total Map"), the number of mappings identified by ReviewSolver (column "#RS Map"), ChangeAdvisor (column "#CA Map"), and Where2Change (column "#W2C Map")

APK Name	#Error Reviews	#Total Map	#RS Map	#CA Map	#W2C Map
K-9 Mail	127	164	11	5	11
SeriesGuide	216	284	17	3	3
WordPress	298	446	15	3	3
Cgeo	179	302	15	3	5
OneBusAway	71	49	6	0	0
AntennaPod	77	94	1	1	3
Total	968	1339	65	15	25

ReviewSolver or not, by using the ground truth created by using bug reports and release notes, we measured how many of the results of ChangeAdvisor and Where2Change are also discovered by ReviewSolver. As shown in Table 10, we can find that the most results of ChangeAdvisor and Where2Change are not discovered by ReviewSolver. In other words, both ChangeAdvisor and Where2Change can complement the results of the ReviewSolver. For example, for the review "*I also don't understand why i cannot move emails in trash (deleted in error) back into my inbox!*". After clustering reviews, ChangeAdvisor extracts four words from this review ("delet", "email", "error", "move"). Then, it finds the source code *MessageViewFragment* that also contains these four words. Although ReviewSolver can extract the verb phrase "*move emails*", it cannot identify the classes that implementing this function.

In detail, when checking the mappings of ground truth created by the using bug reports, we found that, for the 102 mappings discovered by ChangeAdvisor, 84 mappings of them cannot be found by ReviewSolver (Row 2, Column "$\overline{\mathrm{RS}} \cap CA$"). Only 18 mappings found by ChangeAdvisor are also found by ReviewSolver. Moreover, for the 211 mappings discovered by Where2Change, 198 of them cannot be found by ReviewSolver (Row 4, Column "$\overline{\mathrm{RS}} \cap W2C$"). When checking the mappings of ground truth created by using the release notes, we also find that, for the 15 mappings found by ChangeAdvisor, 13 of them cannot be found by ReviewSolver (Row 3, , Column "$\overline{\mathrm{RS}} \cap CA$"). Only 2 mappings found by ChangeAdvisor are also found by ReviewSolver. Moreover, for the 25 mappings found by Where2Change, 23 mappings of them cannot be found by ReviewSolver (Row 6, Column "$\overline{\mathrm{RS}} \cap W2C$").

TABLE 10
Result of identifying the distinct mappings between function error reviews and code found by ReviewSolver, ChangeAdvisor, and Where2Change.

	$RS \cap CA$	$RS \cap \overline{\mathrm{CA}}$	$\overline{\mathrm{RS}} \cap CA$
Ground Truth(Bug Report)	18	305	84
Ground Truth(Release Note)	2	63	13
	$\mathrm{RS} \cap W2C$	$RS \cap \overline{\mathrm{W2C}}$	$\overline{\mathrm{RS}} \cap W2C$
Ground Truth(Bug Report)	13	310	198
Ground Truth(Release Note)	2	63	23

Answer to RQ2: The experimental result shows that: Given the same set of function error related reviews, by using the ReviewSolver can correctly resolve more reviews than ChangeAdvisor and Where2Change. Moreover, the results of ChangeAdvisor and Where2Change can complement the results of the ReviewSolver.

5.4 Resolving Function Error related Reviews

As shown in Table 11, for the 4743 function error related reviews discovered by ReviewSolver, 57.9% (i.e., 2745/4743) of these function error related reviews can be mapped to code by ReviewSolver. This number is much larger than that of ChangeAdvisor, which can only map 442 of them to code (i.e., 9.3%, 442/4743). Moreover, for the 8 apps containing bug reports, ReviewSolver can map 1061 of the function error reviews to code while Where2Change only maps 677 of them (i.e., 38.0%, 677/1782). Other 10 apps are not analyzed by Where2Change since they do not contain any bug reports. The above result shows that ReviewSolver outperforms both ChangeAdvisor and Where2Change in the percentage of reviews mapped to code.

TABLE 11
The number of negative reviews resolved by ReviewSolver and ChangeAdvisor. The meaning of the columns (3-6): the number of function error reviews (column "#Error Review"), the number of function error reviews resolved by ReviewSolver (column "#RS"), ChangeAdvisor (column "#CA"), and Where2Change (column "#W2C")

#	APK Name	#Error Review	#RS	#CA	#W2C
1	Twidere	303	191	17	3
2	SMS Backup+	519	324	8	-
3	Signal	214	154	25	121
4	Shortyz Crosswords	367	189	34	-
5	K-9 Mail	319	216	22	156
6	rif is fun for Reddit	229	107	12	-
7	Focal	435	266	12	-
8	FBReader	158	92	17	-
9	SeriesGuide	320	126	11	63
10	WordPress	235	189	32	150
11	Solitaire	122	58	12	-
12	Cool Reader	293	170	17	-
13	Cgeo	127	76	19	55
14	OneBusAway	178	75	132	90
15	AcDisplay	341	178	39	-
16	AntennaPod	86	34	7	39
17	FrostWire	320	174	7	-
18	AnkiDroid	177	126	19	-
	Total	4743	2745	442	677

Since ReviewSolver uses various context information to map review to code, for each context information, we count the number of function error reviews that can be located by using it for the sake of measuring the effectiveness of different context information. The result is shown in Table 12. It shows that, for the 4743 function error reviews, the context information "General Task" can be used to resolve 42.1% of them. Moreover, the context information "App Specific Task" can resolve 28.7% of them (i.e., 1359/4743): 1203 of them are resolved by using the method names set by developers and 578 of them resolved by using the method names predicted by Code2vec [59]. We also find that the context information "Exception" can only be found in 4 function error reviews. The reason is few normal users have technical knowledge of Android apps. They can obtain the type or detail of exception. Thus, only

4 function error reviews describe "Exception". Note that the distribution shown in Table 12 may not be the same as that in Table 1 since some reviews contain multiple kinds of context information and we only consider the primary context information (i.e., the most important information that can help us locate the error) when creating Table 1. For example, for the review "*After updating the app, I cannot connect server*", we regard "connect server" as the primary context information.

TABLE 12
Number of function error reviews that can be mapped to code by ReviewSolver through different context information.

