Localizing Function Errors in Mobile Apps with User Reviews

Le Yu*, Jiachi Chen*, Hao Zhou*, Xiapu Luo*§ and Kang Liu†
*Department of Computing, The Hong Kong Polytechnic University
†National Laboratory of Pattern Recognition, Institute of Automation Chinese Academy of Sciences
{cslyu, csjcchen, cshaoz, csxluo}@comp.polyu.edu.hk, kliu@nlpr.ia.ac.cn

Abstract—**Removing all function errors is critical for making successful mobile apps. Since app testing may miss such 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 does not take into account apps' code. Moreover, recent studies on mapping reviews to problematic source files just look for the matching between the words in reviews and that in source code, and thus result in many false positives and false negatives. 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 in 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 tool.**

I. 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[1] are very important to developers for learning their apps' bugs[2], limitations[3], and strengths[4]. 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 complaints, 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[5], [6], [7], [8], [9], [10] without taking into account apps' code, and thus they cannot help developers locate the problematic code according to the reviews. Palomba et al. recently propose `ChangeAdvisor` [11] for mapping reviews to source code by first clustering similar reviews and then comparing the topic words identified from the clusters and the words extracted from the names of code components (e.g., methods, classes). Unfortunately, `ChangeAdvisor` may lead to many false positives, because it does not exploit the semantic information in reviews and code, and false negatives, because the information contained in the names of code components is limited.

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 semantic meanings with the hints of context information in user reviews. We aim at Android apps because Android has occupied 85.0% market share of mobile operating systems [12] and there are already 3.3 million apps in Google Play [13]. In particular, `ReviewSolver` exploits three new observations. First, as shown in Section II-B, 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 within. 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 error was related to 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

[§] The corresponding author.

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.

Therefore, to help developers automatically map function error reviews to code, `ReviewSolver` first identifies such kind of negative reviews through supervised machine learning and analyzes each sentence in such reviews to extract useful verb phrase and noun phrase through natural language processing (NLP) techniques. Then, it conducts static bytecode analysis on apps to extract seven kinds of information (Section III-C). Finally, `ReviewSolver` maps the reviews to the code according to their semantic similarity and recommends the most related code to developers.

We carefully evaluate the performance of `ReviewSolver` and compare it with `ChangeAdvisor` [11], 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` needs apps' source code. The experimental results show that `ReviewSolver` can identify function error related reviews with 84.6% precision and 88.5% recall rate. For the same reviews that can be correlated to code files (by checking bug reports), `ReviewSolver` can correctly locate 79 code files whereas `ChangeAdvisor` can only correctly locate 31 code files. Moreover, `ReviewSolver` can map 45.3% of function error related reviews to code whereas `ChangeAdvisor` can only map 7.4% of such reviews.

In summary, our major contributions include:

- 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 apps in Google Play and their reviews, and compare it with `ChangeAdvisor`, the state-of-the-art tool. The results show that `ReviewSolver` outperforms `ChangeAdvisor` in terms of correctly mapping more reviews to code.

The rest of this paper is organized as follows. Section II introduces the background and motivating examples. Section III and Section IV detail the design of `ReviewSolver`. We present the experimental result in Section V and discuss the limitation of `ReviewSolver` in Section VI, respectively. After introducing the related work in Section VII, we conclude the paper in Section VIII.

II. Background and Motivating Examples

A. Function Error Related Reviews

By manually analyzing 6,390 user reviews, Khalid et al. [14] 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.

B. Context Information in Reviews

A key insight behind `ReviewSolver` is that users may describe the context under which an error occurred when writing reviews [15]. Such context information provides us useful hints to locate the problematic codes. To further illustrate it, we randomly select 500 function error reviews with at least 4 words from 18 open-source apps (Section V-D, Table V), manually read them, and summarize the context of the function errors. As shown in Table I, 56.0% function error reviews contain more or less context information. We use 5 examples to illustrate how to locate the problematic code by exploiting such context information in Section II-C.

TABLE I
THE CONTEXT INFORMATION IN FUNCTION ERROR REVIEWS.

Context information	Description	Percentage	Example
(1)App Specific Task	Errors appear when performing app specific tasks	25.2% (126/500)	"Auto backup doesn't work ..."
(2)Updating App	Errors appear after updating the app	8.2% (41/500)	"New update doesn't work with the s3."
(3)GUI	Errors appear when interacting with GUI	3.4% (17/500)	"Note 4 does not have menu hard button."
(4)Error Message	Reviews contains the error messages from apps	2.8% (14/500)	"it just says "c:geo can't load data required to log visit""
(5)Opening App	Errors appear when opening the app	1.6% (8/500)	"It crashed every time I opened it."
(6)Registering	Errors appear when logining/registering account	2.2% (11/500)	"Cannot login to my gmail"
(7)API/URI/intent	Errors appear when accessing resource or information	12.8% (64/500)	"But I cannot save photos to sd card with it"
(8)Other	User does not describe the context	44.0% (220/500)	"Sometimes not working."

