You Are (not) Who Your Peers Are: Identification of Potentially Excessive Permission Requests in Android Apps

Prashanthi Mallojula, Javaria Ahmad, Fengjun Li, and Bo Luo
Department of Electrical Engineering and Computer Science
The University of Kansas, Lawrence, KS, USA
Email: prashanthi.mallojula@ku.edu, javaria.ahmad@ku.edu, fli@ku.edu, bluo@ku.edu

***Abstract*—Millions of Android applications are now deployed on billions of smartphones and tablet devices. An enormous amount of users' private data are being collected and made accessible to such apps. Extensive research efforts have been devoted to smartphone app security. In particular, the current practice of the app markets and app security scanners is to ensure that the *requested permissions* are consistent with the *used permissions*. On the other hand, mobile apps need to seek consent from users to approve various permissions to access user information. However, users often blindly accept permission requests and apps start to abuse this mechanism. For example, a flashlight app may obtain users' locations and send them out to the server. As long as a permission is *requested* by the app developer and *approved* by the users, the state-of-art detection mechanisms will treat it as benign.**

In this paper, we ask the question "are the permission requests really necessary?" The question is difficult to answer because it is hard to autonomously "comprehend" whether a permission is needed for the functionality of the app. We take the first attempt to tackle this challenge by comparing an app's permission requests with its peer apps, i.e., apps with similar functionalities. An app that requests/uses significantly more permissions than its peers is considered potentially malicious that will require further investigation. With this idea, we design a statistical approach to identify potentially excessive permission requests and evaluate it with apps from Play Store. Experiment results and case studies show that the proposed mechanism could effectively identify highly suspicious apps, which request many permissions that are not relevant to their functionalities.

I. Introduction

With the rapid growth of smartphones, tablets, watches, and other smart devices, mobile applications (mobile apps) have been widely adopted. For instance, 2.9 million apps were available on the Google Play Store as of December 2020 [1]. On average, 100,000+ new Android apps are released to the market on a monthly basis. The number of downloads for mobile apps is also steadily increasing [1]. With the advanced capacities in communication, computing, sensing, and user interactions, smartphones and apps have been adopted to assist and affect almost every aspect of our daily life.

The extreme popularity of smartphone apps has also led to increased security concerns, as these apps need to collect, access, and control a significant amount of user information to offer more sophisticated functionalities. In current practice, permission management in the Android ecosystem primarily relies on two mechanisms: (1) permission verification and (2) user consent. Considerable technical efforts have been devoted to the first mechanism, which aims to ensure that all the permissions invoked in the app are indeed declared/requested (in the app manifest), e.g., [2]. Towards this goal, various techniques have been proposed to scan apps' source code and/or monitor their run time status. For example, when an inconsistency is identified in the verification, i.e., an app uses a permission that it fails to request, the app will be denied by the Play Store. However, in this case, the most straightforward fix is to simply add the permission request into the manifest. Meanwhile, when the permission requests are presented to the end-users during installation, users often grant permissions just in order to access the functions. Previous studies show that users are unaware of what extent of data is being shared through the mobile app ecosystems, or they may be uncomfortable with it but do not have any effective mitigation [3] [4]. For example, when a specific flashlight app requests to access device and app history, retrieve running apps, read Home settings and shortcuts, access call information, etc, the app will be approved by Play Store as long as it has included all the permission requests in the manifest. Meanwhile, the end-users simply accept all the permissions so that they can install and use the app (the app has been installed 100,000+ times and received an average review score of 4.3).

In this paper, we would like to ask a question that has yet to attract due attention in the research community: "*Are the permission requests really necessary*?" That is, for each permission requested by an Android app, we ask whether the permission is really essential in fulfilling certain functionalities of the app. Answering this question is difficult since it is non-trivial to retrieve app functions from the descriptions and associate the functions with permissions. In this paper, we take an alternative approach. We assume that the function-permission relationship is relatively stable and consistent so that apps with similar functionalities are expected to require similar permissions. Moreover, if an app requires significantly more permissions than many of its peers, i.e., similar apps, then the app is highly suspicious that warrants further (manual) inspection. To this end, we have developed a statistical approach for excessive permission identification in Android apps. We first design a pair-wise evaluation mechanism for app permissions, and then aggregate all pair-wise assessments

of the seed app to generate an app permission risk score. The new approach does not require any code scanning or static/run-time analysis, rather, it is purely based on app similarities and permission requests. We demonstrate the performance of the proposed solution through experiments with 10,000+ apps. We also provide insightful analysis with detailed case studies.

The rest of the paper is organized as follows: We briefly review the literature in Section II, followed by a formal problem statement in Section III. We present the technical details of the proposed solution in Section IV, followed by experiment results and case studies in Sections V and VI. We discuss the advantages and weaknesses of the approach in Section VII, and finally, conclude the paper in Section VIII.

II. Related Work and Background

There is a wide spectrum of research efforts on mobile app privacy and security, including vulnerability scanning, application-level code analysis, run-time data flow analysis, and employing machine learning techniques on finding malicious apps. Comprehensive surveys of smartphone/Android security could be found at [5]–[7].