Context	#Function Error	Percentage
General Task	1998	42.1%
App Specific Task	1359	28.7%
API/URI/intent	852	18.0%
Updating App	431	9.1%
Registering Account	168	3.5%
Error Message	128	2.7%
GUI	82	1.7%
Opening App	54	1.1%
Exception	4	0.08%

To check the precision of the mapping from reviews to code identified by ReviewSolver, we manually check 50 mappings for each app. The result is shown in the third and fourth columns of Table 13, and we can see that ReviewSolver can achieve 70.0% precision.

Cause of false mappings. The major cause of the false mappings is that some reviews do not contain context information, but we still map them in code. For instance, consider the review "*This app has started crashing more than a 737 airplane*". This review does not describe when the crash appears. But our system still extracts the phrase "started crashing" and then maps it to classes. To remove such false mappings, we need to build up a machine learning classifier to determine if the review contains context information or not. Moreover, some false mappings are generated since ReviewSolver cannot correctly identify the function error review. For example, consider the review "*A nice and clean lockscreen with a cool unlock animation. Only problem I have with this is that there is no PIN feature*". Although this review mentions "problem", it is intended to ask the developers to add the "PIN feature". But the machine learning classifier of ReviewSolver incorrectly classifies it as a function error review and then generate useless mappings between this review and code. To remove such false mappings, we need to increase the dataset used for training the machine learning classifier of identifying function error reviews.

To show that the chances of overfitting is very low in ReviewSolver, we conduct an additional experiment by using 10 new apps. For each app, we use ReviewSolver and ChangeAdvisor to localize the function errors mentioned in user reviews. The final result is shown in Table 14. For the 523 function error related reviews identified by ReviewSolver, ReviewSolver can localize 248 of them in code while ChangeAdvisor only localizes 97 of them. It also shows that ReviewSolver outperforms ChangeAdvisor in the percentage of localizing function error related reviews. Note, since we cannot obtain the bug reports dataset of these 10 new apps, we do not process them with Where2Change.

Answer to RQ3: The experimental result shows that: ReviewSolver can resolve 57.9% function error related reviews.

TABLE 13
Correctness of the mappings from reviews to code found by ReviewSolver

#	APK Name	ReviewSolver #Correct/Check	ReviewSolver Precision
1	Twidere	34/50	68.0%
2	SMS Backup+	40/50	80.0%
3	Signal	33/50	66.0%
4	Shortyz Crosswords	22/38	57.9%
5	K-9 Mail	40/50	80.0%
6	rif is fun for Reddit	32/50	64.0%
7	Focal	40/50	80.0%
8	FBReader	19/50	38.0%
9	SeriesGuide	31/50	62.0%
10	WordPress	40/50	80.0%
11	Solitaire	38/50	76.0%
12	Cool Reader	35/50	70.0%
13	Cgeo	40/50	80.0%
14	OneBusAway	32/39	82.1%
15	AcDisplay	35/50	70.0%
16	AntennaPod	16/29	55.2%
17	FrostWire	37/50	74.0%
18	AnkiDroid	35/50	70.0%
	Total	599/856	70.0%

TABLE 14
Additional dataset: The number of function error reviews resolved by ReviewSolver and ChangeAdvisor. The meaning of the columns (3-5): the number of function error reviews (column "#Error Review"), the number of function error reviews resolved by ReviewSolver (column "#RS"), ChangeAdvisor (column "#CA")

#	APK Name	#Error Review	#RS	#CA
19	Password Store	12	4	1
20	IRCCloud	82	42	7
21	Quasseldroid IRC	22	14	0
22	primitive ftpd	15	7	10
23	Seafile	67	49	16
24	ML Manager	13	4	1
25	CycleStreets	79	43	11
26	Hangar	65	26	6
27	qBittorrent	52	30	39
28	MozStumbler	55	29	6
	Total	462	248	97

6 Discussion

In this section, we discuss running time, the realism of applicability, usability, and limitation of ReviewSolver.

6.1 Time consumption

For each function error review, we measured the average time consumption of using different types of context information to localize the function error. The result is shown in Table 15.

Localizing the "API/URI/intent" related error has the largest time overhead. The reason is ReviewSolver will compare each phrase extracted from the function error review with the descriptions of 26,030 Android framework APIs, which is a time-consuming procedure. In the future, to speed up this procedure, we will select the most commonly invoked Android framework APIs for comparison (i.e., ignore the rarely called APIs). Moreover, two kinds of context information (i.e., "App Specific Task" and "General Task") also consume more than one second for each review. To localize the "App Specific Task" related error, the average time overhead reaches 12.2 seconds per review. This is

because ReviewSolver will compare each phrase of the review with all the method names extracted from the app. To speed up this procedure, we will filter some method names that do not describe tasks implemented in code (e.g., "test") in future. To localize the "General Task" related errors, the average time overhead is about 5.0 seconds per review since ReviewSolver requires comparing each phrase of the review with all the Q&As in Stack Overflow. We can also select the most popular Q&As in Stack Overflow to reduce the time overhead. For the remaining types of context information, the average time overhead is less than 1.0 second for each review, which means each function error review can be processed quickly.

We also measured the time overhead of running ChangeAdvisor and Where2Change, the final result shows that ChangeAdvisor consumes about 5.5 seconds per review and Where2Change consumes about 2.1 seconds. These two systems are faster than ReviewSolver since their analysis are based on review clusters. They do not compare each review with the source code and they only extract words from source code (i.e., they do not extract precise context information from source code). Where2Change also implemented multithread (i.e., process the reviews of multiple apps at the same time) to reduce time overhead.

TABLE 15
Average time consumption of using different context information to localize the function error.

