What Permissions Should This Android App Request?

Lingfeng Bao*, David Lo†, Xin Xia*♯, and Shanping Li*
*College of Computer Science and Technology, Zhejiang University, China
†School of Information Systems, Singapore Management University, Singapore
{lingfengbao, xxia, shan}@zju.edu.cn, davidlo@smu.edu.sg

***Abstract*—As Android is one of the most popular open source mobile platforms, ensuring security and privacy of Android applications is very important. Android provides a permission mechanism which requires developers to declare sensitive resources their applications need, and users need to agree with this request when they install (for Android API level 22 or lower) or run (for Android API level 23) these applications. Although Android provides very good official documents to explain how to properly use permissions, unfortunately misuses even for the most popular permissions have been reported.**

Recently, Karim *et al*. propose an association rule mining based approach to better infer permissions that an API needs. In this work, to improve the effectiveness of the prior work, we propose an approach which is based on collaborative filtering technique, one of popular techniques used to build recommendation systems. Our approach is designed based on the intuition that apps that have similar features – inferred from the APIs that they use – usually share similar permissions. We evaluate the proposed approaches on 936 Android apps from F-Droid, which is a repository of free and open source Android applications. The experimental results show that our proposed approaches achieve significant improvement in terms of the precision, recall, F1-score and MAP of the top-k results over Karim *et al*.'s approach.

***Index Terms*—Android, Permission Recommendation, Association Rule, Collaborative Filtering**

I. Introduction

Android has become a very popular platform that dominated the smartphone market with a market share of 82.8% in the second quarter of 2015 [1]. More and more Android applications (also referred to as "apps") are produced by thousands of developers. In the first quarter of 2016, there are about 1,900,000 apps in *Google Play* [2]. Meanwhile, the huge number of Android apps attracts more attackers to develop malicious apps, which are often designed to steal sensitive data, such as private credentials and financial information.

In order to decrease the threats that Android apps pose to the privacy and security of their users, Android provides a unique permission mechanism to control access of third party applications to sensitive resources, such as the user's contact list, camera, network, etc. Android requires app developers to write the needed permissions explicitly in a config file named `AndroidManifest.xml`. Hence, Android app developers not only need to know how to use APIs to implement certain features of an application, but also the corresponding permissions. For example, if an app requires internet access, the developer not only needs to know the network APIs, such as "android.net.ConnectivityManager" and "java.net.Socket", but also the corresponding permissions namely ACCESS_NETWORK_STATE and INTERNET which need to be written to the AndroidManifest.xml file.

To reduce security risk, Android official document[1] mentions that it is better for developers to minimize the number of permissions that their apps request. However, often it is not easy for Android app developers to decide which permissions are needed. There are 151 system-level permissions available and over 4,000 classes in Android library. Moreover, the official Android documentation for API classes and permissions is incomplete [3], [4]. Stevens *et al.* have also shown that there exists many misuses even for the most popular permissions [5]. Hence, there is a need for a recommendation system that can help developers decide suitable permissions for their apps.

To address this need, some researchers have proposed some tools to recommend permissions by tracing APIs to specific permissions. *Stowaway* extracts APIs used in apps through static analysis and builds a permission map through dynamic analysis of the Android OS/stack [3]. In a later work, Au *et al.* propose *PScout* which maps permissions to APIs based on the static analysis of the Android OS [4]. *Androguard*[2] builds upon *PScout*'s methodology to also output likely API to permission mappings for a given app [6]. Unfortunately, these program analysis approaches are not perfect and many wrong recommendations are made.