Static flow and code analysis. Code analysis was among the initial approaches to detect permission misuses and find data leaks in mobile apps' information flow. Several tools have been designed to utilize static taint analysis on Android app, such as Android leak [8], DroidBench, and FlowDriod [9]. Taint analysis is also experimented with IoT applications, e.g., SAINT is designed to analyze information flow from sensitive sources to external sinks [10]. There are works to detect third-party libraries and validate if they are safe to be used in the apps, e.g., LIBPECKER [11] is a tool that detects reliable libraries in android apps. Semantics-driven code analysis has been applied [12] to find clues to private data usage in mobile apps through semantic dependencies in the app code. Many mobile apps use the cloud as the back end for data storage. There are also research approaches that focus on finding privacy leaks related to cloud-based mobile apps, [13], [14]. In particular, Leakscope was proposed to detect data leaks due to various causes, such as mistakes in authentication or key management, or mis-configurations [13]. Last, there are research efforts that analyze app developer commits and updates to check who is adding/modifying permissions to the apps. [15]

Dynamic analysis. Dynamic analysis approaches aim to find data leaks through run-time information flow analysis. Along this thrust, several tools have been proposed to detect privacy leaks in mobile apps. For example, DrioidTrace is a dynamic analysis system that could identify malicious payloads behaviors [16]. [17] combines static and dynamic analysis to identify permissions employed in an app, and compares them with permissions requested in the manifest. LeakDoctor also integrates dynamic analysis and static taint analysis to examine privacy leaks, and determines if a data disclosure is necessary for an app function [18].

Permission analysis and permission-description mapping. State of art machine learning approaches have been used to "understand" permission usage in mobile apps and identify malicious apps [19]. This thrust of research also attempts to associate app permissions with its descriptions or functionalities. Natural Language Processing (NLP) techniques have been employed to analyze privacy policies, identify privacy practices declared in the policies, and detect any inconsistencies with code analysis, e.g., [20]. [21] maps app permissions with keywords from the app descriptions, to validate if the app is consistent in disclosing user information access. [22] employs NLP methods to parse app descriptions, maps them with app permissions, and identifies any inconsistencies. Last, there is work to provide the user with a personal assistant to manage the permissions of mobile apps on their devices [23].

In this paper, we present a novel approach that identifies permission misuse from a unique angle. We identify potentially malicious apps based on permission statistics and functional similarities among mobile apps.

III. Motivation and Problem Statement

In the current research and practice of mobile app security, a significant amount of effort is devoted on ensuring the consistency between the requested permissions and the actually used permissions [2], [24], [25]. That is, when an app employs sensitive resources or information, they must be requested in the manifest and approved by the user. However, this approach falls short in eliminating the potentially malicious permission usage. A malicious app may employ a large amount of unnecessary/excessive permissions while still passing through all the validations as long as it properly "requests" such permissions in the manifest. During the app stores' review process, if an app is found to use unclaimed permissions, it will be rejected. However, a simple fix is to add the permission request to the manifest and the app will be approved, while nobody is interested in any further investigation on how/why such permission is used or whether the permission is truly needed. Meanwhile, the end-users often blindly accept all the permission requests without questioning their legitimacy. As a result, one can easily identify a significant amount of seemingly legitimate apps, which employ permissions that cannot be tied to any of their functionalities, e.g., a flashlight app that uses Bluetooth, device identities, or USB storage.

To tackle this challenge, we argue that app permission usage needs to be validated against their functions, while permission requests that are not endorsed by valid functions warrant further investigation. It is practically very challenging to automatically parse app descriptions, accurately extract a list of functions, and match the functions with permission requests for each app. In this project, we take a different route which evaluates each app against its peers, i.e., apps with similar functionalities. Intuitively, if an app uses significantly more permissions or accesses significantly more private information than the majority of its peers, the app appears to be highly suspicious. To detect excessive permission requests based on this observation, we need to develop quantitative measurements

TABLE I
TERMINOLOGIES USED IN THIS PAPER.

Term	Notation	Meaning
Seed app	S	The target application to be evaluated
Seed permission	$\mathbf{P_S}$=$\{p_s\}$	A set of permissions requested by S
Peer app	P	An app with similar functions as the seed app
Peer app group	$\mathbf{A_S}$=$\{P_i\}$	The set of all peer apps of seed app S
Peer permission	$\mathbf{P_{P_i}}$	A set of permissions requested by peer app P_i
Common permissions	$\mathbf{P_C}$=$\mathbf{P_S} \cap \mathbf{P_{P_i}}$	A set of permissions requested by both the seed app S and the peer app P_i
Seed-exclusive permissions	$\mathbf{P_{S^*}}$=$\mathbf{P_S}$−$\mathbf{P_C}$	A set of permissions that are only used by the seed app S but not the peer app P_i
Peer-exclusive permissions	$\mathbf{P_{P_i^*}}$=$\mathbf{P_{P_i}}$−$\mathbf{P_C}$	A set of permissions that are only used by the peer app P_i but not the seed app S
Seed Exclusiveness Ratio (SER)	$\lvert\mathbf{P_{S^*}}\rvert/\lvert\mathbf{P_S}\rvert$	The proportion of seed-exclusive permissions out of all seed permissions
Seed to Peer Exclusiveness Ratio (SPER)	$\frac{\lvert\mathbf{P_{S^*}}\rvert}{\lvert\mathbf{P_S^*}\rvert+\mathbf{P_{P_i^*}}\rvert}$	The proportion of seed-exclusive permissions out of all seed-exclusive and peer-exclusive permissions