As shown in Table I, most errors (25.2%) related to the functions specific to an app ("App Specific Task" in Table I). Since different apps have different specific functions, it is difficult to predefine some classes and group these functions into them. Hence we will look for the classes/methods that implement these functions. Since 8.2% 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 (2.8%), and hence we locate such error by checking the classes that display such error message. For function error reviews (1.6%)

that report crashing right after the app is launched, we will locate and check the starting activity. If the errors happen when registering account or during login (2.2%), we look for and examine the account registration and login related activities. 12.8% errors are related to the resource/information of the device. Since such resource/information could be accessed by using APIs/URIs/intent, we will locate the problematic code through the corresponding APIs/URIs/intent.

We divide these errors into two categories and detail how to map them to code in Section IV-A and Section IV-B individually. One category includes app specific errors that are related to the functions implemented by developers (i.e., case (1)-(6) in Table I). The other one refers to the general error that are related to Android interface (i.e., case (7) in Table I).

Table I also shows that 44.0% 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 VI and will investigate them in future work.

C. Motivating Examples

To clearly differentiate our approach (i.e., ReviewSolver) and the state-of-the-art method (i.e., ChangeAdvisor[16]), we use a few 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 [11] 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 following **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()* is found, 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 check 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 check 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 check the class *com.fsck.k9.notification.SendFailedNotifications* since it raises this message.

III. System Design

A. System Overview

Fig. 1 shows the procedure of ReviewSolver. After crawling reviews from Google Play, the review analysis module identifies function error reviews (Section III-B). The static analysis module extracts useful information from the executable of an app (Section III-C). Combining the information from reviews and code, ReviewSolver map the function error reviews to the problematic code (Section IV).

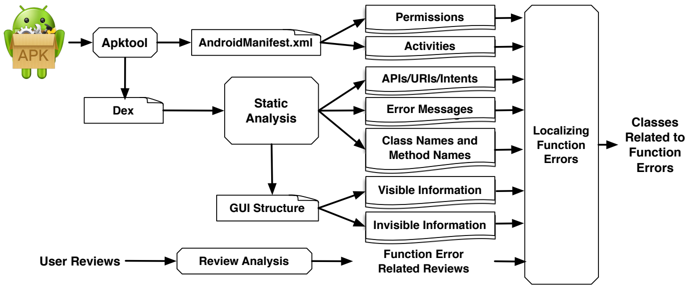
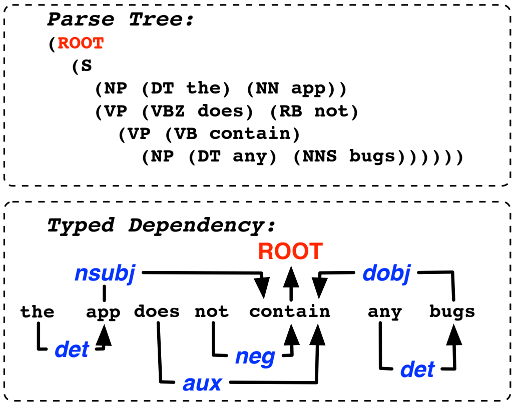


Fig. 1. Overview of `ReviewSolver`: Localizing Function Errors

B. Review Analysis

The review analysis module identifies the function error related reviews from those negative reviews (i.e., those with 1-2 stars), and then extracts the verb phrases and noun phrases from such reviews to facilitate localizing function errors.

Pre-processing user reviews We remove the non-ASCII characters and split the remaining content into distinct sentences by using NLTK [17]. To remove typos, we leverage the edit distance [18] to discover the correct word if the word is not found in dictionary. Abbreviations are replaced with their original words (e.g., “pls” to “please”, “u” to “you”). For each sentence in the review, we leverage Stanford Parser [19] 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 means verb phrase. The typed dependency relation refers to the grammatical relation between two words [20]. For example, *dobj* in Fig.2 means direct object.



```
Parse Tree:
(ROOT
  (S
    (NP (DT the) (NN app))
    (VP (VBZ does) (RB not)
      (VP (VB contain)
        (NP (DT any) (NNS bugs))))))
```

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

Identifying function error reviews We use supervised machine learning algorithms to identify the function error reviews described in Section II. 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 [16], [21], [22], [23]. TF-IDF (i.e., Term Frequency-Inverse Document Frequency) measures how important a word is to a review [24]. It is calculated by multiplying term frequency (TF) and inverse document frequency (IDF). TF measures how frequently a word occurs in a review. IDF measures how important a word is. The frequent words are less important.

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

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

N-Grams are a set of co-occurring words within a given window [25]. 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 [16], [21]) 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 data set with 700 positive reviews and 700 negative reviews. We test multiple algorithms (e.g., random forest, SVM, max entropy) and adopt the boosted regression trees [26] because it has good performance in text classification [16]. 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 [27]. When performing the classification, we did not split reviews into sentences because considering individual sentences may miss the context information in other sentences.

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 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 IV).

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.

C. Static Analysis

Given an app, we analyze its `AndroidManifest.xml` file and `Dex` file to extract seven kinds of information to facilitate the mapping from function error reviews to code. In detail, we use Vulhunter [28] to process the apk 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 [29] 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.