Context	Average Time (seconds/review)
General Task	5.0
App Specific Task	12.2
API/URI/intent	93.4
Updating App	$7.3 * 10^{-5}$
Registering Account	$6.8 * 10^{-4}$
Error Message	$1.7 * 10^{-1}$
GUI	$3.9 * 10^{-3}$
Opening App	$7.3 * 10^{-4}$
Exception	$2.7 * 10^{-4}$

6.2 Scalability

The scalability of ReviewSolver is better than existing tools (i.e., ChangeAdvisor and Where2Change). When applying ChangeAdvisor and Where2Change to process a large number of apps, two major challenges should be solved. First, since these two systems require source code of apps, if the analyzed apps are closed-source apps, we must contact the developers of the apps to obtain the source code. Second, these two systems extract topic words from review clusters, thus their performance will be affected by the result of review clustering. If the new apps only include a small number of function error reviews, they could not divide these reviews into clusters and it will be difficult to extract topic words from them. Moreover, Where2Change also requires using the bug reports as the input. If the new apps do not have any bug reports, Where2Change cannot leverage bug reports to precisely localize the function errors of user reviews. As ReviewSolver only uses the APK files and user reviews as input, it can also process the new closed-source apps. Moreover, ReviewSolver identifies and processes the function error reviews one by one, thus it will not be affected by the number of function error reviews and the result of review clustering. Moreover, when applying RISING to process a large number of new apps, it also needs to solve the challenges of obtaining source code, review clustering, and commit messages. But ReviewSolver do not have such issues.

6.3 Applicability

To measure the realism of applicability of ReviewSolver, we discuss whether ReviewSolver can be used in localizing function errors of ecosystems other than Android. Although we mainly describe how to localize the function error reviews of Android apps in this paper, we can easily customize ReviewSolver to process other ecosystems (e.g., iOS apps installed in iPhone) through modifying the "Static Analysis" module (Section 3.3). This is because other ecosystems require using different static analysis tools to perform static analysis and then extract context information from bytecode. Other two major components of ReviewSolver (i.e., "Review Analysis" in Section 3.2 and "Localizing Function Errors" in Section 4) do not need to be modified. The reason is users use similar natural language sentences to describe the function errors in user reviews. We can follow the same procedure to correlate the context information extracted from reviews and code.

To show that ReviewSolver can also be applied in processing other ecosystems, we select 5 iOS apps as the example. For the context information contained in function error reviews in Table 1, we use static analysis tools to extract them from iOS apps. Different from Android apps that are developed through Java and many tools can be used to perform static analysis (e.g., Soot [86], Flowdroid [60], Apktool [87]), developers usually use Swift and Objective-C to develop iOS apps. Performing static analysis on iOS apps is more challenging and few tools are available [88], [89]. For instance, recent research on iOS apps only extracts the invoked framework APIs [90]. Finally, we select Class-dump [91] to extract three types of context information from the iOS apps:
(1)"App Specific Task". By using the output of Class-dump [91], we extract the class names and the method names defined by developers. Then, we can use them to localize the function errors that appeared during performing app specific tasks (Section 4.1.1).
(2)"GUI". For the classes implementing the callbacks of GUI, we extract all the object names whose types are GUI components (e.g., UIButton [92]). We use the object names to match the UI components mentioned in user reviews (Section 4.1.2).
(3)"API/URI/intent". To localize the function errors that appeared when invoking Framework APIs (Section 4.2.1), for each method defined by developers, we parse the result of Class-dump to extract the invoked framework APIs. To precisely identify these framework APIs, we write a crawler and obtain 6,086 iOS framework APIs from the official website of iOS [93].

Finally, based on these three types of extracted context information, for the 1121 function error reviews of 5 iOS apps identified by ReviewSolver, we can localize 366 of them (i.e., 366/1121, 32.6%) to at least one class defined by developers (Table 16). This result demonstrates that ReviewSolver can be easily customized to localize the function errors of other ecosystems. Due to the time limit, we do not extract other types of context information from

iOS apps. In future, we will extract them to improve the performance of processing user reviews of iOS apps.

TABLE 16
Result of localizing the function error reviews of iOS apps through extracting three types of context information from the apps.

iOS App	# Error Reviews	# RS Map
Nextcloud	80	20
WordPress	403	157
Signal	304	123
Wire	156	30
DuckDuckGo	178	36
Total	1121	366

6.4 Usability

In this section, we compare the usability of `ReviewSolver` with existing systems [13]–[15].

When localizing the function error related user reviews of Android apps, `ReviewSolver` only requires using the APK files downloaded from the app markets as the input. It means that `ReviewSolver` can handle both open-source and closed-source apps. `ChangeAdvisor` [13], `Where2Change` [14], and `RISING` [15] focus on open-source apps and they will compare the words extracted from user reviews and source code to localize the error. Moreover, `Where2Change` integrates bug reports and `RISING` uses commit messages to achieve better performance. From this point, `ReviewSolver` can handle more apps when compared with existing systems [13]–[15].

Compared with existing systems [13], [14], when running `ReviewSolver`, the users need to wait a longer time before obtaining the final result. The main reason is, apart from the apps' bytecode, `ReviewSolver` also use the Q&A of Stack Overflow, official API documents, and class/method names of the apps to localize the function error. We will design some mechanisms to decrease the time overhead of `ReviewSolver` in future.

6.5 Findings

Other researchers can benefit from the following findings of this paper:

(1) **Context information contained in function error related reviews**. Previous researchers [22], [51] only summarize different types of topics described in user reviews, including the bug/function error related reviews. None of them performs fine-grained analysis on function error related reviews and evaluates different types of context information described in them. In this paper, we are the *first* to summarize various types of context information contained in function error related user reviews. Other researchers can use the context information summarized in this paper to design some new systems (e.g., recommend code changes for developers).

(2) **New framework design for localizing function errors of reviews**. We design a novel framework that combines NLP and program analysis to correlate the reviews and bytecode through their semantic meanings and then localize the function errors. Other researchers can use this framework design when developing other similar systems. Compared with other existing systems, our framework has two significant differences: (a) Although a few other studies also localize the function errors in user reviews [13]–[15], they do not extract precise semantic information from the user reviews and apps' bytecode. (b) We observe that it is possible to localize the function errors in user reviews without source code. Previous papers rely on source code of apps [13]–[15].

(3) **New techniques for bridging the gap between the text and code**. We design a series of methods to bridge the gap between natural language and bytecode. For each type of context information contained in user reviews, we design a specified method to localize it according to the semantic information extracted from apps' bytecode. Other researchers can employ these methods to correlate other kinds of software artefacts (e.g., bug reports, commit messages, description) with the corresponding code.