Recently Karim *et al.* process likely API to permission mappings output by *Androguard* using association rule mining, a popular data mining technique, to recommend the required permissions of an app [7]. An experiment is conducted on 600 apps from F-Droid [3] and the results show that their proposed approach named *APMiner* outperforms *PScout* and *Androguard* [7]. Although their approach outperforms tools which are based on static analysis, its effectiveness is not optimal. Moreover, there are many other algorithms proposed in the recommendation system area which could have been applied to recommend permissions for Android apps. Hence, in this paper, we want to investigate collaborative filtering which has been widely adopted in many recommendation

♯Corresponding author

[1]https://developer.android.com/training/articles/security-tips.html#RequestingPermissions
[2]https://github.com/androguard/androguard
[3]https://f-droid.org/

978-1-5090-4517-4/16 $31.00 © 2016 IEEE
DOI 10.1109/SATE.2016.13

systems [8] and compare its effectiveness with Karim *et al.*'s approach which is based on association rule mining. We refer to the best performing variant of *APMiner* as $APRec^{RULE}$ to make this name be consistent with the name of our proposed approach in this paper.

Our approach, which is based on collaborative filtering, refers to as $APRec^{CF}$. The intuition of using collaborative filtering is that apps which use similar APIs, usually support similar features, so the required permissions are usually similar too. Hence, $APRec^{CF}$ first finds a list of most similar apps to a target app, and then recommend permissions based on the used permissions of these similar apps. We measure similarity of two apps based on the APIs used by the apps.

We evaluate the proposed permission recommendation approach on 936 open source Android apps from F-droid which have corresponding *Github* repositories. We measure the effectiveness of our approach in terms of precision, recall, F1-score and Mean Average Precision (MAP) of the top-k recommendations. The experiment results show that $APRec^{CF}$ outperforms $APRec^{RULE}$.

The remainder of the paper is organized as follows. We first describe the details of the two permission recommendation approaches in Section II. Our experiment results are presented in Section III. Related work is briefly reviewed in Section IV. Finally, we conclude and outline potential future directions in Section V.

II. Approach

In this section, we first describe how to extract the data including APIs and permissions from Android apps. We then present the details of two proposed recommendation approaches which are based on different data mining techniques, i.e., $APRec^{RULE}$, $APRec^{CF}$.

A. Data Collection and Processing

We use Android applications from F-Droid which is a catalogue of free and open source applications for the Android platform as data for this study. In total, we find 1,993 apps on F-Droid[4]. The Android applications on F-Droid are hosted in different platforms, such as *Github*, *Bitbucket*, *Google Code*, etc. In this study, we only consider applications whose source code is hosted on *Github*, since most of projects on *Github* provide a readme file and we can get the readme file easily by very convenient REST APIs provided by *Github*. Among the 1,993 apps, 936 of them put their source code in *Github* and have readme files.

Given an Android app, its permissions are specified in the manifest file, i.e. `AndroidManifest.xml`, which is located at the root level of the app, while its APIs can be found in its Java source code files, declared in import statements. To extract APIs from Java source code, we first transform the source code into a single xml file using *srcML* which is a lightweight static analysis tool [9]. Then we can extract permissions and APIs using an XML processing tool very easily. We only consider the API class names in the Android software stack and Java standard libraries, such as *android.content.ContentResolver* and *java.net.URL*, and ignore user-defined classes.

[4]https://f-droid.org/wiki/page/Repository_Maintenance

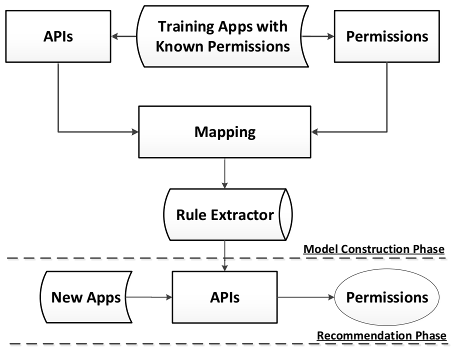


Fig. 1. The Framework of $APRec^{RULE}$

We use *Androguard* to obtain likely mapping of each API classes and permissions. This forms the transactions that are used by $APRec^{RULE}$. The list of APIs that are used by an app is transformed into a feature vector and input to $APRec^{CF}$.