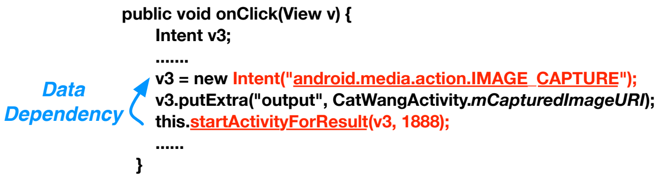
Extracting APIs/URIs/intents, error message, and class/method names We analyze the APG to identify three kinds of information (i.e., APIs/URIs/intents, class/method names, error messages).

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.

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 [30]. 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 [31] 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 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).

If error occurs, an app may notify user the detail [32] 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.



```
public void onClick(View v) {
    Intent v3;
    .......
    v3 = new Intent("android.media.action.IMAGE_CAPTURE");
    v3.putExtra("output", CatWangActivity.mCapturedImageURI);
    this.startActivityForResult(v3, 1888);
    ......
}
```

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

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 [33], we extract them and use them to locate app specific task errors described in user reviews (Section IV-A).

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 contained 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").

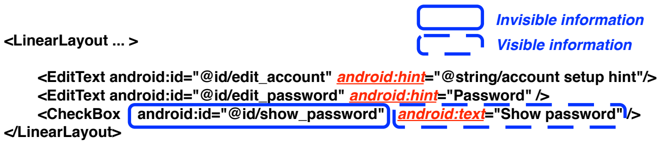
Developers can design GUI via the layout XML file or dynamically change it through APIs (e.g., *TextView.setText()*) [34]. We enhance `GATOR` [35] to recover the GUI structure of all activities. GATOR first parses the manifest file (to identify the activities), the layout file (to get the parent-child 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 layout file and the dynamically generated widgets inferred from the constraint graph to reconstruct the GUI structure. We enhance GATOR from four aspects.

First, we empower `GATOR` to handle the self-defined namespace. Note that the "xmlns" attribute defines the namespace in each `AndroidManifest.xml` and layout file. To use intrinsic attributes provided by Android (e.g., `android:name`, `android:id`), developers can set the `namespace-prefix` as "android" and set the `namespaceURI` as "http://schemas.android.com/apk/res/android". However, developers can set them to the namespaces defined by themselves whereas `GATOR` does not handle them.

Second, we enhance `GATOR` to process the "activity-alias" tag. The element with this tag contains two attributes (i.e., `namespace:name` and `namespace:target`), which means that the two activities defined in the attributes have the alias relationship. Without considering the "activity-alias" tag, we cannot identify the child widget of the alias activities.

Third, GATOR only identifies the mapping between id names and values by parsing the the res/values/public.xml but miss the fields of automatically generated classes (e.g., “android.R$...”, “com.android.internal.R$...”). Thus, some widget obtained in code by calling *findViewById(int)* cannot be located in the corresponding GUI structure. We parse these generated classes to get the ids of these fields.

Fourth, the android:text attribute defines the text displayed while the android:hint attribute provides suggestive information. Since both of them provide hints for correlating function error reviews, we extend GATOR to parse them. Fig.4 shows an example of layout file that contains these two attributes.


Invisible information
Visible information

```
<LinearLayout ... >
    <EditText android:id="@id/edit_account" android:hint="@string/account setup hint"/>
    <EditText android:id="@id/edit_password" android:hint="Password" />
    <CheckBox android:id="@id/show_password" android:text="Show password"/>
</LinearLayout>
```

Fig. 4. Snippet of a layout file

After reconstructing the GUI structure of each activity, to identify the text displayed by the app, we filter out the widgets that are not subclasses of *TextView* classes. For the remaining widgets (e.g., *Button*), we extract the visible label information from their android:text and android:hint if any. For example, in Fig. 4, we extract the text “Show password” of the *CheckBox* object. Then, we extract the invisible label information by parsing their ids. For example, in Fig. 4, we collect the id (i.e., show_password) of *CheckBox* and split it into a series of words (i.e., “show”, “password”).

IV. Localizing Function Errors

By correlating the information extracted from user reviews and apps, ReviewSolver first localizes app specific errors (Section IV-A) and general errors (Section IV-B), and then rank the selected classes before recommending them.

A. Localizing App Specific Errors

To localize the app specific errors, we leverage various information extracted from the apk file.

Using Class/Method Name If the error appears when performing app specific tasks, for each verb phrase extracted from function error review, we check whether it is similar to that of each method. If so, we recommend the developer to check the corresponding method. Motivated by the method in [33], we leverage the camel case to convert the method name to verb phrase. 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 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 use Word2Vec [36] 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 [37], [38]. More precisely, by using the model trained on Google News dataset (contains 300-dimensional vectors for 3 million words and phrases) [39], 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 that threshold value (0.68 by referring [40]), we regard them as similar ones.

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

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 the noun phrase extracted from review. 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 for modifying the widget (e.g., “reply”) and look it up 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 manually reading the function error review, we find that users can also vaguely describe the error by using the two semantic patterns shown in Table II. $[function]$ 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 P1 and P2, and then look them up on the GUI’s visible label information. The activities that contain these words will be recommended to developer. For example, for P2, we recommend the developer to check the activity that contains the verb “register”.

TABLE II
Two 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*”

Localizing Errors Related to Error Message Users can describe the error message accurately. 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 true, 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()*.

Localizing Errors Related to Opening App 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.