6.6 Limitation of `ReviewSolver`

Some factors may affect the performance of `ReviewSolver`. When identifying the function error related reviews, the training dataset does not cover all kinds of reviews that describe the error implicitly (e.g., "... is hard to load"). To overcome this limitation, we will try to identify them through deep learning methods [94], [95]. When mapping function error related reviews to code, `ReviewSolver` cannot filter out all useless phrases that can be mapped to code mistakenly. To address this issue, we will try using machine learning classifier to identify useless phrases. Some function error related reviews cannot be located since they are related to the compatibility issues of specified device. We can use information retrieval technique to recognize the types of devices and report them to developer automatically.

When measuring the performance of localizing function error reviews, the ground truth (i.e., mapping between function error reviews and the corresponding code) cannot be obtained from the internet. Currently, we tried to manually construct the ground truth by using two kinds of documents (i.e., bug reports and release notes). In the future, we will use other methods (e.g., using the commit messages [15], [96]) to construct the ground truth.

7 Related Work

7.1 Review Analysis

A number of studies have been conducted on the user reviews of app store [2]. However, the majority of them just analyze user reviews without correlating them with apps' code. Chen et al. [97] combine static features (e.g., app name, category) and dynamic features (e.g., current rate count, description) with comment features (e.g., user rate, comment title) to predict the popularity of apps. Khalid et al. manually analyze user reviews and uncover 12 types of user complaints [22]. To identify correlations between error-sensitive permissions and error-related reviews, Gomez et al. [98] leverage LDA and J48 to process the permissions and reviews.

Some other studies extract app features from reviews. Iacob et al. [7] define a set of linguistic rules to match feature request related reviews, and then use LDA to identify common topics in these reviews. AR-Miner [12] employs machine learning technique to filter out non-informative reviews and then performs clustering on the remaining reviews to provide an intuitive summary for developers. Ciurumelea et al. manually analyze user reviews and define

a high level taxonomy (e.g., compatibility, usage) and low level taxonomy (e.g., device, UI) [26] and apply machine learning techniques to classify the reviews. AutoReb [99] combines machine learning and crowdsourcing technique to identify four kinds of privacy related reviews, including spamming, financial issue, over-privileged permission, and data leakage. To identify the part of the app loved by users, SUR-Miner [6] extracts the semantic dependence relation between words and utilizes clustering algorithms to identify users' opinion towards corresponding aspect. In [34], Walid Maalej et al. combine text classification, NLP, and sentiment analysis to classify reviews into four categories. To generate summaries of users feedback, SURF [100] classifies review sentences into different categories by utilizing the intentions and topics of the reviews. Panichella et al. [101] exploit machine learning and deep learning for automatic classification of requirements from elicitation sessions and user feedback.

`ChangeAdvisor` [13] and `Where2Change` [14] are most closely related study since they also analyze code. `ChangeAdvisor` [13] employs the HDP algorithm [102] to extract topic words from the clusters of function error related reviews. Then it calculates the asymmetric dice similarity coefficient [103] between these topic words and the words extracted from source code file. If the result reaches a threshold, it recommends the developer to check the corresponding code file. The major differences between our system and `ChangeAdvisor` include: 1) When analyzing the reviews, `ChangeAdvisor` does not consider the part-of-speech tags of each word, which will cause false mappings. `ReviewSolver` conduct syntactic analysis on each review to avoid such problem. 2) We employ static analysis to extract the starting activity, requested permissions, APIs/URIs/intents, error messages, class/method names, visible information and invisible information of GUI from APK. We do not simply split the code into distinct words like [26] [13] to avoid including many useless words, which will affect the correctness of mapping reviews to code. 3) When mapping the review to code, `ChangeAdvisor` [13] checks the number of words shared by review and code file. It does not consider the synonyms, thus leading to many false negatives. We leverage the word embedding method to measure semantic similarity between the two phrases, and our method can find similar phrases even when some words are different. `Where2Change` [14] employs bug reports to improve the performance of mapping reviews to code. After clustering user reviews to clusters, `Where2Change` map review clusters to bug reports by using Word2vec. Then, `Where2Change` combines the topic words of review clusters with the words contained in bug reports. The resultant enriched text is used to identify the source code classes that should be changed. Similar to `ChangeAdvisor`, `Where2Change` also fails to consider the part-of-speech tags of each word in the reviews and the context information contained in code, which may cause false positives and false negatives.

`RISING` [15] adopts existing system `ARDOC` [104] to transform user reviews into individual sentences and then identify informative sentences. Then, `RISING` groups the informative sentences into fine-grained clusters. Finally, to localize the file changes, it computes the similarity between review clusters and the words extracted from source code and commit messages. The main differences between our system and `RISING` include: 1) `RISING` extracts file paths, class summary, method summary, method name, and field declaration from source code. Similar to `ChangeAdvisor`, it does not extract the starting activity, requested permissions, APIs/URIs/intents, error messages, class/method names, visible information and invisible information of GUI. Thus, its correctness will be affected when mapping reviews to code. Our system performs static analysis to extract these different kinds of information from APK file. 2) `RISING` does not analyze the part-of-speech tag of each word in the review. For the same word (e.g., "contact"), it cannot determine if it is a verb (e.g., "contact us") or a noun (e.g., "use contact information"), which may cause false positives when mapping the review to code. Our system avoids this problem by extracting verb phrases and noun phrases from the parse tree and typed dependency of the sentence. 3) When mapping the user reviews to app code, `RISING` includes the commit messages to assist file localization. However, only open-source apps have the commit messages. The performance of `RISING` will be affected when processing the closed-source apps, which only provide APK file. Our system does not rely on source code or commit messages. The performance will not be affected when processing the closed-source apps. In summary, when applying `RISING` to process a large number of new apps, it also needs to solve the challenges of obtaining source code, review clustering, and commit messages. But `ReviewSolver` do not have such issues.

To localize the crashes in user reviews, Becloma [105] adopts MONKEY [106]/SAPIENZ [107] to run apps dynamically and trigger crashes. Then, it calculates the similarity between the words in stack traces and the words in reviews. The main limitation is that they can only localize the function errors triggered through dynamic analysis. Otherwise, the function errors cannot be localized. Our system does not require dynamically run the app to trigger the function error (i.e., only static analysis is required).

