HAL
archives-ouvertes.fr

A Recommender System of Buggy App Checkers for App Store Moderators

Maria Gomez, Romain Rouvoy, Martin Monperrus, Lionel Seinturier

▶ **To cite this version:**

Maria Gomez, Romain Rouvoy, Martin Monperrus, Lionel Seinturier. A Recommender System of Buggy App Checkers for App Store Moderators. Danny Dig and Yael Dubinsky. 2nd ACM International Conference on Mobile Software Engineering and Systems, May 2015, Firenze, Italy. IEEE, <10.1109/MobileSoft.2015.8>. <hal-01117376>

HAL Id: hal-01117376
https://hal.inria.fr/hal-01117376

Submitted on 22 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Recommender System of Buggy App Checkers for App Store Moderators

María Gómez, Romain Rouvoy, Martin Monperrus and Lionel Seinturier
University of Lille / Inria
firstname.lastname@inria.fr

***Abstract*—The popularity of smartphones is leading to an ever growing number of mobile apps that are published in official app stores. However, users might experience bugs and crashes for some of these apps. In this paper, we perform an empirical study of the official Google Play Store to automatically mine for such error-suspicious apps. We use the knowledge inferred from this analysis to build a recommender system of buggy app checkers. More specifically, we analyze the permissions and the user reviews of $46,644$ apps to identify potential correlations between error-sensitive permissions and error-related reviews along time. This study reveals error-sensitive permissions and patterns that potentially induce the errors reported online by users. As a result, our systems give app store moderators efficient static checkers to predict buggy apps before they harm the reputation of the app store as a whole.**

I. Introduction

The popularity of smartphones is leading to a rapidly growing number of mobile apps, which are distributed through dedicated stores. According to a recent study, 70% of mobile application developers targeted the Android platform in Q3 2014 [32], and the *Google Play Store*[1] currently offers over $1,300,000+$ apps [3] for download.

However, there are many different app stores, depending on the platform, the country, and the business model. The role of an app store moderator is a complex one as s/he stands between app developers and end-users. On the one hand, end-users want high-quality apps, on the other hand, app developers want open platforms to express their creativity. Thus, app store moderators constantly make business and technical trade-offs to keep both users and developers happy and stay competitive in the ever-changing mobile computing landscape.

Any smartphone user knows that the quality of mobile apps greatly varies. For instance, we discovered a large number of exceptions raised in the wild on Android apps [18]. As Google Play Store lacks of source inspection before publishing apps, low-quality and malicious apps (*e.g.*, malware) easily reach the store. Recently, CHABADA [16] and WHYPER [26] introduced novel approaches to identify malware Android apps, which misbehave with regard to their description. However, these approaches do not cover another challenging threat for app stores: *buggy apps*, which disrupt user experience and inevitably store quality.

An app checker is a mechanism to predict whether an app is potentially error-prone when the developer uploads it. If an app store uses app efficient checkers, the overall quality of apps would raise. An app checker can be either dynamic or static, and in this paper, we focus on the latter. As all checkers (whether on mobile, desktop or server applications), app store checkers may suffer from false positives. In our case, it means that an app would be flagged as being potentially buggy while it is actually working fine. If an app store moderator enables checkers with too many false positives, it would be a deal-breaker for app developers. On the other end, if s/he disables all checkers, s/he risks hosting buggy apps, which would degrade the app store reputation.

To solve this problem, this paper proposes a recommender system of mobile app checkers for application moderators. This system enables app store moderators to make informed decisions on which checkers to enable based on quantitative measurements of their capabilities. Beyond existing researches on the use of permissions by Android apps [5], [13], [14], [29], we investigate the correlation between permission requests and errors reported by end-users. In particular, we perform an empirical study of the official Google Play Store to automatically mine permissions that correlate with bugs. From this study, we devise a taxonomy of permissions and a family of app checkers that can predict error-proneness upon submission of new apps.

To validate our approach, we identify a set of buggy apps using a data-mining technique and we compute the accuracy of each checker. From an analysis of the negative user reviews of $46,644$ apps, we build an oracle of error-proneness to measure the quality of our app checkers. These checkers exhibit a prediction accuracy between 61.42% to 61.96%. Note that we identify symptoms of permission-related bugginess, but we do not intend to point out the underlying causes of bugs. Our evaluation shows that app store moderators can be provided with large scale quantitative information regarding the relevance of enabling app checkers.

To sum up, the contributions of this paper are:

- We set up a *family of permission-based checkers for mobile apps* and integrate them into a *recommender system for app store* moderators;
- We propose a *novel evaluation scenario based on automatically mining* $1,400,000+$ *online user reviews* to identify a set of error-suspicious apps;
- We report on a *taxonomy of permissions that Android apps can request based on the study of* $46,644$ *apps available on Google Play Store. Using a grounded theory method, the taxonomy classifies permissions under* 4 *categories and* 5 *specific classes;*
- We conduct an *extensive analysis of the relevance of the recommender system of app checkers* and the performance of the considered permission-based checkers.

The remainder of this paper is organized as follows. Section II provides an overview of the Android permission

[1] https://play.google.com/store

model and a summary of our proposal. Section III describes the dataset we built for performing the empirical study. Section IV describes an approach to mine online user reviews and to identify apps with reviews discussing about errors. Section V proposes a taxonomy of permission types for Android apps. Section VI presents the recommender system of buggy app checkers proposed. Section VII evaluates the approach. Section VIII summarizes the related work. Finally, Section IX concludes the paper and outlines future work.

II. Overview

In this section, we first provide a brief background on the Android permission model, and we then present an overview of our proposal.

A. Background on Android Permissions

Android apps run in a sandbox—*i.e.*, an isolated area of the system that does not have access to system resources by default. Android provides a set of APIs to enable apps to access to data, resources, and privileged operations. The APIs are protected by a permission model: Apps must explicitly request the required permissions to access protected resources. Every app has a manifest file (`AndroidManifest.xml`) that summarizes essential information about the app. In particular, the manifest declares the *permissions* that the app requires for its execution, and a *package name* that serves as unique identifier for the app. When an app attempts to access a resource protected by a permission, the system checks the content of the manifest at runtime.

App developers are solely responsible for identifying the set of permissions required by their apps. Sometimes, developers fail assigning the adequate permissions, often because the available documentation of the Android permission system is incomplete [4], [13], [31]. Indeed, a permission error often results in an exception (*e.g.*, `SecurityException`) being thrown back to the app during execution [2], often leading to the app crash.

As a matter of clarification, we distinguish between two terms that are used along the paper: **Permission requests** refer to permissions declared by apps in their manifest file; **permission types** group available permissions that any app can request.