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 registering account related phrases (e.g., "sign in", "login").

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.

B. Localizing General Errors

We propose Algorithm 1 to locate the errors related to API/URI/intent. For API, we compare the verb phrase extracted from review with the verb phrase 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 phrase 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 phrase 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 method 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 method signature into verb phrase by using the method described in Section IV-A. We will also extract verb phrases from its official description by using the typed dependency [20]. For example, we can extract verb phrases such as "open communication link", "establish connection" from the description of the API *URLConnection.connect()*.

For the verb phrase extracted from review, if its verb (or its synonyms) is included in the method name of the API and the object is included in the class description of the API, we also recommend the developer to check the API and corresponding class. For example, "connect server" can be mapped to the API *HttpURLConnection.connect()* since the verb "connect" is included in the API's method name and "server" in the official description of the class *HttpURLConnection*. 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 [41] and its object is similar to the personal information protected by the permission, we also recommend the developer to check the permission related API and corresponding class.

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

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 [31] to get the permission related to the URI. Then, we regard the noun phrase extracted from the permission description [42] 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 [43] provides 11 kinds of common intents. For example, "camera" is related to the intent with the action `android.media.action.IMAGE_CAPTURE`.

C. Ranking the Classes

Since we employ multiple approaches to map function error reviews to code, one review can be mapped to multiple classes. We compute the importance of these classes, and recommend the top N most related ones to developers. 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 IV-A and Section IV-B. 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 increase by one. The selected classes are ranked according to their importance.

V. Experimental Result

In this section, we conduct a series of experiments to answer the following research questions:

RQ1: Can ReviewSolver correctly identify function error related reviews (Section V-B)?

RQ2: How is the performance of ReviewSolver when comparing with the another system ChangeAdvisor [11] (Section V-C)?

RQ3: How many function error related reviews can be addressed by ReviewSolver (Section V-D)?

For RQ2, to check the performance of ReviewSolver, we first employ ReviewSolver to identify the function error related reviews. Then we use ReviewSolver and ChangeAdvisor [11] to map these function error related reviews to code respectively. For each review, we search the related bug reports to find the code files modified by developers. These code files are regarded as the code files related to the function error review (i.e., ground truth). We compare how many code files in the ground truth can be found by ReviewSolver and ChangeAdvisor.

A. Data Set

To measure how many function error related reviews can be solved, we select the 18 apps that provide source code in F-Droid or Github as data set. All these 18 apps can be downloaded from Google Play. For each app, we download the latest version of apk file and new user reviews. We crawl 69,359 reviews from Google Play and 12,735 of them are negative ones (rated 1 or 2 starts).

For RQ2, to compare the performance of ReviewSolver with ChangeAdvisor (Section V-C), we ask three students to build up ground truth that contains links between the function error related reviews and code files. This is done by reading the apps' function error related reviews and bug reports. As shown in Fig. 5, after reading the function error related review, the student identifies the bug described in the review. Then he searches the bug in existing bug reports. If the bug has been fixed in some bug reports, the files modified by the developers are regarded as the code files related to the review (i.e., ground truth). Since some apps do not contain bug reports, we only include 8 apps that contain bug reports in the ground truth. For each app, 200 function error related reviews are analyzed.

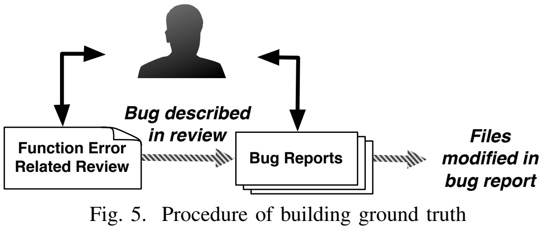


Fig. 5. Procedure of building ground truth

B. Review Identification Performance

To test the performance of classifying function error related reviews, we employ the datasets provided by Ciurumelea el al. [16]. The function error dataset contains 199 reviews (87 of them are function error related ones). The result is shown in Table III. Our system achieves 84.6% precision and 88.5% recall rate when detecting function error related reviews. We manually analyze the cause of false positives/negatives.

TABLE III
Result of Classifying Function Error related Reviews

	TP	FP	FN	Precision	Recall	F-1
Function Error	77	14	10	84.6%	88.5%	86.5%

False positives. The major cause of false positive is that although some reviews contain error related words (e.g., "bug", "problem"), the objects that user really wants 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 can analyze the tense of the review to identify the fixed bugs (e.g., "... has been fixed"). To remove the false positives caused by bugs of other apps, we need to check the subject related to the bug (e.g., "my apps").

False negatives. The major cause of false negative is that user can describe function error 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 describe that the error makes the tablet "Slow", thus ReviewSolver cannot recognize it. We can add these 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: ReviewSolver can solve 84.6% precision, 88.5% recall rate when identifying function error related reviews.

C. Performance of ReviewSolver

To measure the performance of ReviewSolver, for the review to code links contained in ground truth (described in Section V-A), we compare how many of them can be found by ReviewSolver and ChangeAdvisor. The "#Total Links" column of Table IV shows the total number of links between reviews and code files that we use as ground truth. From the "#RS True links" and "#CA True links" columns of the Table IV, we can see that the number of reviews solved by ReviewSolver is greatly larger than existing system ChangeAdvisor. For example, for the app com.fsck.k9, ReviewSolver can identify 17 links, while ChangeAdvisor can only identify 2. In total, the number of links identified by ReviewSolver (i.e., 79 links) is twice as large as that identified by ChangeAdvisor (i.e., 31 links).