7.2 Code Analysis

Many static analysis systems have been proposed to analyze the APK file of mobile apps [108]. EdgeMiner [109] conducts static analysis on Android framework to identify callbacks and their corresponding registration functions. Since developers can use obfuscation technique to hide the class, method, variable names, DeGuard [110] proposes to build up probabilistic model for third-party libraries by analyzing non-obfuscated apps. Then it employs the probabilistic model to recover the obfuscated class, method, and variable names of new APKs. If the developer use packing services to hide the dex files, DexHunter [111] and PackerGrind [112] can be used to recover the original dex files. FlowDroid [60] performs static taint analysis to identify the source to sink path. To analyze the inter component communication of apps, IccTA leverages IC3 (an advanced string analysis tool) to discover the ICC links and create dummy method for them [57]. To detect piggybacked apps, Fan et al. propose using sensitive subgraphs to profile the app and extract features from them [113], [114]. Xue et al. [115] use dynamic analysis to identify the factors that will affect the network measurement result of mobile apps.

Some studies use static analysis to analyze the GUI of apps. To generate precise privacy policies, AutoPPG [116], [117] leverages Vulhunter [56] to analyze the callbacks of GUI and the conditions of sensitive behaviors. To find

the contradiction between user interface and code, AsDroid [118] compares the behavior found by static analysis with the behavior identified from UI to find contradiction. To identify the sensitive user input, UIPicker [119] determines sensitive input fields by using a supervised learning classifier that is based on the features extracted from the texts of the UI elements.

7.3 Linking Document to Code

Information retrieval (IR) [103] technique has been used to link document to code. The document is used as a query and the code is regarded as document collection. IR uses some models (e.g., vector space model, probabilistic model) to calculate the relevance between the input document and code files. Then it ranks the relevance of code files. BLUIR [120] extracts words from the class names, method name, and variable names of source code and then it employs VSM to link bug reports to code. TRASE [121] builds up probabilistic topic model for software artifacts. This model can be used to classify artifacts based on semantic meaning and visualize the software with topic words. CRISTAL [4] compares crowd reviews with code changes to measure the extent to which the crowd request have been accommodated. The system also monitors changes of user ratings to measure user reactions. To discover the inconsistency between app description and permissions, TAPVerifier [122] uses NLP to analyze the privacy policy and uses static analysis to analyze the code. To locate feature related code, SNIAFL [123] first transforms the feature description and method/variable names in code into index terms. Then it uses vector space model to calculate the cosine similarity between the feature description and methods in code. PPChecker [124] compares the privacy policy with APK file to detect three kinds of problems contained in privacy policy. To enrich the content of new bug reports and facilitate software maintenance, Zhang et al. [125] propose to utilize sentence ranking to select proper sentences from historical bug reports.

8 Conclusion

User reviews of mobile apps can help developers discover the function errors uncaught by app testing. Manually processing reviews is time-consuming and error-prone whereas the state-of-the-art automated approaches may lead to many false positives and false negatives. In this paper, we propose and develop a novel tool named `ReviewSolver` to automatically localize the function error by correlating the context information extracted from reviews and the bytecode. The experimental result shows that `ReviewSolver` can identify the function error reviews with a high precision and recall rate. Moreover, it locates much more function error related code than the state-of-the-art tools `ChangeAdvisor` and `Where2Change`.

Acknowledgment

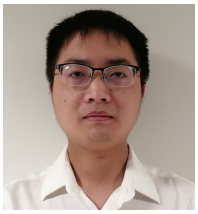
The authors thank the reviewers for their quality reviews and suggestions. This work was supported in part by the Hong Kong RGC Projects (PolyU15223918, PolyU15224121), HKPolyU Start-up Fund (BD7H), National Natural Science Foundation of China under Grant (61972359,61831022), Zhejiang Provincial Natural Science Foundation of China under Grant LY19F020052, and National Science Foundation under the grant CCF-2046953.

References

[1] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A survey of app store analysis for software engineering," *IEEE trans. TSE*, 2016.
[2] N. Genc-Nayebi and A. Abran, "A systematic literature review: Opinion mining studies from mobile app store user reviews," *Journal of Systems and Software*, 2017.
[3] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *Proc. RE*, 2013.
[4] F. Palomba, M. Linares-Vásquez, G. Bavota, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "User reviews matter! tracking crowdsourced reviews to support evolution of successful apps," in *Proc. ICSME*, 2015.
[5] B. Fu, J. Lin, L. Li, C. Faloutsos, J. Hong, and N. Sadeh, "Why people hate your app: Making sense of user feedback in a mobile app store," in *Proc. KDD*, 2013.
[6] X. Gu and S. Kim, "What parts of your apps are loved by users?" in *Proc. ASE*, 2015.
[7] C. Iacob and R. Harrison, "Retrieving and analyzing mobile apps feature requests from online reviews," in *Proc. MSR*, 2013.
[8] L. V. G. Carreño and K. Winbladh, "Analysis of user comments: an approach for software requirements evolution," in *Proc. ICSE*, 2013.
[9] E. Guzman and W. Maalej, "How do users like this feature? a fine grained sentiment analysis of app reviews," in *Proc. RE*, 2014.
[10] P. M. Vu, T. T. Nguyen, H. V. Pham, and T. T. Nguyen, "Mining user opinions in mobile app reviews: A keyword-based approach (t)," in *Proc. ASE*, 2015.
[11] D. H. Park, M. Liu, C. Zhai, and H. Wang, "Leveraging user reviews to improve accuracy for mobile app retrieval," in *Proc. SIGIR*, 2015.
[12] N. Chen, J. Lin, S. C. Hoi, X. Xiao, and B. Zhang, "Ar-miner: mining informative reviews for developers from mobile app marketplace," in *Proc. ICSE*, 2014.
[13] F. Palomba, P. Salza, A. Ciurumelea, S. Panichella, H. Gall, F. Ferrucci, and A. De Lucia, "Recommending and localizing change requests for mobile apps based on user reviews," in *Proc. ICSE*, 2017.
[14] T. Zhang, J. Chen, X. Zhan, X. Luo, D. Lo, and H. Jiang, "Where2change: Change request localization for app reviews," *IEEE Trans. TSE*, 2019.
[15] Y. Zhou, Y. Su, T. Chen, Z. Huang, H. C. Gall, and S. Panichella, "User review-based change file localization for mobile applications," *IEEE Trans. TSE*, 2020.
[16] G. Grano, A. Ciurumelea, S. Panichella, F. Palomba, and H. C. Gall, "Exploring the integration of user feedback in automated testing of android applications," in *Proc. SANER*, 2018.
[17] DAVID CURRY, "Android statistics (2021)," https://www.businessofapps.com/data/android-statistics/, 2021.
[18] App Brain, "Number of android apps on google play," https://www.appbrain.com/stats/number-of-android-apps, 2021.
[19] L. Yu, J. Chen, H. Zhou, X. Luo, and K. Liu, "Localizing function errors in mobile apps with user reviews," in *Proc. DSN*, 2018.
[20] W. Maalej, Z. Kurtanović, H. Nabil, and C. Stanik, "On the automatic classification of app reviews," *Requirements Engineering*, 2016.
[21] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? on automatically classifying app reviews," in *Proc. RE*, 2015.
[22] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, 2015.
[23] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? on automatically classifying app reviews," in *Proc. RE*, 2015.
[24] "Stack overflow," https://stackoverflow.com/, 2019.
[25] "Csdn," https://www.csdn.net/, 2019.
[26] A. Ciurumelea, A. Schaufelbühl, S. Panichella, and H. Gall, "Analyzing reviews and code of mobile apps for better release planning," in *Proc. SANER*, 2017.
[27] Subkhan Sarif, "Link inside webpage always return "404 not found" error if opened from webview ∼android ∼java," https://shorturl.at/mJXYZ, 2020.
[28] weakwire, "How can i check from android webview if a page is a "404 page not found"?" https://shorturl.at/mtxMR, 2020.
[29] Android Developers, "Socket," https://shorturl.at/djsJO, 2020.
[30] S. Bird, E. Loper, and E. Klein, "Natural language toolkit," http://www.nltk.org/, 2017.