B. Overview of the Proposal

Our proposed approach consists of five steps, illustrated in Figure 1 together with the techniques and tools used: (1) We start by collecting Android app metadata from Google Play Store (Section III). (2) Then, for each app, we retrieve online reviews made by users in Google Play Store (Section IV-A). (3) Using *Latent Dirichlet Allocation* (LDA) on the reviews, we isolate the reviews discussing about errors, and we identify a set of error-suspicious apps (Section IV-C). (4) The fourth step consists in analyzing the existing permissions declared by apps and proposing a taxonomy of these permissions (Section V). (5) Taking as input the permission classification performed in step 4 and the error-suspicious apps in step 3, we automatically mine buggy-permission patterns using a data mining algorithm and build a family of buggy app checkers that constitutes the basis of a recommender system for assisting app store moderators (Section VI). App store moderators can activate these app checkers to score the quality of new submitted apps. Then, the store can notify developers about potential existence of bugs in the app before making it publicly available to users.

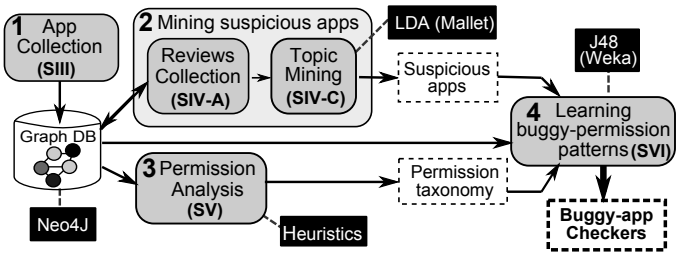

Fig. 1. Overview of our approach to identify buggy app checkers.

III. The App Analytics Framework

In this section, we describe the dataset we build to perform our empirical study as well as some key statistics.

A. Dataset Collection

We built a dataset that consists of a random sample of all the mobile apps available on the Google Play Store. The apps belong to the 27 different categories[2] defined by Google, and the 4 predefined subcategories (free, paid, new free, and new paid). For each category-subcategory pair (tools-free, tools-paid, sports-new_free, etc.), we collect a maximum of 500 samples[3], resulting in a median number of apps per category of $1,978$. In addition, for each app, we retrieve the following metadata: name, package, creator, version code, version name, number of downloads, size, upload date, star rating, star counting, and the set of permission requests.

Since the Google Play Store is continuously evolving (adding, removing and/or updating apps), we updated the dataset twice. We started by collecting available apps in November 2013 (D_0). Then, we updated the dataset in January 2014 (D_1) and March 2014 (D_2). In the rest of this paper, we use D_1 for performing the empirical study and D_2 for assessing the approach.

B. Statistics on the Collected Apps

Figure 2 depicts the evolution of our dataset along time in terms of additions, removals and updates of apps and permission types. D_1 contains $38,781$ apps requesting $7,826$ different permissions, while D_2 contains $46,644$ apps and $9,319$ different permission requests. The observed evolution tends to add new apps, more than updating or removing existing ones. In January, $15,023$ new apps appeared, $9,001$ apps ($33,13\%$) were updated, and $3,411$ apps ($12,55\%$) were removed. Similarly in March, $12,543$ new apps appeared, $8,970$ apps ($23,13\%$) were updated and $4,680$ apps ($12,07\%$) disappeared.

[2]Books&Reference, Business, Comics, Communication, Education, Entertainment, Finance, Games, Health&Fitness, Libraries&Demo, Lifestyle, Live Wallpaper, Media&Video, Medical, Music&Audio, News&Magazines, Personalization, Photography, Productivity, Shopping, Social, Sports, Tools, Transportation, Travel&Local, Weather, Widgets.

[3]This number is a constraint enforced by the Google Play Store.

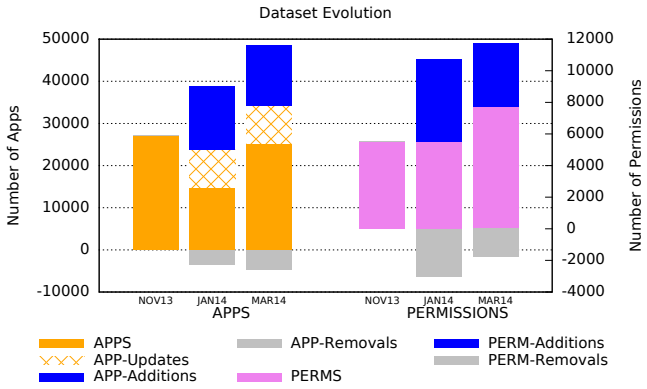

Fig. 2. Evolution of apps and permissions in Google Play Store.

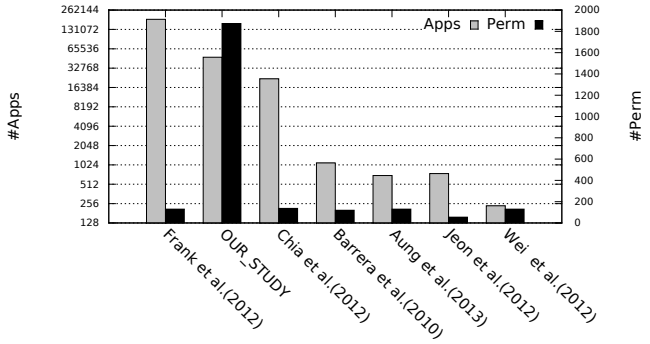

Fig. 3. Comparison of the number of apps and permission types used in our dataset and related work's datasets.

We observe that the app updates tend to rather add permissions than remove existing ones. Out of the $3,411$ apps that updated to a new version in January 2014, there are $2,677$ apps (78.48%) that update their set of permission requests. Specifically, there are $3,007$ permission request removals and $5,225$ permission request additions. This confirms that the trend of adding more and more permissions pointed out by previous studies [33] still holds. We have also observed that some apps are adding unnecessary permission requests when updating to a new version. For example, there are 74 apps adding the deprecated permission `[appPackage].permission.MAPS_RECEIVE` previously required to use the Google Maps API v1. The update of Google Play Services 3.1.59 (in July 2013) made this permission useless[4]. Furthermore, apps request 6 permissions by median, and at maximum, request 83 different permissions. In contrast, there are $2,592$ apps that request 0 permissions.

Figure 3 compares our dataset with the datasets used in the related work in order to evaluate its representativity. The *left Y axes* reports in logarithmic scale the number of different apps considered. Our dataset is the second largest among the datasets reported in the literature on this topic. The *right Y axes* focuses on the number of different permissions. Our dataset contains the largest number of permissions: $1,874$ different permissions. While related work only focus on Android permissions ($\approx$140 permissions), we consider other types of permissions available for Android apps, *e.g.* Google-specific permissions (cf. Section V for details).

C. Representing Apps Ecosystem as a Graph

To store the dataset, we created a *graph database* with *Neo4J*[5]. This dataset therefore consists of a graph describing the apps as nodes and edges. Graph databases provide a powerful and scalable data modelling and querying technique capable of representing any kind of data in a highly accessible way [28]. We chose a graph database because the graph visualization helps to identify connections among data (*e.g.*, clusters of apps sharing similar sets of permission requests). In particular, our dataset graph contains five types of nodes: APP nodes grouping the details of each app, PERMISSION nodes describing permission types, CATEGORY nodes describing app categories, SUBCATEGORY nodes describing app subcategories, and REVIEW nodes storing user reviews. Furthermore, there are four types of relationships between APP nodes and each of the remaining nodes *e.g.*, USES_PERMISSION relationships between APP and PERMISSION nodes. In total, our graph contains $1,449,361$ nodes and $1,901,703$ relationships among them. To extract statistics from the dataset, we can query the graph database using one of the available graph query languages (*e.g.*, Cypher, SPARQL, Gremlin). We chose *Cypher* [11], which is a widely used pattern matching language. This graph database is available for download [25].