Links missed by ReviewSolver. As described in Table IV, ReviewSolver can only cover part of the links contained in the ground truth. One reason is that the classes of the apk file published on Google Play are not consistent with the classes mentioned in bug report. Some classes

TABLE IV
Total number of function error reviews that can be mapped to bug reports, the number of files identified by using bug reports, the number of correct files identified by ReviewSolver (column “#RS True Links”) and ChangeAdvisor (column “#CA True Links”)

Apk Name	#Error Reviews	#Total Links	#RS True Links	#CA True Links
org.mariotaku.twidere	78	314	15	6
org.thoughtcrime.securesms	50	132	4	0
com.fsck.k9	55	125	17	2
com.battlelancer.seriesguide	65	216	8	3
org.wordpress.android	94	275	11	2
cgeo.geocaching	38	342	11	7
com.joulespersecond.seattlebusbot	24	154	9	8
de.danoeh.antennapod	43	230	4	3
Total	447	1,788	79	31

mentioned in bug report are not included in the apk file. For example, we find that the “*Some pictures can’t be viewed*” bug of the app `org.mariotaku.twidere` is fixed by modifying three files (e.g., `RapidImageDecoder.java`). However, we cannot find the corresponding class in the apk file. `ReviewSolver` cannot locate error in it. Another reason is that `ReviewSolver` cannot identify the classes indirectly called by other classes. For example, when fixing the “*Error connecting to MMS provider*” bug of the app `org.thoughtcrime.securesms`, the class *org.thoughtcrime.securesms.jobs.MmsSendJob* and *org.thoughtcrime.securesms.util.TelephonyUtil* are used. The former class uses the latter class to get the telephone number of the device. `ReviewSolver` cannot recognize the latter since it is indirectly called by the former. To decrease the number of missed links caused by obfuscation, we need to infer the function of the methods by checking the API dependency graph [44]. To reduce the missed links generated by the function call relationship, we can use the method call graph to find more methods related to the error.

Answer to RQ2: The experimental result shows that: For the same number of function error related reviews, `ReviewSolver` can find 79 classes acknowledged in bug reports while `ChangeAdvisor` can only find 31.

D. Results of Solving Function Error related Reviews

As shown in Table V, `ReviewSolver` discovers 4,912 function error related reviews, which account for 38.6% (i.e., 4,912/12,735) in all negative reviews. 45.3% (i.e., 2,226/4,912) of these function error related reviews can be mapped to code level information. This number is much larger than that of `ChangeAdvisor`, which can only map 365 of them to code (i.e., 7.4%, 365/4,912). `ChangeAdvisor` groups these function error related reviews into clusters first and then it maps these clusters to source code files. This means for most clusters and their corresponding function error related reviews, `ChangeAdvisor` cannot map them to any code files.

1) Distribution of Context Information Used When Locating Error: `ReviewSolver` uses different kinds of context information to map review to code. To measure the effectiveness of different kinds of context information, for each context information, we measured the number of function error reviews that can be located by using it. The result is shown in Table VI. We can find that, for the errors that can be located by `ReviewSolver`, nearly half of them were based on the app specific task related information. This result is consistent with the result of manually check (shown in Table I). In Table I, “(1) App Specific Task” is the most frequent kind of context information.

TABLE V
Total number of negative reviews (column “Negative Review”), the number of function error reviews identified by ReviewSolver (column “#Error Review”), the number of function error reviews that can be mapped to code by using ReviewSolver (column “#RS Map”), the number of function error reviews that can be mapped to code by using ChangeAdvisor (column “#CA Map”).

#	Apk Name	Negative Review	#Error Review	#RS Map	#CA Map
1	org.mariotaku.twidere	375	145	86	13
2	com.zegoggles.smssync	1310	526	162	36
3	org.thoughtcrime.securesms	319	112	87	14
4	com.totsp.crossword.shortyz	646	327	142	20
5	com.fsck.k9	1155	547	351	27
6	com.andrewshu.android.reddit	289	150	52	16
7	fr.xplod.focal	645	317	179	18
8	org.geometerplus.zlibrary.ui.android	593	161	72	11
9	com.battlelancer.seriesguide	417	110	56	3
10	org.wordpress.android	1182	499	363	32
11	com.kmagic.solitaire	1547	531	107	66
12	org.coolreader	939	381	144	24
13	cgeo.geocaching	475	168	114	12
14	com.joulespersecond.seattlebusbot	469	132	20	12
15	com.achep.acdisplay	907	293	72	9
16	de.danoeh.antennapod	94	38	21	10
17	com.frostwire.android	923	253	108	9
18	com.ichi2.anki	450	190	90	33
	Total	12,735	4,880	2,226	365

TABLE VI
Number of function error reviews that can be located with different context information.