[31] M. Gilleland, "Levenshtein distance," https://goo.gl/TVA9Ga, 2017.
[32] M.-C. De Marneffe, B. MacCartney, C. D. Manning *et al.*, "Generating typed dependency parses from phrase structure parses," in *Proc. LREC*, 2006.
[33] M.-C. De Marneffe and C. D. Manning, "Stanford typed dependencies manual," Stanford NLP Group, Tech. Rep., 2008.
[34] W. Maalej, Z. Kurtanović, H. Nabil, and C. Stanik, "On the automatic classification of app reviews," *Requirements Engineering*, 2016.
[35] W. B. Cavnar, J. M. Trenkle *et al.*, "N-gram-based text categorization," *Ann Arbor MI*, 1994.
[36] Z. Wei, D. Miao, J.-H. Chauchat, R. Zhao, and W. Li, "N-grams based feature selection and text representation for chinese text classification," *International Journal of Computational Intelligence Systems*, 2009.
[37] S. N. Group, "Tf-idf weighting," https://goo.gl/RrWHVc, 2009.
[38] A. . M. Text Mining, "What are n-grams?" https://goo.gl/bwAHtp, 2017.
[39] J. Elith, J. R. Leathwick, and T. Hastie, "A working guide to boosted regression trees," *Journal of Animal Ecology*, 2008.
[40] Bhuvan Sharma, "What are the advantages/disadvantages of using gradient boosting over random forests?" https://goo.gl/N6y2Kw, 2015.
[41] B. Liu, "Sentiment analysis and opinion mining," *Synthesis lectures on human language technologies*, 2012.
[42] R. K. Bakshi, N. Kaur, R. Kaur, and G. Kaur, "Opinion mining and sentiment analysis," in *Proc. INDIACom*, 2016.
[43] R. Jongeling, S. Datta, and A. Serebrenik, "Choosing your weapons: On sentiment analysis tools for software engineering research," in *Proc. ICSME*, 2015.
[44] Yong Zhun Hung, "Python 3 wrapper for sentistrength," https://github.com/zhunhung/Python-SentiStrength, 2021.
[45] "Sentistrength," http://sentistrength.wlv.ac.uk/, 2021.
[46] D. Devarajan, "Retirement of alchemyapi service," https://www.ibm.com/cloud/blog/announcements/bye-bye-alchemyapi, 2017.
[47] Stanford NLP Group, "Stanford corenlp," https://github.com/stanfordnlp/CoreNLP, 2020.
[48] "Python 3 wrapper for sentistrength," http://text-processing.com/docs/sentiment.html, 2021.
[49] Gutte Dnyanoba, "Adversative coordinating conjunctions," http://www.dhgutte.com/adversative-coordinating-conjunction/, 2019.
[50] Archana Singh, "Conjunction," https://shorturl.at/bdEU7, 2018.
[51] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *Proc. ICSME*, 2015.
[52] F-Droid Limited and Contributors, "F-droid," https://f-droid.org/, 2020.
[53] GitHub, Inc., "Github," https://github.com/, 2020.
[54] apkmonk.com, "apkmonk: One stop for all android apps," https://www.apkmonk.com/, 2020.
[55] APKPure.com, "Apkpure," https://apkpure.com/, 2020.
[56] C. Qian, X. Luo, Y. Le, and G. Gu, "Vulhunter: toward discovering vulnerabilities in android applications," *IEEE Micro*, 2015.
[57] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "Iccta: Detecting inter-component privacy leaks in android apps," in *Proc. ICSE*, 2015.
[58] guardsquare, "Proguard: Open source optimizer for java and kotlin," https://www.guardsquare.com/en/products/proguard, 2020.
[59] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proc. POPL*, 2019.
[60] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, 2014.
[61] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proc. CCS*, 2012.
[62] stackoverflow, "Best practice for displaying error messages," https://goo.gl/odr84X, 2013.
[63] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Trans. TSE*, 2016.
[64] A. Continella, Y. Fratantonio, M. Lindorfer, A. Puccetti, A. Zand, C. Kruegel, and G. Vigna, "Obfuscation-resilient privacy leak detection for mobile apps through differential analysis," in *Proc. NDSS*, 2017.
[65] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Java 14m model," https://s3.amazonaws.com/code2vec/model/java14m_model.tar.gz, 2020.
[66] Dacong Yan, "Gator: Program analysis toolkit for android," http://web.cse.ohio-state.edu/presto/software/gator/, 2020.
[67] A. Rountev and D. Yan, "Static reference analysis for gui objects in android software," in *Proc. CGO*, 2014.
[68] ribot.co.uk, "Project guidelines," https://github.com/ribot/android-guidelines/blob/master/project_and_code_guidelines.md, 2020.
[69] Faysal Ahmed, "Android naming convention," https://stackoverflow.com/questions/12870537/android-naming-convention/34428819, 2020.
[70] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *arXiv preprint arXiv:1402.3722*, 2014.
[71] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.
[72] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations." in *hlt-Naacl*, 2013.
[73] Google Code, "word2vec," https://goo.gl/NBJgk1, 2017.
[74] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proc. CCS*, 2014.
[75] A. Di Sorbo, C. A. Visaggio, M. Di Penta, G. Canfora, and S. Panichella, "An nlp-based tool for software artifacts analysis," in *Proc. ICSME*, 2021.
[76] L. Yu, X. Luo, C. Qian, and S. Wang, "Revisiting the description-to-behavior fidelity in android applications," in *Proc. SANER*, 2016.
[77] "Android developers: Manifest.permission," https://goo.gl/vWoIU, 2017.
[78] "Android developer: Common intents," https://goo.gl/gHv9sF, 2017.
[79] Ashutosh KS, "Top 10 sites to ask all your programming questions," https://www.hongkiat.com/blog/programming-questions-websites/, 2017.
[80] Stack Exchange Data Dump, "stackexchange," https://archive.org/details/stackexchange, 2020.
[81] Chris Thunes, "javalang 0.13.0," https://pypi.org/project/javalang/, 2020.
[82] A. Rohatgi, A. Hamou-Lhadj, and J. Rilling, "An approach for mapping features to code based on static and dynamic analysis," in *Proc. ICPC*, 2008.
[83] "Homepage for the changeadvisor project," https://sites.google.com/site/changeadvisormobile/, 2021.
[84] "Repository for where2change," https://github.com/Jiachi-Chen/ReviewBugLocalization, 2021.
[85] Amy Nordrum, "The strange art of writing app release notes," shorturl.at/qrPX2, 2017.
[86] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *Proc. CASCON*, 1999.
[87] iBotPeaches, "A tool for reverse engineering android apk files," https://github.com/iBotPeaches/Apktool, 2022.
[88] J. Damian, "Basic ios mobile app reverse engineering," https://www.nowsecure.com/blog/2021/09/08/basics-of-reverse-engineering-ios-mobile-apps/, 2021.
[89] cpholguera, "ios tampering and reverse engineering," https://github.com/OWASP/owasp-mstg/blob/master/Document/0x06c-Reverse-Engineering-and-Tampering.md, 2022.
[90] S. Zimmeck, R. Goldstein, and D. Baraka, "Privacyflash pro: automating privacy policy generation for mobile apps," in *Proc. NDSS*, 2021.
[91] "Class-dump: Generate objective-c headers from mach-o files." https://github.com/nygard/class-dump, 2018.
[92] "Uibutton." https://developer.apple.com/documentation/uikit/uibutton, 2021.
[93] "Apple developer documentation." https://developer.apple.com/documentation/technologies, 2021.
[94] K. Kowsari, D. E. Brown, M. Heidarysafa, K. J. Meimandi, M. S. Gerber, and L. E. Barnes, "Hdltex: Hierarchical deep learning for text classification," in *Proc. ICMLA*, 2017.
[95] S. Minaee, N. Kalchbrenner, E. Cambria, N. Nikzad, M. Chenaghlu, and J. Gao, "Deep learning–based text classification: A comprehensive review," *ACM CSUR*, 2021.