B. $APRec^{RULE}$ Approach

Figure 1 presents the process of $APRec^{RULE}$. $APRec^{RULE}$ is an implementation of the best performing variant of *APMiner* referred to as *FilteredMiner*' in the original paper [7]. *APMiner* utilizes association rule mining technique and outperforms the prior state-of-the-art approaches: *Androguard* and *PScout*. Hence, we use $APRec^{RULE}$ as a baseline to compare our proposed approach. Each input transaction of $APRec^{RULE}$ contains a single permission and several APIs which can be traceable to the permission. These transactions are the output of *Androguard*.

$APRec^{RULE}$ works in two phases: model construction and recommendation phase. In the model construction phase, $APRec^{RULE}$ takes as an input a set of transactions that is generated by running Androguard on a training set of apps with known permissions. A sub-component of $APRec^{RULE}$ named $RuleExtractor$ employs association rule mining to mine API-permission rules (e.g., $APIs \implies Permission$), referred to as $APRules$. In the recommendation phase, given a set of APIs of a new app $currentAPIs$, a rule matches $currentAPIs$ if its precondition is a subset of $currentAPIs$. $APRec^{RULE}$ then recommends permissions, based on the post-conditions of the matching rules.

We assign a score to assess the probability of a permission for which an app requires. The rule-based recommendation score for a permission P is the sum of confidence of any matching rule whose post-condition is P. This score is computed by the following formula:

$$RecScore^{RULE}(P) = \sum_{R \in RMatched(P)} conf(R)$$

In the above equation, $RMatched(P)$ is the set of rules whose pre-conditions is a superset of $currentAPIs$

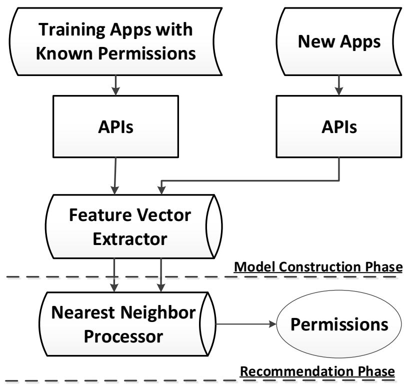


Fig. 2. The Framework of $APRec^{CF}$

and whose post-condition is a permission P. If the set $RMatched(P)$ is empty, the recommendation score of P is 0. Next, we will normalize $RecScore^{RULE}$ to make the value ranges from 0 to 1. The permissions with the highest recommendation scores are deemed to be the most appropriate permissions based on the mined association rules.

C. $APRec^{CF}$ Approach

Figure 2 presents the process of $APRec^{CF}$. $APRec^{CF}$ utilizes a collaborative filtering technique to recommend permissions for Android apps. $APRec^{CF}$ use *cosine similarity* to calculate the similarity between two apps. For each Android app, $APRec^{CF}$ uses the used APIs to form a feature vector, then get the nearest neighbor apps from the training dataset to perform permission recommendation.

$APRec^{CF}$ recommends permissions based on those that are used by similar Android apps, following a nearest-neighbor-based collaborative filtering approach. We measure the similarity of two apps based on their set of commonly used APIs. $APRec^{CF}$ works in two phases: model construction and recommendation phase. It consists of the $FeatureVectorExtractor$ component and the $NearestNeighborProcessor$ component. The first component is called in the model construction phase, while the latter is called in the recommendation phase.

1) $FeatureVectorExtractor$: This component converts the list of APIs used by each app in a training set into a feature vector. Let $allAPIs$ be the set of all APIs arranged in alphabetical order of their names. Each API can then be assigned a unique index in $allAPIs$ and referred to as $allAPIs[i]$. The feature vector of app A, denoted as $V(A)$, is defined as follows:

$$V(A) = (ind(allAPIs[0], A), ..., ind(allAPIs[|allAPIs|], A))$$

where $ind(I, A) = 1$ if A uses API I, and $ind(I, A) = 0$, otherwise.