Context Type	#Function Error	Percentage
App Specific Task	1052	47.1%
Updating app	799	35.8%
API/URI/intent	587	26.3%
Registering Account	266	11.9%
Opening App	69	3.1%
GUI	82	3.7%
Error Message	31	1.4%

The distribution values of Table VI are different from the values shown in Table I due to two reasons. The first reason is that, when calculating Table VI, the reviews do not contain any context information are ignored and we only consider the reviews that can be located in code by using context information. However, for Table I, we put those review that do not contain any context information in the case “(8) Other”. Such reviews account for more than 40% in all function error related reviews. Another reason is that, when calculating Table VI, we find some reviews that contain multiple kinds of context information. For example, for the error reviews that describing updating app related behavior, 36.7% (293/799) of them also contain other context information (e.g., API/URI/intent). However, we only consider the major context information when building Table I.

2) Correctness of the Linkings between Function Error Reviews and Code: In order to check the precision of the mapping between reviews and code generated by `ReviewSolver`, we manually check 50 mappings between reviews and code for each app. The result is shown in the third and fourth columns of Table VII. We can find that ReviewSolver can achieve 73.1% precision when mapping reviews to code.

TABLE VII
CORRECTNESS OF THE MAPPINGS BETWEEN REVIEWS AND CODE FOUND BY REVIEWSOLVER

#	Apk Name	ReviewSolver #Correct/Check	ReviewSolver Precision
1	org.mariotaku.twidere	38/50	76.0%
2	com.zegoggles.smssync	33/50	66.0%
3	org.thoughtcrime.securesms	40/50	80.0%
4	com.totsp.crossword.shortyz	38/50	76.0%
5	com.fsck.k9	43/50	86.0%
6	com.andrewshu.android.reddit	37/50	74.0%
7	fr.xplod.focal	40/50	80.0%
8	org.geometerplus.zlibrary.ui.android	19/50	38.0%
9	com.battlelancer.seriesguide	29/50	58.0%
10	org.wordpress.android	41/50	82.0%
11	com.kmagic.solitaire	43/50	86.0%
12	org.coolreader	36/50	72.0%
13	cgeo.geocaching	42/50	84.0%
14	com.joulespersecond.seattlebusbot	14/23	60.8%
15	com.achep.acdisplay	34/50	68.0%
16	de.danoeh.antennapod	17/25	68.0%
17	com.frostwire.android	38/50	76.0%
18	com.ichi2.anki	38/50	76.0%
	Total	620/848	73.1%

Cause of false mappings. The major cause of the false mappings is that the phrase extracted from review is not related to the error, but we still use it to locate error. For example, consider the review "*I was able to add cards to the decks, but now I'm unable to even view the cards*". As user is able to "add cards", the error is not related to this behavior. ReviewSolver still map this phrase to classes. To remove such false mappings, we need to conduct sentiment analysis on the sub-sentences of the review. Those positive/neutral ones will be filtered when localizing error.

In order to show the correctness of the function error related classes found by ReviewSolver, we select 10 apps that contain commit messages in Github. For each app, we select 50 function error related reviews and their related classes found by ReviewSolver. After checking the time of the comment, we count the number of classes that were modified within three months. The result is shown in Table VIII. We can find that 38.6% (832/2,153) of the classes found by ReviewSolver have been modified within three months.

TABLE VIII
FUNCTION ERROR RELATED CLASSES MODIFIED WITHIN THREE MONTHS

#	Apk Name	# modified / # found
1	org.mariotaku.twidere	59/260
3	org.thoughtcrime.securesms	100/354
5	com.fsck.k9	148/251
9	com.battlelancer.seriesguide	92/212
10	org.wordpress.android	153/255
13	cgeo.geocaching	110/205
14	com.joulespersecond.seattlebusbot	30/75
16	de.danoeh.antennapod	46/209
17	com.frostwire.android	44/154
18	com.ichi2.anki	50/178
	Total	832/2,153

Answer to RQ3: The experimental result shows that: ReviewSolver can solve 45.3% function error related reviews.

VI. DISCUSSION

Some factors will affect the performance of ReviewSolver. When identifying the function error related reviews, the training set cannot cover all kinds of reviews that describe the error implicitly (e.g., "... is hard to load"). To overcome this limitation, we will increase the training set in future. Currently, we only consider low score reviews but we will leverage sentiment analysis to identify negative sentences from positive/neutral reviews in future. When mapping function error related reviews to code, ReviewSolver cannot filter out all useless phrases that can be mapped to code incorrectly. To decrease this threat, we will try use machine learning algorithm to identify useless phrase. When conducting static analysis on apk file, if the developer employs obfuscation technique to hide the class names and method names, ReviewSolver cannot generate summary for each method. The app specific task related errors cannot be located in code. To overcome this limitation, we will extract the API dependency graph to infer the function of the method [44]. Some function error related reviews cannot be located since they are related to compatibility issues of specified device. We can use information retrieval technique to extract the types of devices and report them to developer automatically. Other function error related review that simply describes that the app do not work can be solved by grouping them together to get more detailed information for analysis.

VII. RELATED WORK

A. Review Analysis

A number of studies have been conducted on the user reviews of app store [1]. However, the majority of them just analyze user reviews without correlating them with apps' code. Chen et al. [45] combine static and dynamic features with comment features to predict the popularity of apps. Khalid et al. manually analyze user reviews and uncovered 12 types of user complaints [14]. To identify correlations between error-sensitive permissions and error-related reviews, Gomez et al. [46] leverage LDA and J48 process the permissions and reviews.