[96] S. Liu, C. Gao, S. Chen, N. L. Yiu, and Y. Liu, "Atom: Commit message generation based on abstract syntax tree and hybrid ranking," *IEEE Trans. TSE*, 2020.
[97] M. Chen and X. Liu, "Predicting popularity of online distributed applications: itunes app store case analysis," in *Proc. iConference*, 2011.
[98] M. Gómez, R. Rouvoy, M. Monperrus, and L. Seinturier, "A recommender system of buggy app checkers for app store moderators," in *Proc. MOBILESoft*, 2015.
[99] D. Kong, L. Cen, and H. Jin, "Autoreb: Automatically understanding the review-to-behavior fidelity in android applications," in *Proc. CCS*, 2015.
[100] 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 *Proc. FSE*, 2016.
[101] S. Panichella and M. Ruiz, "Requirements-collector: automating requirements specification from elicitation sessions and user feedback," in *Proc. RE*, 2020.
[102] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei, "Sharing clusters among related groups: Hierarchical dirichlet processes," in *Advances in neural information processing systems*, 2005.
[103] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.
[104] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "Ardoc: App reviews development oriented classifier," in *Proc. FSE*, 2016.
[105] L. Pelloni, G. Grano, A. Ciurumelea, S. Panichella, F. Palomba, and H. C. Gall, "Becloma: Augmenting stack traces with user review information," in *Proc. SANER*, 2018.
[106] Google Developers, "Ui/application exerciser monkey," https://developer.android.com/studio/test/monkey, 2020.
[107] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proc. ISSTA*, 2016.
[108] A. Sadeghi, H. Bagheri, J. Garcia, and S. Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Trans. TSE*, 2017.
[109] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "Edgeminer: Automatically detecting implicit control flow transitions through the android framework." in *Proc. NDSS*, 2015.
[110] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of android applications," in *Proc. CCS*, 2016.
[111] Y. Zhang, X. Luo, and H. Yin, "DexHunter: toward extracting hidden code from packed Android applications," in *Proc. ESORICS*, 2015.
[112] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proc. ICSE*, 2017.
[113] M. Fan, J. Liu, W. Wang, H. Li, Z. Tian, and T. Liu, "Dapasa: detecting android piggybacked apps through sensitive subgraph analysis," *IEEE Trans. TIFS*, 2017.
[114] M. Fan, J. Liu, X. Luo, K. Chen, Z. Tian, Q. Zheng, and T. Liu, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Trans. TIFS*, 2018.
[115] L. Xue, X. Ma, X. Luo, L. Yu, S. Wang, and T. Chen, "Is what you measure what you expect? factors affecting smartphone-based mobile network measurement," in *Proc. INFOCOM*, 2017.
[116] L. Yu, T. Zhang, X. Luo, and L. Xue, "Autoppg: Towards automatic generation of privacy policy for android applications," in *Proc. SPSM*, 2015.
[117] L. Yu, T. Zhang, X. Luo, L. Xue, and H. Chang, "Toward automatically generating privacy policy for android apps," *IEEE Trans. TIFS*, 2017.
[118] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "Asdroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proc. ICSE*, 2014.
[119] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang, "Uipicker: User-input privacy identification in mobile applications." in *Proc. USENIX Security*, 2015.
[120] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. ASE*, 2013.
[121] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proc. ICSE*, 2010.
[122] L. Yu, X. Luo, C. Qian, S. Wang, and H. K. Leung, "Enhancing the description-to-behavior fidelity in android apps with privacy policy," *IEEE Trans. TSE*, 2017.
[123] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a static noninteractive approach to feature location," *Trans. TOSEM*, 2006.
[124] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps?" in *Proc. DSN*, 2016.
[125] T. Zhang, J. Chen, H. Jiang, X. Luo, and X. Xia, "Bug report enrichment with application of automated fixer recommendation," in *Proc. ICPC*, 2017.