[4]Google Maps Android API v2 Release Notes: https://developers.google.com/maps/documentation/android/releases?hl=en#july_2013

[5]http://www.neo4j.org

IV. Mining Error-Suspicious Apps

We are interested in revealing potential correlations between the usage of error-sensitive permissions and the emergence of bugs, thus we first have to mine buggy apps. To automatically distinguish buggy apps that tend to crash during execution, we analyze the reviews written by users in the Google Play Store. This provides us an oracle of error-proneness. However, it is inconceivable to manually analyze millions of reviews available in the Google Play Store. Then, we use unsupervised machine learning (specifically topic mining) to automatically identify clusters of error-related topics.

A. User Review Mining

For each app, we collect up to a maximum of the latest 500 reviews posted by users in the Google Play Store. For each review, we retrieve its metadata: *title*, *description*, *device*, and *version* of the app. None of these fields are mandatory, thus several reviews lack some of these details. As app updates might fix bugs from previous releases, some errors reported in the version 1.1 of a given app may be fixed in the new release 1.2. Therefore, from all the reviews attached to an app, we only consider the reviews associated with the latest version of the app —*i.e.*, we discard unversioned and old-versioned reviews. Thus, resulting in a corpus of $1,402,717$ reviews.

B. Topic Mining

To identify online reviews that treat topics related to bugs and errors, we extract topics discussed in these reviews using a *topic modelling* technique. Specifically, we apply the *Latent Dirichlet Allocation* (LDA) algorithm [7]. LDA identifies topics discussed in an entire corpus of documents and is able to

analyze a large volume of unlabelled text. For each document, LDA estimates a probability distribution over the mined topics. For example, one document may have a probability of 0.7 to relate to topic #2 and of 0.3 to belong to topic #1 (the sum is always 1). In this case, the document mostly belongs to topic #2. The LDA model takes as an input parameter the number of topics to extract. The rational of using topic models is to automatically (1) cluster reviews discussing about errors, and (2) extract keywords that characterize error themes without sketching them beforehand. We are particularly interested in topic models that reveal bug-related keywords, which were unforeseen initially.

To implement this approach, we use MALLET (*MAchine Learning for LanguagE Toolkit*) [23], a Java library that provides an implementation of LDA. We consider each single review of apps as a document, thus leading to $1,402,717$ documents. To reduce noise in the modeled topics, we filter out English stop words[6] from the entire corpus. In order to achieve a better precision, in a preliminary exploratory phase, we ran the LDA algorithm using different number of topics (*e.g.*, 20, 40, 60, and 100). We chose 100 topics since it generates fine-grained topics that help to identify bug-related issues.

C. Identifying Error-suspicious Apps

Our approach to identify error-suspicious apps comprises the following steps:

1) *Extract topics.* Using LDA, we mine the topics discussed in the app reviews. With our default parameter, we obtain 100 topics.
2) *Select relevant topics.* From the mined topics, we manually select the topics that are related to bugs and crashes. From our dataset, we automatically identify the following 3 relevant topics:

> **#24:** *fix update problem fixed bug issue crashes phone stars bugs plz pls time crashing problems working issues crash hope*
> **#48:** *work doesn doesnt won didn working open kit show load kat properly anymore sucks bad note worked android wont*
> **#81:** *app crashes force time open fix close closes won crashing work crash start freezes working times constantly closing doesn*

3) *Select error-related reviews.* To select error-related reviews, we analyze the probability distribution given by LDA for each review. Since a review can discuss several topics, we consider as error-related reviews those with at least 5% of its probability related to a topic discussing buggy issues. Examples of error-related reviews obtained[7]:

> Astoundingly buggy. *Get ready for the app to crash when you want it to work, and for it to linger in your notification bar even after you force close. I have no idea how such a phenomenal show has such a disappointing app.*
>
> Crashes all the time. *I love TAL but this app is horrible. Actually it is a good app but it literally crashes every time I put it to use. Sometimes it will open up and I can start a show and listen to the whole thing and when it is done it crashes. Sometimes it crashes as it is opening.*

4) *Select error-suspicious apps.* We consider as *error-suspicious apps*, the set of apps containing at least n error-related review. The minimum number of error-related reviews is a parameter to be decided by app store moderators depending on the quality they want to ensure in their stores, and the confidence they have on the users reporting reviews. In this work, we have considered the minimum possible evidence of problems ($n > 1$ review).

To sum up, by applying unsupervised machine learning on the reviews of apps, we automatically learn a class label for each app: *"Buggy"* or *"NonBuggy"*.

D. Empirical Results

Our process enables the identification of $10,658$ error-suspicious apps (27.48%) in our dataset. Figure 4 shows the number of error-suspicious apps identified in each category of the Google Play Store. For each category, we illustrate the distribution of error-suspicious apps according to the number of error-related reviews published by users. With 68.90% of apps having error-related reviews, the category GAME contains the largest number of error-suspicious apps. These data are in line with the results of the study performed by *Crittercism* based on observations in real devices, showing that gaming apps have the highest crash rate [10].

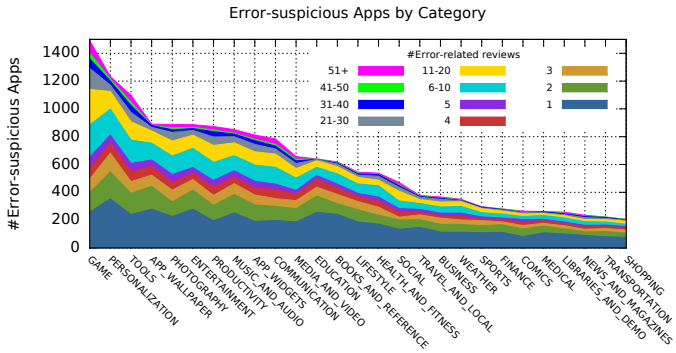

Fig. 4. Distribution of error-suspicious apps by categories in Google Play.

V. Analysing App Permissions

Now that we have identified error-suspicious apps, the next step is to analyze app permissions, in order to find correlations between permission and error-proneness. We start by proposing a taxonomy to classify the types of permissions that Android apps can request. This classification will later support the identification of error-sensitive permissions to propose a family of app permission checkers. To the best of our knowledge, we are the first to provide such a kind of taxonomy.

A. Taxonomy Creation

To create the taxonomy, we follow a grounded theory approach [30]. We first review the literature and define a draft

[6]We use the list of 524 common English stop words included in Mallet.