2) $NearestNeighborProcessor$: Given a new app, $NearestNeighborProcessor$ first converts the list of APIs used in the app into a feature vector in the same manner as is done by the $FeatureVectorExtractor$. It then calculates the distance between this feature vector and the feature vectors of apps in the training set. In this study, we use *cosine similarity* to compute the distance.

The *cosine similarity* score of a new app A and an existing app B in the training set is calculated as follows:

$$S_{cosine}(A, B) = \frac{V(A) \cdot V(B)}{|V(A)||V(B)|}$$

In the above equation, $\cdot$ denotes dot product, and $|V(i)|$ denotes the size of a vector $V(i)$, which is defined as the square root of the sum of the squares of its constituent elements. Cosine similarity ranges from 0 to 1, since the term frequencies cannot be negative.

The higher the similarity score is, the more similar an app in the training set is to the new app. We rank apps in the training data based on their similarity scores. Then, we pick top-n apps with the highest similarity score as the nearest neighbors of the new app. The next step is to compute a recommendation score for each permission. Given a permission P, the collaborative filtering-based recommendation score of an app A is calculated as follows:

$$RecScore^{CF}(P) = \sum_{B_i \in Nearest} S_x(A, B_i), \text{ if } P \in B_i$$

In the above equation, $Nearest$ is the nearest neighbors, $P \in B_i$ means app B_i has permission P and S_x is one of three similarity scores. Then, we normalize the recommendation score. The permissions with the highest recommendation scores are the most appropriate permissions based on the collaborative filtering.

III. Results

In this section, we describe the experiment setup, followed by our evaluation metrics. We then present our research questions and the results of our experiments. Finally, we discuss some threats to validity.

A. Experiment Setup

The dataset in this evaluation is 936 Android apps from F-Droid. The apps in the dataset have 4.45 ± 2.57 (mean±standard deviation) permissions and 104.01 ± 83.59 APIs in average. We also use *Androguard* to filter the APIs which are not traceable to the used permissions. Thus, the apps have 8.31 ± 5.35 traceable APIs in average.

In our study, we use a fast Apriori [10] algorithm implementation [11] for association rule mining. Karim *et al.* has shown that the best performing variant of their approach (referred to as $APRec^{RULE}$ in this paper) has the best performance when the confidence is set to 0.4, so we set confidence value to 0.4 too. For the collaborative filtering approach $APRec^{CF}$, the default numbers of nearest neighbors of them are all set to 10.

The experimental environment is a 64-bit, Intel(R) Core(TM) i7-6500 2.50GHz computer with 8GB RAM running Windows 10

B. Evaluation Metrics

To evaluate the three permission recommendation approaches in this study, we use several well-known evaluation metric: precision@k, recall@k, F1-score@k and the Mean Average Precision (MAP) which have been used as yardsticks in many studies, e.g., [12], [13], [14]. Consider m Android apps in the testing dataset that should receive permission recommendation. For each app A_i, let the actual set of permissions of A_i be P_i^t, and N_i^k be the number of permissions that are correctly recommended in the top-k permissions P_i^k recommended by a permission recommendation system. The precision@k is the ratio of N_i^k over k, i.e., $(precision@k)_i = \frac{N_i^k}{k}$, while recall@k is the ratio of N_i^k over the actual number of permissions, i.e., $(recall@k)_i = \frac{N_i^k}{|P_i^t|}$ Then the F1-score@k is a summary measure that combines both precision@k and recall@k, i.e.,

$$(F1-score@k)_i = 2 \times \frac{(precision@k)_i \times (recall@k)_i}{(precision@k)_i + (recall@k)_i}$$

Finally, for m apps in a testing dataset, we calculate the average precision@k, recall@k and F1-score@k.