Some other researchers extract app features from reviews. Iacob et al. [5] define a set of pre-defined linguistic rules to match feature request related reviews. Then they use LDA to identify common topics in these reviews. AR-Miner [10] 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/low level taxonomy [16] and apply machine learning to classify the reviews. AutoReb [47] 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 [4] extracts the semantic dependence relation between words and utilizes cluster algorithm to identify users' opinion towards corresponding aspect. In [21], Walid Maalej et al. combine text classification, NLP, and sentiment analysis to classify reviews into four categories.

To generate summaries of users feedback, SURF [48] classifies review sentence into different categories by utilizing the intentions and topics of the reviews.

ChangeAdvisor [11] is most close to our work since they also analyze code. ChangeAdvisor [11] employs the HDP algorithm [49] to extract topic words from the clusters of function error related reviews. Then it calculates the asymmetric dice similarity coefficient [50] between these topic words and the words extracted from source code file. If the result reaches a threshold, then it recommends the developer to check the corresponding code file. The major difference between our system and ChangeAdvisor is that: 1) We employ static analysis to extract the starting activity, requested permissions, APIs/URIs/intents, error message, class/method names, visible information and invisible information of GUI from apk. We do not simply split the code into distinct words like [16] [11] since many useless words will be included and they will affect the correctness of mapping reviews to code. 2) When mapping the review to code, ChangeAdvisor [11] checks the number of words shared by review and code file. It does not consider the synonyms, which will generate false negatives. We leverage word embedding method to measure semantic similarity between the two phrases.

B. Code Analysis

Many static analysis systems have been proposed to analyze the apk file of mobile apps [51]. EdgeMiner [52] 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 [53] proposes to build up model for third-party libraries by analyzing non-obfuscated apps. Then it employs this model to recover the obfuscated class, method, and variable names of new apks. If the developer use packing services to hide the dex files, PackerGrind [54] and Malton [55] can be used to unpack the app. FlowDroid [30] build up dummy main method for each component and then it performs static taint analysis to identify the source to sink path. To analyze the inter component communication of apps, IccTA leverage IC3 to discover the ICC links and create dummy method for them [29]. To detect piggybacked apps, Fan et al. generate a sensitive subgraph to profile the app and extract features from it [56], [57].

Some researchers employ static analysis to analyze the GUI of apps. AutoPPG [58], [59] leverage Vulhunter [28] to analyzes the callbacks of GUI and the conditions of sensitive behaviors to generate precise privacy policies. To find the contradiction between user interface and code, AsDroid [60] compare the behavior found by static analysis with the behavior identified from UI to find contradiction. To identify the sensitive user input, UIPicker [61] determines sensitive input fields by using a supervised learning classifier that is based on the features extracted from the texts of the UI elements. The major difference between our system and [60], [61] is that we enhance Gator [35] to extract the text set in code.

C. Linking Document to Code

Information retrieval (IR) [50] 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 technique can rank the relevance between the input document and code files. BLUIR [62] extract 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 [63] 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 [64] 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 locate feature related code, SNIAFL [65] 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 [66] 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. [67] propose to utilize sentence ranking to select proper sentences from historical bug reports.

VIII. Conclusion

User reviews of mobile apps can help developer discover the function errors uncaught by app testing. Since manually processing reviews is time-consuming and error-prone, the state-of-the-art automated approach results in many false positives and false negative. 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 ChangeAdvisor, the state-of-the-art tool. In total, ReviewSolver solved 45.3% function error reviews whereas ChangeAdvisor only handled 7.4%. We have uploaded the apk files, source code, reviews of these apps, and the implemented system to the following URL: https://github.com/yulele/ReviewSolver.

Acknowledgements

This work is supported in part by the Hong Kong GRF (152279/16E, 152223/17E), the Hong Kong RGC Project (CityU C1008-16G), and the National Natural Science Foundation of China (No. 61502493).

References

[1] 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.

[2] D. Pagano and W. Maalej, "User feedback in the appstore: An empirical study," in *Proc. RE*, 2013.

[3] 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.

[4] X. Gu and S. Kim, "What parts of your apps are loved by users?" in *Proc. ASE*, 2015.

[5] C. Iacob and R. Harrison, "Retrieving and analyzing mobile apps feature requests from online reviews," in *Proc. MSR*, 2013.

[6] L. V. G. Carreño and K. Winbladh, "Analysis of user comments: an approach for software requirements evolution," in *Proc. ICSE*, 2013.

[7] E. Guzman and W. Maalej, "How do users like this feature? a fine grained sentiment analysis of app reviews," in *Proc. RE*, 2014.

[8] 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.

[9] D. H. Park, M. Liu, C. Zhai, and H. Wang, "Leveraging user reviews to improve accuracy for mobile app retrieval," in *Proc. SIGIR*, 2015.

[10] 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.

[11] 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.

[12] IDC, "Smartphone os market share, 2017 q1," https://goo.gl/ymAhw9, 2017.

[13] statista, "Number of available applications in the google play store," https://goo.gl/My5ATw, 2017.