[7]The reviews belong to the app: *"This American Life"* (version 2.1.8), https://play.google.com/store/apps/details?id=org.prx.talbot. The dates of the reviews are 13-12-2013 and 18-12-2013, respectively.

taxonomy based on observations made by prior works [5], [13]. As a matter of example, Felt et al. [13] state that they consider neither developer-defined permissions nor Google-defined ones in their analysis. In addition, they reveal the existence of a set of non-official Android APIs. We include those types of permissions as categories in the taxonomy and we investigate them in depth. Another example is Barrera *et al.* [5], who manually observed that some apps in their dataset request non-existent permissions and deprecated permissions. We also consider information provided in the official Android documentation for developers[8] and in Android community forums (*e.g.*, Android Developers Blog[9]).

B. Permission Types Taxonomy

The taxonomy classifies permissions into four main *categories*:

1) *Android-specific permissions* refers to permission types to use the official APIs provided by the Android SDK[10]. For example, `android.permission.READ_CONTACTS` allows an app to read the user's contacts data.
2) *Google-specific permissions* groups the permission types to use the APIs provided by the Google Play Services SDK[11], which include services such as: *Google Play Licensing, Google Play In-app Billing, Google Maps, Google Cloud Messaging (GCM)*, etc. For example, the permission `com.android.vending.BILLING` is required to use the In-app Billing service that enables to sell products from inside an app.
3) *Vendor-specific permissions* encloses permission types to use the APIs included in SDKs provided by specific mobile device vendors (*e.g.*, Samsung, HTC) to create apps specialized for their devices. As an illustration, Samsung provides the *AllShare Framework SDK* that includes APIs[12] to implement convergence services (*e.g.*, media sharing, screen sharing). Apps using AllShare APIs should request the following permission: `com.sec.android.permission.PERSONAL_MEDIA`[13]. Another example is the permission `com.sonymobile.permission.CAMERA_ADDON` that is part of the *Sony Add-on SDK*[14] (Camera Add-on API) required to develop apps that can be launched from the native Xperia camera.
4) *Developer-specific permissions* finally refers to permission types defined by third-party developers. Android enables apps to define their own permissions to protect the functionality they expose to other apps. For example, `android.webkit.permission.PLUGIN` from *Adobe Flash plugin*.

We group permissions within each category around 5 *classes* of permissions:

(a) *Official permissions* groups permissions that are available in the public SDK's documentations for developers.
(b) *Removed permissions* refers to permissions that were available in previous versions of APIs and do not exist anymore.
(c) *Internal permissions* encloses permissions that are intended for internal use by the system and system apps.
(d) *Incorrect permissions* groups permissions defined erroneously by developers (*e.g.*, misspelled permissions).
(e) *Unclassified permissions* groups permissions that do not fit into any of the previous classes.

As we will show later, all the classes of permissions have been identified in the four categories of permissions.

C. Taxonomy Exploitation

We extracted all the permissions collected in our graph database and classify them according to the proposed taxonomy. We use heuristics, based on regular expression matching and string analysis, to catalog permissions under categories:

Heuristic 1 (Android-specific permissions): First, to identify permissions belonging to the Android-specific category, we filter permissions which follow the patterns: `android.permission.[*]` or `com.android.[*].permission.[*]`. From these permissions, we tag as *official* the permissions that match the list provided in the public Android documentation for developers according to Android 4.4 (API level 19)[15].

Heuristic 2 (Google-specific permissions): To identify permissions belonging to the Google-specific category, we filter permissions which follow the patterns: `com.google.android.[*].permission.[*]`, or `com.android.[*]`. The last pattern excludes the word ``permission'' as it matches the Android-specific pattern. Examples of Google-specific permissions that follow this pattern are `com.android.vending.BILLING` and `com.android.vending.CHECK_LICENSE` to use the billing and licensing services, respectively. From these permissions, we define the class of *internal Google permissions* with the permissions defined by Google apps (*e.g.*, Voice Search, GMail). These permissions follow the patterns `com.google.android.apps.[GoogleAppName].[*]` or `com.google.android.googleapps.permission.[*]`. For example, `com.google.android.voicesearch.SHORTCUTS_ACCESS` in the *Voice Search* app.

Heuristic 3 (Vendor-specific permissions): To identify permissions belonging to the Vendor-specific category, we filter permissions following the patterns `com.[vendor].[*].permission.[*]` or `android.permission.[vendor].[*]`. We compile a predefined list of available vendor names: sonymobile, htc, huawei, sec (Samsung Electronics Co), dell, and motorola.

Heuristic 4 (Classes refinement): We build lists of permissions to refine the permissions contained in each category under specific classes: official, removed, and internal. These lists are manually created by collecting documentation available in different sources. As a matter of example, the Android SDK contains a set of internal APIs that are intended for

[8] http://developer.android.com
[9] http://android-developers.blogspot.com
[10] http://developer.android.com/sdk
[11] http://developer.android.com/google
[12] Samsung documentation for developers: http://developer.samsung.com/develop
[13] "sec" comes from: *Samsung Electronics Co*
[14] Sony documentation for developers: http://developer.sonymobile.com
[15] http://developer.android.com/reference/android/Manifest.permission.html

exclusive use by the system and system apps. However, they are available from the Android source code and third-party apps can access internal APIs using Java reflection [13]. The internal APIs reside in the Android source code in the package `com.android.internal`, and in the other packages with the annotation `@hide`.

Heuristic 5 (Identifying incorrect permissions): We select all the permissions within a category that are considered as unclassifiable in previous classes. To automatically identify incorrect permissions, we compute the *Damerau-Levenshtein distance* [12] to measure the similarity between two input strings. We normalize the distance to the range $[0, 1]$ by dividing the distance by the length of the longest string, thus defined as:

$$dist_{NDL}(s1, s2) = \frac{dist_{DL}(s1, s2)}{\max(|s1|, |s2|)} \quad (1)$$

We compute the normalized Damerau-Levenshtein distance between each unclassified permission P and each official permission O_i. Finally, we set the similarity score of an unclassified permission as the minimum distance of all the obtained normalized Damerau-Levenshtein distances:

$$score(P) = \min\{dist_{NDL}(P, O_i)\}, i = \{0...n\} \quad (2)$$

Considering different similarity ranges, we observed 5 types of permission request mistakes in the Google Play Store:

Misspell. For example, requesting `android.permission.READ_CONCACTS` instead of `android.permission.READ_CONTACTS`.

Wrong prefix. Each permission is identified by a unique label. Typically, the label is defined by a string starting with a prefix (*e.g.*, `android.permission.`) followed by a constant in capital letters. We have identified a set of permissions defining incorrect prefixes, specifically two common mistakes:

Prefix absence. For example, `INTERNET` is missing the prefix `android.permission`. The rational for this mistake seems to come from the official Android documentation on permissions, which first provides a table with the permissions showing only the constant part.

Prefix confusion. For example, requests to `android.permission.SET_ALARM`, instead of `com.android.alarm.permission.SET_ALARM`.

Misuse. It is a special case of prefix misuse. Apps request as permissions other Android elements (*e.g.*, libraries, features). For example: `android.hardware.CAMERA`.

Lacking. Finally, we also observed some permissions that look like being incomplete: `android.permission.`

In addition to these mistakes, we build a cluster of *unclassified* permissions: all the permissions within a category that do not fit in any of the defined classes. Finally, the *developer-specific permissions* category covers permissions that do not fit into any of the previous categories.