Since a permission recommendation system returns a ranked list of permissions, it is desirable to also consider the order in which the returned permissions are presented. Hence, we also use Mean Average Precision (MAP) which is one of the most popular measures to evaluate ranked retrieval results as an evaluation metric. MAP is known to be a stable [15] and highly informative [16] measure. In this study, Average Precision (AP) is the average of precisions computed at the point of each of the permissions that are correctly recommended in the ranked permission list. It is computed as follows:

$$AP = \frac{\sum_{k=1}^{n}(P(k) \times rel(k))}{\text{the number of actual permissions}}$$

where k is the rank in the sequence of recommended permissions, n is the number of recommended permissions, $P(k)$ is the precision at cut-off k in the list and $rel(k)$ is an indicator function which is equal to 1 if the item at rank k is a permission that the target app uses, and 0 otherwise. Then, for the m apps in the testing dataset, MAP is calculated as follows: $MAP = \frac{\sum_{i=1}^{m} AP_i}{m}$

C. Research Questions

RQ1: How effective are our permission recommendation approach based on collaborative filtering? How much improvement can our approach achieve over the baseline approach?

Motivation. The better performance $APRec^{CF}$ have, the more benefit they would give to their users. Thus, in this research question, we evaluate the effectiveness of $APRec^{CF}$ and compare them with the baseline approach $APRec^{RULE}$.

Approach. To answer RQ1, we use 10 fold cross validation to compute the top-k precision, recall and F1-score ($k = (1, 2, ..., 10)$) and MAP to evaluate the performance of each permission recommendation approach.

To check if the difference between the results of the baseline $APRec^{RULE}$ and $APRec^{RULE}$ is significant, we apply the Wilcoxon signed-rank test [17] at 95% significance level on 10 paired data which represents the results of 10 fold cross validation of compared approach. We also use Cliff's delta (δ) [18], which is a non-parametric effect size measure that quantifies the amount of difference between the baseline $APRec^{RULE}$ and $APRec^{RULE}$. Table I describes the meaning of different Cliff's delta values and their corresponding interpretations.

TABLE I
CLIFFS DELTA AND THE EFFECTIVENESS LEVEL [18]

Cliff's Delta ($\|\delta\|$)	**Effectiveness Level**
$\|\delta\| < 0.147$	Negligible
$0.147 \leq \|\delta\| < 0.33$	Small
$0.33 \leq \|\delta\| < 0.474$	Medium
$0.474 \leq \|\delta\|$	Large

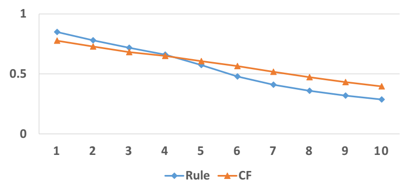
TABLE II
THE RESULTS OF PRECISION@K, RECALL@K, F1-SCORE@K (K=5, 10) AND MAP

	$APRec^{RULE}$	$APRec^{CF}$
Precision@5	0.5739	0.6064
Recall@5	0.6322	0.7487
F1-score@5	0.5671	0.6183
Precision@10	0.2869	0.3960
Recall@10	0.6322	0.9172
F1-score@10	0.3759	0.5176
MAP	0.6275	0.7651

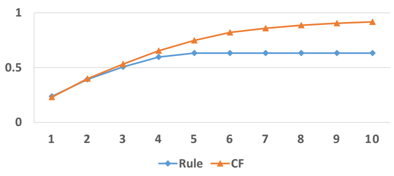
Results. The results of top-k precision, recall and F1-score are presented in Figure 3(a), 3(b) and 3(c), respectively. In these figures, 'Rule' and 'Cosine', represent $APRec^{RULE}$, $APRec^{CF}$, respectively. From these figures, we can see that the precisions of two approaches decrease when the number of recommended permissions (i.e. top-k) increases while the recalls increase when the number of recommended permissions increases. This results make sense. For precision, the permissions with high recommendation scores are more likely to be correct, so the precision is high if k is small. But when k increases, more permissions are wrongly recommended. For recall, the more permissions are recommended, the higher the recall is.