Le Yu received the Ph.D degree in computer science from the Hong Kong Polytechnic University, under the supervision of Dr.Xiapu Luo. Currently, he is a research assistant professor with the Department of Computing, Hong Kong Polytechnic University. His current research focuses on mobile security and privacy, IoT security, and vehicle security.

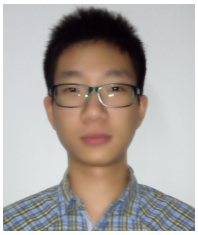
Haoyu Wang is a Full Professor in the School of Cyber Science and Engineering at Huazhong University of Science and Technology (HUST). He is leading the SECURITY PRIDE Research Group (Security, Privacy, and Dependability in Emerging Software Systems). His research covers a wide range of topics in Software Analysis, Privacy and Security, eCrime, Internet/System Measurement, and AI Security. He has published over 100 peer-reviewed papers. He has been awarded three best/distinguished paper awards, including WWW 2020 Best Student Paper Award (the first award from China), and ACM OOPSLA 2020 Distinguished Paper Award. More information can be found at: https://howiepku.github.io/

Xiapu Luo is an associate professor in the Department of Computing, The Hong Kong Polytechnic University. His research focuses on Blockchain/smart contracts, Mobile/IoT security and privacy, Network/Web Security and Privacy, Software Engineering and Internet Measurement with papers published in top conferences and journals. His research led to eight best/distinguished paper awards, including ACM SIGSOFT Distinguished Paper Award in ICSE'21, Best Paper Award in INFOCOM'18, Best Research Paper Award in ISSRE'16, etc. and several awards from the industry. He regularly serves in the program committee of major security conferences and is currently an editor of IEEE/ACM Transactions on Networking.

Tao Zhang received the BS degree in automation, the MEng degree in software engineering from Northeastern University, China, and the PhD degree in computer science from the University of Seoul, South Korea. After that, he spent one year with the Hong Kong Polytechnic University as a postdoctoral research fellow. Currently, he is an associate professor with the School of Computer Science and Engineering, Macau University of Science and Technology (MUST). Before joining MUST, he was the faculty member of Harbin Engineering University and Nanjing University of Posts and Telecommunications, China. He published more than 60 high-quality papers at renowned software engineering and security journals and concerences such as TSE, TIFS, TDSC, TR, ICSE, etc. His current research interests include AI for software engineering and mobile software security. He is a senior member of IEEE and ACM.

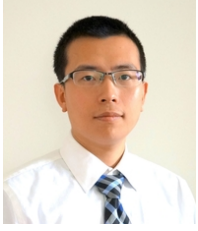
Kang Liu is a Professor in National Laboratory of Pattern Recognition (NLPR), Institute of Automation, Chinese Academy of Sciences. He obtained the Ph.D. degree in Computer Science from NLPR in 2010, under the supervision of Prof. Jun Zhao. His research interests include sentiment analysis, information extraction, question answering, and natural language processing. He has published more than 60 papers at top conferences and journals including TKDE, ACL, IJCAI, EMNLP, WWW, CIKM, COLING etc.

Jiachi Chen is an assistant professor at the School of Software Engineering, Sun Yat-Sen University. He received his Ph.D degree at the Faculty of Information Technology, Monash University, Australia, under the supervision of Prof.John Grundy, Dr.Xin Xia and Dr.Jiangshan Yu. Prior to joining Monash University, he spent two years at the Hong Kong Polytechnic University as a research assistant advised by Dr. Xiapu Luo. His research interests include blockchain, smart contracts, mining software repository, software security, and empirical study.



Hao Zhou is a PhD student at the Hong Kong Polytechnic University. Before attending PolyU, he worked at PolyU as a research assistant from 2016 to 2018. Hao's research generally focuses on software and system security. Specifically, his research interest includes mobile app analysis, IoT security, fuzzing and software testing.

Yutian Tang received the BSc degree in Computer Science from Jilin University, China, and the PhD degree in software engineering from The Hong Kong Polytechnic University, Hong Kong SAR, China. He is currently an assistant professor with School of Information Science and Technology, ShanghaiTech University. His current research interests include mobile security and privacy, software product line, empirical software engineering, and testing. He has published papers in top-tier software engineering conferences and journals. He is a member of IEEE, HKCS, CCF and EuroSys.



Xusheng Xiao is an Assistant Professor in the Department of Computer and Data Sciences at Case Western Reserve University (CWRU). His general research interests span between software engineering and computer security, with the focus on developing advanced analysis techniques to analyze complex software behaviors for improving the reliability and the security of complex software and systems.