We manually validated the resulting permission classification. We took a random sample of 300 permissions. Our manual inspection found 5.33% of false positives. Some false positives derive from the interpretation of permissions. For example, the permission `android.permission.SEND_MMS`[16] was tagged as *Misspelled Android* permission while it is obviously not. The nearest match is the permission `android.permission.SEND_SMS`. However, we consider the developer is trying to access to a functionality related with MMS instead of SMS, and it is not a misspelling.

Table I synthesizes the results of measuring the abundance of each class of the permission taxonomy in our dataset. For each category and class of the taxonomy, we show the number of permissions and permission requests. We observed that official Android-specific permissions are the most commonly requested ones among apps. Removed permissions is the second most popular class among the dataset.

D. Influence of Permissions on Error-related Reviews

We define as *error-sensitive permissions those permissions that are suspected to induce bugs. We consider as* error-sensitive permissions *the set of permissions belonging to the classes* removed*,* internal*,* incorrect *and* unclassified*, since* the use of non official permissions can be the source of problems*.* To clarify, not all the bugs are related to the suspicious permissions themselves (*e.g.*, missing permission, wrong permission due to a typo), but rather to the fact that some APIs for which the permission is requested are buggy or obsolete [22]. We study the correlation between permission requests and bugginess without claiming the underlying causes of bugs.

First, we ask the following research question: *To what extent apps which have error-related reviews request error-sensitive permissions?* In our dataset, there are 15,136 apps having error-related reviews. Out of these 15,136 apps, 1,220 request error-sensitive permissions (11.68%). This confirms that: 1) app bugs may be related to permissions and 2) there are many other reasons behind app bugs (*e.g.*, runtime errors, performance bugs). In the next section, we further explore the statistical relationship.

Thus, we raise the second research question: *Do apps that request error-sensitive permissions have error-related reviews?* In our dataset, there are 3,373 apps requesting error-sensitive permissions, and from these, there are 2,146 apps with at least one review (not necessarily bad). Out of these 2,146 apps, 1,068 have error-related reviews. This means that 49.77% of the apps requesting error-sensitive permissions can be considered as error-suspicious. This can be explained because some permissions that we have identified as error-sensitive do not really exhibit errors at runtime. For example, including some deprecated permissions in the manifest does not necessarily produce any side-effects. Furthermore, as pointed by prior studies, many apps declare more permissions than they actually use [13]. Now, we aim to identify potential permission patterns that correlate with bugs.

VI. Designing a Recommender System of Buggy App Checkers

Based on our identification of error-suspicious apps (cf. Section IV) and error-sensitive permissions (cf. Section V),

[16]This permission does not exist in the official Android documentation.

TABLE I. SUMMARY OF APPLYING THE PERMISSION TAXONOMY TO OUR GOOGLE PLAY STORE DATASET

	Official		**Removed**		**Internal**		**Incorrect**		**Unclassified**	
	#Perm	#Req	#Perm	#Req	#Perm	#Req	#Perm	#Req	#Perm	#Req
Android	137	245,274	19	1,699	50	1,040	144	561	152	1,240
Google	5	18,246	837	1,107	51	200	39	61	3	22
Vendor	184	1,514	-	-	-	-	-	-	-	-
Samsung	88	302	-	-	-	-	-	-	-	-
Sony	38	291	-	-	-	-	-	-	-	-
HTC	28	251	-	-	-	-	-	-	-	-
Dell	14	14	-	-	-	-	-	-	-	-
Motorola	12	649	-	-	-	-	-	-	-	-
Huawei	4	7	-	-	-	-	-	-	-	-
Developer	1,085	2,875	-	-	-	-	-	-	-	-
TOTAL	**1,411**	**267,909**	**856**	**2,806**	**101**	**1,240**	**183**	**622**	**155**	**1,262**

we now study the relationship between them in order to identify potential correlations between apps that use error-sensitive permissions and those reported as error-suspicious by end-users. We use the knowledge inferred from this study to propose buggy app checkers that could be embedded in app stores, such as the Google Play, to anticipate the emergence of app crashes in the wild.

A. Generating App Permission Checkers

Now, we ask the following research question: *Are there some common permission patterns in the presence of error-suspicious apps?* With the objective of mining common permissions in error-suspicious apps, we use supervised machine learning, and specifically a classifier where independent variables are permissions and the dependent variable is error-proneness.

1) Dataset Preprocessing: Let D be a dataset of apps, we represent each app (A) as a binary vector: $A = [p_1, ..., p_n, c_1, ..., c_n, L]$, where $p_i \in \{0, 1\}$, $c_i \in \{0, 1\}$, and $L \in \{Buggy, NonBuggy\}$. Each p_i depicts a permission available in our database, c_i represents a class of permission (according to our taxonomy), and L represents the class label learnt from its reviews (cf. Section IV-C). The value $p_i = 1$ indicates that the app requests the permission p_i, and p_i=0 indicates the absence of the permission. Similarly, $c_i = 1$ indicates that the app requests a permission that belongs to the class c_i (*e.g.*, `INCORRECT` class) in our taxonomy.

For the purpose of this analysis, we group all the classes of infrequent single permissions (*e.g.*, the misspelled ones) within a common abstract class. Indeed, a specific misspelled permission usually only appears in one app, but we are rather interested in knowing if the group of incorrect permissions is frequent in error-suspicious apps. There are also some permission names that are customized in each app. For example, apps requesting the GCM Google service must include a specific permission for receiving messages: `[appPackage].permission.C2D_MESSAGE`. The permission name must exactly match the pattern, but each app substitutes `[appPackage]` by its own package name in the manifest. This permission prevents other apps from registering and receiving their messages. Therefore, each specific GCM permission is only requested once at maximum. Moreover, we grouped all the official Android permissions in a single dimension[17]. We notice that the 10 most requested official Android permissions in our dataset are the same top requested permissions observed by other studies [1], [14][18].

Finally, the 4 classes of permissions considered are: `ANDROID-OFFICIAL`, `INCORRECT`, `GOOGLE-GCM`, and `GOOGLE-REMOVED` permissions. Each app is represented by a 1,563-dimensions vector, where the first $1,558$ dimensions represent single permissions (p_i), the following 4 dimensions refer to classes of permissions (c_i), and the last dimension is the assigned class label (L).

2) Mining Error-suspicious Permission Patterns: To identify permission patterns that correlate with bugs, we use *J48 Decision Tree* algorithm (a Weka[19] implementation of C4.5 [27]), which is a predictive machine-learning model. J48 predicts the value of a dependent variable based on the value of various attributes (independent variables) of the data. In our setup, the independent variables are permissions, and the dependent variable is bugginess. We choose J48 because it enables the direct extraction of rules to predict a label. From the resulting decision tree model, we extract permission patterns that lead to the label *'Buggy'*. We only consider presence of permissions in the patterns. However the absence of a permission also causes the app to crash. This kind of bug is partially handled by our approach, when the permission is missing because of an incorrect or incomplete declared permission. Furthermore, there already exists some developer support [6] to automatically check if apps declare all the permissions required to run.