From Figure 3(a), we can see that when k is small (from 1 to 4), the precision of baseline $APRec^{RULE}$ is higher than $APRec^{CF}$, but the difference is very small. But the recalls and F1-scores of $APRec^{RULE}$ is almost the same as $APRec^{CF}$ when k is small, see Figure 3(b) and 3(c). However, when k is bigger than 5, the precisions, recalls and F1-scores of $APRec^{RULE}$ are all smaller than $APRec^{CF}$. The precisions of $APRec^{RULE}$ decrease more rapidly than $APRec^{CF}$ and its recalls almost do not increase when k exceeds 5. As F1-score is a harmonic mean of precision and recall, so the F1-score of $APRec^{RULE}$ also decreases rapidly. But there are small difference on the results of precision, recall and F1-score for the $APRec^{CF}$ when k is varied from 1 to 10. In summary, we can see our proposed approach $APRec^{CF}$ in this paper outperforms the baseline approach $APRec^{RULE}$.

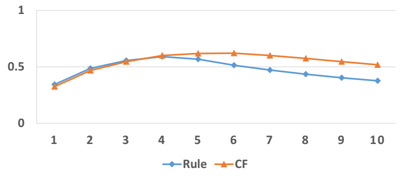
We present the detailed top-k (k=5,10) precision, recall, F1-score and MAP for two approaches in Table II. From this table, we can see all metrics of $APRec^{RULE}$ are smaller than



(a) Precision



(b) Recall



(c) F1-score

Fig. 3. The Results of Top (1-10) Precision, Recall and F1-score

TABLE III
P-VALUE AND CLIFF DELTA (δ) BETWEEN $APRec^{CF}$ AND $APRec^{RULE}$

	p-value	δ
Precision@5	6.43E-03	0.49
Recall@5	< 0.0001	0.89
F1-score@5	< 0.0001	0.80
Precision@10	< 0.0001	0.90
Recall@10	< 0.0001	0.90
F1-score@10	< 0.0001	0.90

those of $APRec^{CF}$. And the improvements in terms of top-10 metrics are larger than those in terms of top-5 metrics. The improvements on precision@10, recall@10 and F1-score@10 are more than 10%, 25%, and 13% respectively. The MAP of $APRec^{CF}$ is larger than that of $APRec^{RULE}$ by ∼14%.

Table III represents the p-values and cliff's delta values for $APRec^{CF}$ with the baseline $APRec^{RULE}$ in terms of precision@k, recall@k, F1-score@k (k=5, 10) and MAP. In this table, all of the p-values are less than 0.05 and all of the Cliff's delta values are in large effectiveness level which means the improvement of our approach over the baseline $APRec^{RULE}$ is significant.

RQ2: How does the size of training data affect the results of the permission recommendation approaches?

Motivation. We want to investigate whether different sizes of training data affect the performance of the permission recommendation approaches investigated in this study.

Approach. We run n fold cross validation to evaluate the performance of each approach, where n ranges from 2 to 10. As we reduce the value of n, we reduce the amount of training data. We evaluate the results in terms of F1-score@5, F1-score@10 and MAP.

Results. Figure 4(a), 4(b) and 4(c) present the F1-score@5, F1-score@10 and MAP of these permission recommendation approaches for different n fold cross validation, respectively. We notice that both the F1-score@5, F1-score@10 and MAP change only very little when we vary the size of the training dataset for two approaches in this study. Hence, we find that the permission recommendation approaches evaluated in our study perform well across a wide range of training data sizes.

D. Threats to Validity

One of threats to internal validity relates to errors in our code and experiment bias. We have double-checked our code, still there could be errors that we did not notice.