[14] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Software*, 2015.

[15] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? on automatically classifying app reviews," in *Proc. RE*, 2015.

[16] A. Ciurumelea, A. Schaufelbühl, S. Panichella, and H. Gall, "Analyzing reviews and code of mobile apps for better release planning," in *Proc. SANER*, 2017.

[17] S. Bird, E. Loper, and E. Klein, "Natural language toolkit," http://www.nltk.org/, 2017.

[18] M. Gilleland, "Levenshtein distance," https://goo.gl/TVA9Ga, 2017.

[19] M.-C. De Marneffe, B. MacCartney, C. D. Manning *et al.*, "Generating typed dependency parses from phrase structure parses," in *Proc. LREC*, 2006.

[20] M.-C. De Marneffe and C. D. Manning, "Stanford typed dependencies manual," Tech. Rep., 2008.

[21] W. Maalej, Z. Kurtanović, H. Nabil, and C. Stanik, "On the automatic classification of app reviews," *Requirements Engineering*, 2016.

[22] W. B. Cavnar, J. M. Trenkle *et al.*, "N-gram-based text categorization," *Ann Arbor MI*, 1994.

[23] 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.

[24] S. N. Group, "Tf-idf weighting," https://goo.gl/RrWHVc, 2009.

[25] A. . M. Text Mining, "What are n-grams?" https://goo.gl/bwAHtp, 2017.

[26] J. Elith, J. R. Leathwick, and T. Hastie, "A working guide to boosted regression trees," *Journal of Animal Ecology*, 2008.

[27] Bhuvan Sharma, "What are the advantages/disadvantages of using gradient boosting over random forests?" https://goo.gl/N6y2Kw, 2015.

[28] C. Qian, X. Luo, Y. Le, and G. Gu, "Vulhunter: toward discovering vulnerabilities in android applications," *IEEE Micro*, 2015.

[29] 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.

[30] 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.

[31] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proc. CCS*, 2012.

[32] stackoverflow, "Best practice for displaying error messages," https://goo.gl/odr84X, 2013.

[33] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for java methods," *IEEE Trans. TSE*, 2016.

[34] A. Tutorials, "Static and dynamic textview creation," https://goo.gl/BdyCW2, 2012.

[35] A. Rountev and D. Yan, "Static reference analysis for gui objects in android software," in *Proc. CGO*, 2014.

[36] Y. Goldberg and O. Levy, "word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method," *arXiv preprint arXiv:1402.3722*, 2014.

[37] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[38] T. Mikolov, W.-t. Yih, and G. Zweig, "Linguistic regularities in continuous space word representations." in *hlt-Naacl*, 2013.

[39] Google Code, "word2vec," https://goo.gl/NBJgk1, 2017.

[40] 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.

[41] L. Yu, X. Luo, C. Qian, and S. Wang, "Revisiting the description-to-behavior fidelity in android applications," in *Proc. SANER*, 2016.

[42] "Android developers: Manifest.permission," https://goo.gl/vWoIU, 2017.

[43] "Android developer: Common intents," https://goo.gl/gHv9sF, 2017.

[44] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "Semantics-aware android malware classification using weighted contextual api dependency graphs," in *Proc. CCS*, 2014.

[45] M. Chen and X. Liu, "Predicting popularity of online distributed applications: itunes app store case analysis," in *Proc. iConference*, 2011.

[46] M. Gómez, R. Rouvoy, M. Monperrus, and L. Seinturier, "A recommender system of buggy app checkers for app store moderators," in *Proc. MOBILESoft*, 2015.

[47] D. Kong, L. Cen, and H. Jin, "Autoreb: Automatically understanding the review-to-behavior fidelity in android applications," in *Proc. CCS*, 2015.

[48] 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.

[49] 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.

[50] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[51] 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.

[52] 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.

[53] B. Bichsel, V. Raychev, P. Tsankov, and M. Vechev, "Statistical deobfuscation of android applications," in *Proc. CCS*, 2016.

[54] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *Proc. ICSE*, 2017.

[55] L. Xue, Y. Zhou, T. Chen, X. Luo, and G. Gu, "Malton: Towards on-device non-invasive mobile malware analysis for art," in *Proc. USENIX Security*, 2017.

[56] 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.

[57] 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.

[58] L. Yu, T. Zhang, X. Luo, and L. Xue, "Autoppg: Towards automatic generation of privacy policy for android applications," in *Proc. SPSM*, 2015.

[59] L. Yu, T. Zhang, X. Luo, L. Xue, and H. Chang, "Toward automatically generating privacy policy for android apps," *IEEE Trans. TIFS*, 2017.

[60] 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.

[61] 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.

[62] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *Proc. ASE*, 2013.

[63] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *Proc. ICSE*, 2010.

[64] 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.

[65] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a static noninteractive approach to feature location," *Trans. TOSEM*, 2006.

[66] L. Yu, X. Luo, X. Liu, and T. Zhang, "Can we trust the privacy policies of android apps?" in *Proc. DSN*, 2016.

[67] T. Zhang, J. Chen, H. Jiang, X. Luo, and X. Xia, "Bug report enrichment with application of automated fixer recommendation," in *Proc. ICPC*, 2017.