The algorithm takes as input parameters a *confidence factor* and a *minimum number of objects* ($minNumObj$). The $confidencefactor$ limits the prediction error. For example, a confidence factor of 0.25 indicates that a permission pattern fails as maximum in the 25% of predictions. Thus, the lower the confidence factor, the more accurate the classifier. The $minNumObj$ parameter sets the minimum number of

[17]The request of Android permissions should not make the app crash, and we are rather interested in revealing the existence of patterns in error-sensitive permissions. First, we run the experiment considering all the official Android single permissions, but the most requested permissions appeared in many checkers, leading to high amount of irrelevant patterns.

[18]The most requested permissions in our dataset are: `INTERNET`, `ACCESS_NETWORK_STATE`, `READ_EXTERNAL_STORAGE`, `WRITE_EXTERNAL_STORAGE`, `READ_PHONE_STATE`, `ACCESS_WIFI_STATE`, `WAKE_LOCK`, `ACCESS_FINE_LOCATION`, `ACCESS_COARSE_LOCATION`, and `VIBRATE`

[19]Weka is an open-source Java library that provides implementations of several data mining algorithms: http://www.cs.waikato.ac.nz/ml/weka

instances that reach a leaf of the tree model. In our case it represents the minimum number of apps that must exhibit a permission pattern to be considered as a predictor of the class.

We train a J48 decision tree model using the dataset D_1 that contains $10,658$ apps labelled as `Buggy` and $11,539$ labelled as `NonBuggy`. Varying the $confidence$ and $minNumObj$ thresholds impacts the number of permission checkers obtained and their respective performance. We run the algorithm several times for different input values in order to identify the best calibration. We set cross-validation 10 folds. Then, we train the model with different confidence factors (ranging from 0.05 to 0.50 by increments of 0.05) with three different $minNumObj$ limits—*i.e.*, 100, 50 and 20 apps. Figure 5 reports on the results of the sensitivity analysis performed. The resulting family of checkers ranges from 4 to 12 different permission checkers.

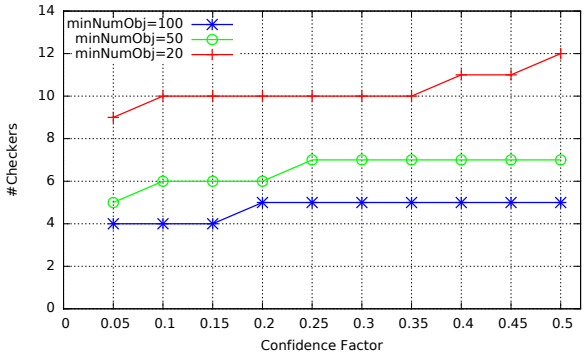

Fig. 5. Calibration of *confidence factor* and *minNumObj* parameters.

Table II details 4 families of permission checkers (F1–F4) obtained varying the $confidence$ factor (c) and $minNumObj$ (m) values. For example, the family $F2$ includes the 4 permissions pointed by $F1$ and one additional permission. In the family $F1$, we observe two official Google permissions: `CHECK_LICENSE` and `BILLING`. Contrary to our expectations, these official Google permissions are involved in some bugs. After searching in online forums for Android developers, we realize that there are many Android developers complaining because they have experienced crashes (due to security exceptions) in their apps after the update of Google Play services 4.3 (March 2014).

Note that the buggy permission checkers presented in this paper are not meant to be permanent. The app ecosystem is continuously evolving, and app store moderators can use our approach regularly (say monthly) for updating existing checkers, discovering new ones, and discarding outdated ones. The proposed system reveals interesting insights for isolating bugs in real devices. Nevertheless, we do not claim causality, but rather we suggest permissions that correlate with bugs.

The family of permission checkers obtained forms the knowledge of the recommender system. We evaluate the performance of the different families obtained, thus app store moderators can make informed decisions on which checkers to enable regarding their performance, in order to predict potential buggy-apps before they are published in the store.

VII. Evaluation

To assess the effectiveness of our approach, we investigate three main research questions:
RQ1: *What is the accuracy of the inferred checkers?*
RQ2: *To what extent checkers are able to flag new apps as buggy?*
RQ3: *What is the effect of removing error-sensitive permissions on error reviews?*

A. Evaluating the Accuracy of App Checkers

Checkers often have false positives, which in our case means that the app is flagged as buggy while still works fine. To sum up, if the moderator enables checkers with too many false positives, it can often flag apps as having bugs while they are working fine, then disturbing app developers. On the other end, if s/he disables all checkers, s/he risks hosting buggy applications, which disrupts app user experience and degrades the store reputation. For answering RQ1, we therefore evaluate the accuracy of each checker. To evaluate the accuracy of the proposed app checkers, we use the *Laplace expected error estimate* [9], which is computed as follows:

$$LaplaceAccuracy = (n_c + 1)/(n_{tot} + k) \quad (3)$$

where k is the number of classes in the domain, *e.g. Buggy/NonBuggy* ($k = 2$). n_{tot} is the total number of examples covered by the checker. In our case, is the total number of apps requesting the permissions captured by the checker. n_c is the number of examples in the predicted class by the checker. In our case, is the number of *Buggy* apps that request the permissions captured by the checker.

We set cross-validation 10-folds and we perform a sensitivity analysis of the accuracy of the checkers for different values of input parameters. Figure 6 shows the accuracy of the recommender system. The family of checkers exhibits an accuracy that ranges from 61.42% to 61.96%. We observe that the reported accuracy does not change significantly for different values of the parameters. Table II shows the *accuracy*

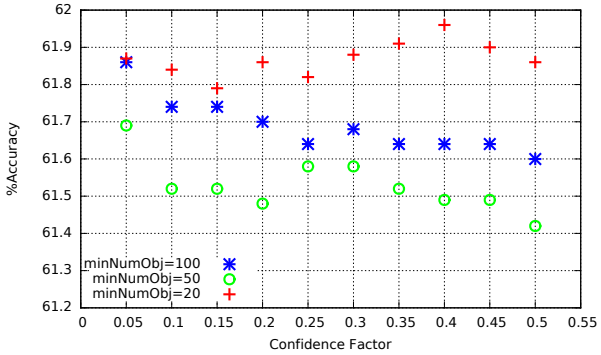

Fig. 6. Evaluation of the accuracy of the recommender system.

(acc.) values for some of the permission checkers identified. We also report the number of *Buggy* (B) and *NonBuggy* (NB) apps in the testing set which request the permissions captured by the checkers.