One of threats to external validity is the dataset used in our study. We have analyzed 936 open source Android apps from F-Droid. In the future, we plan to consider more open source Android apps even closed source apps to reduce this threat to validity. Closed source apps, such as those distributed on GOOGLE PLAY, require reverse engineering which can be performed by existing tools (e.g., *Androguard*) to extract the API usage. Another threat to external validity is that not all permissions are covered in out study. Out of the 151 system-defined permission in Android, 45 permissions are used in our dataset. We also do not consider the customized permissions.

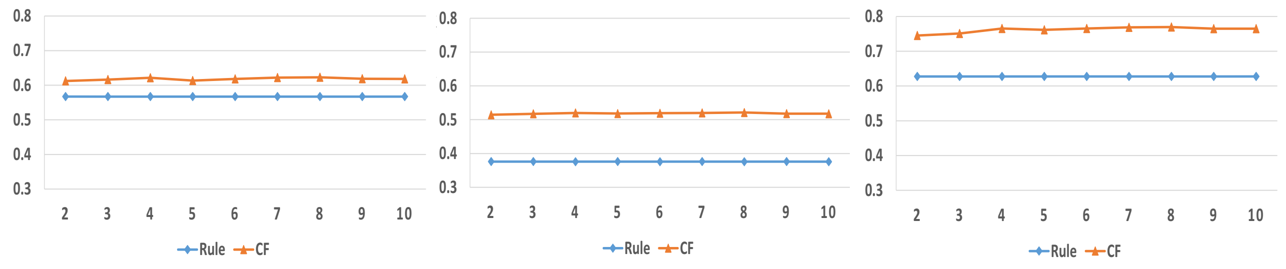
Threats to construct validity refers to the suitability of our evaluation measures. We use top-k precision, recall and F1-score, and MAP which are also used by many prior automated software engineering studies [12], [13], [14].

IV. RELATED WORK

Our work is inspired by the work of Karim *et al.* [7], which uses association rule mining technique to recommend permissions. The extracted rules in their study are based on the co-occurrence of Android APIs and permissions. To our best knowledge, there are no other studies that use recommendation system algorithms to predict possible required permissions for an app. In our study, we propose an approach based on collaborative filtering and find that our approach achieve better performance than that of the association rule mining approach.

Many researchers have proposed different approaches to identify the mappings between APIs and permissions. Some tools (e.g. *Stowaway* [3], *PScout* [4], and *Androguard* [6]) rely on static analysis to extract the mappings between APIs and permissions. The results of *PScout* is more complete and accurate than those of *Stowaway*. *PScout* have been applied on four versions of Android and help figure out that about 22% of the non-system permissions are unnecessary. *Androguard* is a reverse engineering tool which embeds PScouts methodology and enables API to recommend permissions to a given app. In our study, *Androguard* is used to extract the possible mappings between APIs and permisions. Note that these program analysis approaches are not perfect and many mappings that are recovered are not correct. Furthermore, the approach based on association rule mining has been proved to have better performance than these tools which only rely on program analysis in the study of Karim *et al.* [7]. Thus, we do not compare our proposed approach with these tools.

The misuse of permissions in Android has been investigated by researchers [5], [19]. For example, to understand whether Android users pay attention to, understand, and act on permission information during installation, Felt *et al.* [19] conduct



(a) F1-score@5 (b) F1-score@10 (c) MAP

Fig. 4. F1-score@5, F1-score@10 and MAP for N-Fold Cross Validation

an Internet survey with 308 Android users and a laboratory study with 25 users. They find that only 17% of users pay attention to permissions during installation and 3% of Internet survey respondents could correctly answer all three permission comprehension questions. These works provide the motivation of our research. The phenomenon of permission misuses exists in Android. Hence, permission recommendation approaches could be very helpful to both developers and end-users.

The permission recommendation approaches in this study could be used to detect malicious behavior, since the permission misuses could be indicative to stealthy and malicious behavior in Android apps. There are many approaches proposed by researchers to help identify malicious behavior in Android apps [20], [21], [22], [23], [24]. For example, *AsDroid* [20] could identifying stealthy behavior by analyzing user interface and program behavior contradiction.

V. Conclusion and Future Work

Android provides permission mechanism to help protect the privacy and security of Android app users. Unfortunately, there still exist many misuses of permissions [5], [19] which may be caused by the incompleteness of Android documentation. An approach based on association rule mining, proposed by Karim *et al*., has been reported to be useful to recommend appropriate permissions for an app [7]. This gives us a hint that other algorithms proposed in the recommend system area may also be applied to recommend permissions to apps. Hence, in this paper we propose another approach which is based on collaborative filtering technique to recommend permissions for a new app. We evaluate the approach on 936 Android Apps from F-Droid and compare our proposed approach with the association rule approach proposed by Karim *et al*. Results show that our proposed approach outperforms the baseline approach in terms of top-k precision, recall and F1-score and MAP.

Acknowledgment

This work is supported by NSFC Program (No.61602403 and 61402406) and National Key Technology R&D Program of the Ministry of Science and Technology of China under grant 2015BAH17F01.

References

[1] "Smartphone market share," http://www.idc.com/prodserv/smartphone-os-market-share.jspl.
[2] "Google play," https://en.wikipedia.org/wiki/Google_Play.
[3] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proc. CCS*. ACM, 2011, pp. 627–638.
[4] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proc. CCS*. ACM, 2012, pp. 217–228.
[5] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen, "Asking for (and about) permissions used by android apps," in *Proc. WCRE*. IEEE Press, 2013, pp. 31–40.
[6] A. Desnos, "Androguard: Reverse engineering, malware and goodware analysis of android applications... and more (ninja!)."
[7] M. Y. Karim, H. Kagdi, and M. Di Penta, "Mining android apps to recommend permissions," in *Proc. SANER*. IEEE, 2016, pp. 427–437.
[8] F. Ricci, L. Rokach, and B. Shapira, *Introduction to recommender systems handbook*. Springer, 2011.
[9] M. L. Collard, H. H. Kagdi, and J. I. Maletic, "An xml-based lightweight c++ fact extractor," in *Proc. of 11th IEEE International Workshop on Program Comprehension*. IEEE, 2003, pp. 134–143.
[10] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.
[11] F. Bodon, "A fast apriori implementation," in *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, vol. 90, 19. November 2003.
[12] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *Proc. MSR*. ACM, 2011, pp. 43–52.
[13] X. Xia, D. Lo, X. Wang, C. Zhang, and X. Wang, "Cross-language bug localization," in *Proc. ICPC*. ACM, 2014, pp. 275–278.
[14] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Proc. ICSE*. IEEE, 2012, pp. 14–24.
[15] C. Buckley and E. M. Voorhees, "Evaluating evaluation measure stability," in *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2000, pp. 33–40.
[16] J. A. Aslam, E. Yilmaz, and V. Pavlu, "The maximum entropy method for analyzing retrieval measures," in *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*. ACM, 2005, pp. 27–34.
[17] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.
[18] N. Cliff, *Ordinal methods for behavioral data analysis*. Psychology Press, 2014.
[19] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM, 2012, p. 3.
[20] 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*. ACM, 2014, pp. 1036–1046.
[21] A.-D. Schmidt, H.-G. Schmidt, L. Batyuk, J. H. Clausen, S. A. Camtepe, S. Albayrak, and C. Yildizli, "Smartphone malware evolution revisited: Android next target?" in *Proc. MALWARE*. IEEE, 2009, pp. 1–7.
[22] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets." in *NDSS*, 2012.
[23] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *Proc. ESEM*, 2016.
[24] L. Bao, D. Lo, X. Xia, X. Wang, and C. Tian, "How android app developers manage power consumption?: an empirical study by mining power management commits," in *Proc. MSR*. ACM, 2016, pp. 37–48.