TABLE II. EVALUATION: APP PERMISSION CHECKERS AND THEIR ACCURACY.

	c	m		Error-sensitive Permission Checker	#B	#NB	Acc(%)

F1	0.1	100		`android.permission.WRITE_INTERNAL_STORAGE`
				`android.permission.ACCESS_SUPERUSER`
				`com.android.vending.CHECK_LICENSE`
				`com.android.vending.BILLING`
					2,888	1,789	**61.74**

F2	0.2	100	F1	`com.android.launcher.permission.READ_SETTINGS`	2,932	1,824	**61.64**
F3	0.2	50	F1	`android.permission.WRITE_OWNER_DATA` `com.sonyericsson.extras.liveware.aef.EXTENSION_PERMISSION`	2,928	1,834	**61.48**

F4	0.2	20	F3	`com.google.android.googleapps.permission.GOOGLE_AUTH`
				`android.permission.STORAGE`
				`com.android.email.permission.READ_ATTACHMENT`
				`com.google.android.gm.permission.READ_CONTENT_PROVIDER`
					2,986	1,841	**61.86**

B. Evaluating Checkers with New Apps

For answering RQ2, we use the dataset D_2 updated in March 2014 (cf. Section III-B). We select the subset of new apps that appeared in D_2 and did not exist in D_1. D_2 contains $6,783$ new apps (with reviews). We use the user reviews as our ground-truth for determining if an app is buggy or not[20]. Thus, we check if the D_2-version apps flagged as suspicious by our checkers have been reported as buggy by end users (after the publication of the version of D_2). Our system flagged $1,896$ apps as error-suspicious. Out of $1,896$ suspicious apps, 56% of apps were also reported as buggy by end users.

> The 56% of new apps flagged as suspicious by our checkers were also reported as buggy by end users *a posteriori*.

We further evaluate the performance of our approach by comparing it with alternatives. First, we compare our checkers built from permission patterns against: 1) a classifier that flags as suspicious apps the apps that request some single error-sensitive permissions (cf. section V-D) without learning permission patterns; and 2) a random classifier that flags suspicious apps randomly. As Table III shows, our approach learning permission patterns performs better than the others.

TABLE III. COMPARISON TO ALTERNATIVES

	Pattern-based	Single perm.	Random
Flagged apps	1,896	814	1,718
Bug-reported apps	**1,062 (56%)**	404 (50%)	815 (47%)

Remark that this measure is only an approximation, since we build an oracle of app bugginess from user reviews without performing in-depth analysis of apps. In fact, the lack of bug-related reviews does not necessarily imply that the app is bug-free, since an error can exist without being reported. Although this oracle is incomplete, it is a useful tool that helps store moderators to make informed decisions about app bugginess considering only information sources—*i.e.*, reviews—available on stores.

C. Impact of Removing Error-sensitive Permissions

To answer RQ3, we investigate if the apps that remove error-sensitive permissions (captured by the checkers) in the update get less error-related reviews. We noticed 30 apps (with error-related reviews) that remove error-sensitive permissions after updating. For these apps, we compute the percentage rate of error-related reviews before and after the update. Figure 7 illustrates the evolution of the reviews in these apps. We observe that after removing error-sensitive permissions, 22 apps (out of 30) remove error-related reviews. If the checkers had been enabled in the app store, all those apps would have been flagged as suspicious before publication.

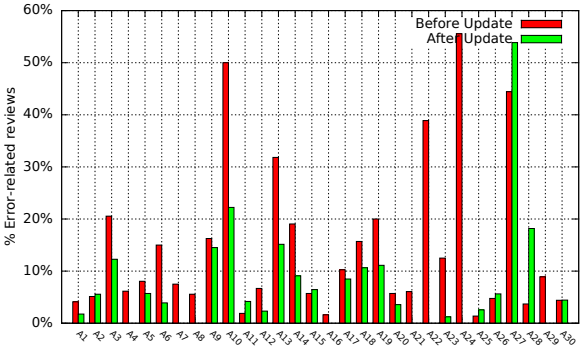

Fig. 7. Reviews evolution after removing error-sensitive permissions.

D. Discussion and Threats

We focus on permission requests and user reviews to build a family of buggy app checkers that can help app store moderators to score the quality of a submitted app. One threat to the internal validity of our study is that we have not analyzed the source or binary code of apps to ensure that the declared permissions are actually used.

We consider user reviews as a ground truth of bugginess, but we are aware that this measure is only an approximation. Users do not always report crashes when faced, and some bugs can be reported later in time. In addition, apps can crash for many reasons, not only due to permission-related issues. Currently, we are working on experiments in real devices to address this threat.

The conceptual foundations of our approach are independent of Android. We only need an oracle of bugginess and some observation features (in our case, the user reviews and the requested permissions, respectively). To gain in confidence in the external validity of our evaluation, more evaluations are needed on other platforms, using other oracles of bugginess and features.

[20]We are working on deploying experiments in real devices in the wild to complement the quantitative study performed in this paper

TABLE IV. SUMMARY OF THE APPROACH'S PARAMETERS

Parameter	Value
LDA model number of $topics$ threshold (IV-B)	100
Error-related reviews threshold in suspicious apps (IV-C)	> 1
Composition threshold in error-related reviews (IV-C)	0.05
J48 model $confidence$ factor (VI-A2)	0.4
J48 model $minNumObjects$ (VI-A2)	20

Regarding sensitivity, the proposed approach is based on predefined thresholds specified by various parameters. Table IV summarizes these parameters together with the reasonable values used in the experiments reported in this paper. The calibration of these parameters can impact the results and constitute a potential threat to validity. We have performed a sensitivity analysis for different values of these parameters. The reported accuracy does not change significantly for different values.

VIII. RELATED WORK

We divide related work into two major groups, and survey the literature in each group.

Android permission analysis. Frank *et al.* [14] propose a probabilistic model to identify permission patterns in Android and Facebook apps. They found that permission patterns differ between high-reputation and low-reputation apps. Barrera *et al.* [5] studied the permission requests made by apps in the different categories of the Google Play store by mapping apps to categories based on their set of requested permissions. They show that a small number of Android permissions are used very frequently while the rest are only used occasionally. In [20], Jeon *et al.* proposed a taxonomy that divides official Android permissions into four groups based on the permission behaviours (*e.g.*, access to sensors, access to structured user information). For each category, they propose new fine-grained variants of the permissions. Chia *et al.* [8] performed a study on Android permissions to identify privacy risks on apps. They analyze the correlation between the number of permissions requested by apps and several signals such as app popularity and community rating. Our taxonomy has a different goal, the aim of our classification is helping to identify error-sensitive permissions. All previous studies only focus on official Android-specific permissions, we also consider Google-defined, Vendor-defined and Developer-defined permissions in our analysis. Xu in his thesis [21] presents a systematic study of apps from Google Play Store. With reference to permissions, the study observed developer-related errors. However, Xu does not propose a technique for setting up buggy app checkers.

Our approach is also related to approaches that focus on identifying malicious behaviours in Android apps. CHABADA [16] proposes an API-based approach to detect apps which misbehave regarding their descriptions. CHABADA clusters apps with similar descriptions and identifies API usage outliers in each cluster. These outliers point out potential malware apps. Similarly to our approach, CHABADA uses topic modelling techniques. They apply LDA in the app descriptions for grouping apps with similar themes. In contrast, we use topic models on online user reviews (and not on app descriptions) to automatically identify buggy apps (as opposed to identify malware). Our aim is to identify permission patterns that correlate with bugs.

On the static analysis side, there are several works focusing on analyzing the source code and API calls in Android apps to check if the declared permissions are actually used: COPES [6], PScout [4], and Permlizer [34]. This is only indirectly related to checkers for identifying buggy apps.

Online user review analysis. Several approaches have analysed user reviews posted on the Google Play Store with different purposes. Ha *et al.* [17] manually analysed user reviews available on Google Play Store to understand what users write about apps. Performing this task manually becomes infeasible due to the large amount of available reviews. Chen *et al.* [24] present AR-Miner, a tool for mining reviews from Google Play Store and extract user feedbacks. They filter reviews that contain useful information for developers to improve their apps. As us, they use LDA to group the reviews discussing about the same thematics. Fu *et al.* [15] propose WisCom, a system to analyze user reviews and ratings in order to identify the reasons why users like or dislike apps. Iacob and Harrison [19] propose MARA, a prototype for extracting feature requests from online reviews of apps. First, they identify the sentences of the online reviews referencing to feature requests using a set of predefined rules. Finally, they use LDA for identifying the most common topics among the feature requests. As our approach, all these approaches use LDA for identifying topics discussed in the reviews. However, none of them use reviews as an oracle of error-proneness. We focus on reviews to identify a set of error-suspicious apps and to automatically learn a class label for each app.

Linares-Vásquez *et al.* [22] demonstrate a correlation between the stability of APIs and the success of Android apps. They also use user reviews for finding evidence of problems. They manually analyze the reviews of apps which contain low ratings. They look for reviews that contain some predefined error-related keywords. Their aim is to assess that the identified unsuccessful apps express the cause of the errors in their reviews. On the contrary, we automatically analyze the reviews of all apps, and identify error-related reviews using topic models.

IX. CONCLUSION AND FUTURE WORK

In this paper, we propose a recommender system that predicts potentially buggy apps by using the correlation between permission patterns and error-proneness. We collected $46,644$ Android apps from Google Play Store with the aim of studying the correlation between permission requests and the emergence of errors and crashes. We started by mining $1,400,000+$ online user reviews to identify error-related reviews that point out error-suspicious apps. We used unsupervised machine learning techniques to automatically identify error-suspicious apps available in the Google Play Store. We then propose a taxonomy of types of permissions requested by Android apps that supports the analysis of permissions. With the knowledge inferred from this analysis, we built permission-based checkers that are then ranked by our the recommender system. The permission-based checkers recommended by this system have an accuracy between 61.42% to 61.96%.

Currently, we are working on developing experiments in real devices in the wild with the set of error-suspicious apps identified. We will use the insights obtained in this study as starting point for isolating errors.

REFERENCES

[1] Android Observatory. http://androidobservatory.org.

[2] Android-System Permissions. http://developer.android.com/guide/topics/security/permissions.html.

[3] AppBrain. http://www.appbrain.com/stats/number-of-android-apps, Aug. 2014.

[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and Communications Security*, pages 217–228, 2012.

[5] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 73–84, New York, NY, USA, 2010. ACM.

[6] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automatically securing permission-based software by reducing the attack surface: An application to android. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE'12, pages 274–277, New York, NY, USA, 2012. ACM.

[7] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[8] P. H. Chia, Y. Yamamoto, and N. Asokan. Is this app safe?: A large scale study on application permissions and risk signals. In *Proceedings of the 21st International Conference on World Wide Web*, WWW'12, pages 311–320, New York, NY, USA, 2012. ACM.

[9] P. Clark and R. Boswell. Rule induction with cn2: Some recent improvements. In *Machine learningEWSL-91*, pages 151–163. Springer, 1991.

[10] Crittercism. Mobile experience benchmark. Technical report, Mar. 2014.

[11] Cypher. http://docs.neo4j.org/refcard/2.0.

[12] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, Mar. 1964.

[13] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[14] M. Frank, B. Dong, A. Felt, and D. Song. Mining permission request patterns from android and facebook applications. In *Proceedings of the 12th IEEE International Conference on Data Mining*, ICDM'12, pages 870–875, 2012.

[15] 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 *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD'13, pages 1276–1284, New York, NY, USA, 2013. ACM.

[16] A. Gorla, I. Tavecchia, F. Gross, and A. Zeller. Checking app behavior against app descriptions. In *ICSE'14: Proceedings of the 36th International Conference on Software Engineering*, Hyderabad (India), 31 May - 7 June, 2014.

[17] E. Ha and D. Wagner. Do android users write about electric sheep? examining consumer reviews in google play. In *Proceedings of the IEEE Consumer Communications and Networking Conference*, CCNC'13, pages 149–157, 2013.

[18] N. Haderer, R. Rouvoy, and L. Seinturier. Dynamic deployment of sensing experiments in the wild using smartphones. In *Proceedings of the International Conference on Distributed Applications and Interoperable Systems*, DAIS'13, pages 43–56, 2013.

[19] C. Iacob and R. Harrison. Retrieving and analyzing mobile apps feature requests from online reviews. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR'13, pages 41–44, Piscataway, NJ, USA, 2013. IEEE Press.

[20] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein. Dr. android and mr. hide: Fine-grained permissions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 3–14, New York, NY, USA, 2012. ACM.

[21] Liang Xu. *Techniques and Tools for Analyzing and Understanding Android Apps*. PhD thesis, University of California, 2013.

[22] M. Linares-Vásquez, G. Bavota, C. Bernal-Crdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. API change and fault proneness: A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 477487, New York, NY, USA, 2013. ACM.

[23] A. K. McCallum. MALLET: a machine learning for language toolkit. http://mallet.cs.umass.edu, 2002.

[24] Ning Chen, Jialiu Lin, Steven C. H. Hoi, Xiaokui Xiao, and Boshen Zhang. AR-Miner: mining informative reviews for developers from mobile app marketplace. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE'14, 2014.

[25] Online appendix. https://sites.google.com/site/androidbuggyappcheckers.

[26] R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie. WHYPER: Towards automating risk assessment of mobile applications. In *USENIX Security*, volume 13, 2013.

[27] J. R. Quinlan. *C4. 5: programs for machine learning*, volume 1. Morgan kaufmann, 1993.

[28] I. Robinson, J. Webber, and E. Eifrem. *Graph Databases*. O'Reilly, June 2013.

[29] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen. Asking for (and about) permissions used by android apps. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR'13, pages 31–40, Piscataway, NJ, USA, 2013. IEEE Press.

[30] A. Strauss and J. Corbin. Grounded theory methodology: An overview. In N. K. Denzin and Y. S. Lincoln, editors, *Handbook of Qualitative Research*, pages 273–285+. Sage Publications, Thousand Oaks, CA, 1994.

[31] T. Vidas, N. Christin, and L. Cranor. Curbing android permission creep. In *Proceedings of the Web 2.0 Security and Privacy 2011 workshop*, W2SP'11, Oakland, CA, May 2011.

[32] VisionMobile. Developer economics q3 2014: State of the developer nation. Technical report, July 2014.

[33] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 31–40, New York, NY, USA, 2012. ACM.

[34] W. Xu, F. Zhang, and S. Zhu. Permlyzer: Analyzing permission usage in android applications. In *Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering*, ISSRE'13, pages 400–410, 2013.