for three components: (1) app similarity assessment based on app functions; (2) pair-wise evaluation of app permission requests; and (3) aggregation of all pair-wise evaluations. In this paper, we primarily focus on (2) and (3), while we rely on Play Store's app similarity measurement for (1). Formally, the objective of this paper is to develop a quantitative evaluation mechanism to detect potentially excessive app permission requests based on a statistical analysis of the permission requests.

In this paper, we denote the target app to be evaluated as the *seed app*. We denote the apps with similar functions as the seed app as the *peer apps*. The terminologies used in this paper are summarized in Table I.

IV. EXCESSIVE PERMISSION IDENTIFICATION

In this section, we present the technical details of the proposed app permission analysis and excessive permission detection mechanism.

A. Data Collection and Statistics

We have developed a simple crawler to collect app information from Google Play Store. We crawl at a very low rate so that we do not impact the operation of the Play Store. In a 6-month time span (from September 2020 to February 2021), we have collected information of 11,338 mobile apps, which will be used in the experiments. The crawled apps are from 48 different categories. Table II demonstrates the top categories. This data set includes 5,107 paid apps and 6231 free apps.

Information for the seed apps is crawled from each app's Google Play Store page. Peer apps are collected through Play Stores' "similar apps" link. All the similar apps recommended by the Play Store are crawled and considered as the peer group ($\mathbf{A}_S$) of the seed app. The size of $\mathbf{A}_S$, i.e., the number of peer apps, varies for each seed. On average there are approximately 30 peer apps per group. The largest peer group, which contains 196 peer apps, belongs to the seed application "Smash Hit".

TABLE II
SEED APP COUNT IN THE TOP CATEGORIES.

Category	Count	Category	Count
Education	1585	Tools	750
Books & Reference	702	Business	573
Finance	573	Personalization	516
Productivity	486	Puzzle	441
Travel & Local	416	Entertainment	393

The highest number of permissions requested by a seed is 143 for app "MultiSpace for FreeFire". Whereas the lowest number of permissions is 0, which means the seed app does not request any special permission at all. In this project, all the seeds without any permission requests are considered *not potentially malicious*. There are 10 applications with no permissions requested.

B. Pair-wise Permission Evaluation

In this project, we adopt a two-phase process for app permission analysis, as introduced in Section III. In the first phase, we evaluate the permission requests of the seed app against *each* of its peer apps. Intuitively, if the seed employs a significant portion of permissions that are not requested by its peer, the seed could be considered somewhat suspicious in comparison with this peer. In practice, we denote the set of permissions requested/used by the seed app as the *seed permissions* ($\mathbf{P_S}$), the set of permissions requested by the peer app P_i as *peer permissions* ($\mathbf{P_{P_i}}$). The permissions shared by both the seed and peer apps correspond to the intersection of these two sets, i.e., the *common permissions*: $\mathbf{P_C} = \mathbf{P_S} \cap \mathbf{P_{P_i}}$. Moreover, the permissions that are only used by the seed app are denoted as the seed-exclusive permissions $\mathbf{P_{S^*}}$=$\mathbf{P_S}$−$\mathbf{P_C}$, and correspondingly we also have the peer-exclusive permissions $\mathbf{P_{P_i^*}}$=$\mathbf{P_{P_i}}$−$\mathbf{P_C}$. In comparing $\mathbf{P_S}$ and $\mathbf{P_{P_i}}$, the relationship between S and P_i could be roughly categorized into four cases: (1) in the ideal scenario, two functionally similar apps shall employ a similar set of permissions, i.e., $|\mathbf{P_C}| >> |\mathbf{P_{S^*}}| \approx |\mathbf{P_{P_i^*}}|$, where $|\mathbf{P}|$ denotes the size of set $\mathbf{P}$. (2) When $\mathbf{P_S}$ and $\mathbf{P_{P_i}}$ appear to be disjoint ($\mathbf{P_S} \cap \mathbf{P_{P_i}} \approx \varnothing$), S and P_i are likely to be two less relevant apps. (3) When S employs a significant amount of permissions that are not used by P_i, i.e., $|\mathbf{P_{S^*}}| >> |\mathbf{P_C}|$ and $|\mathbf{P_{P_i^*}}|$ is small, S appears to be suspicious in comparing with P_i. And finally (4) When P_i uses significantly more permissions than S, P_i appears to be suspicious.

Since this analysis targets the seed app, we are concerned when the seed is potentially malicious out of these four cases. That is, we design a quantitative measure that distinguishes case (3) from above. In particular, we define the Seed Ex-

TABLE III
RISK CATEGORIZATION FOR PAIR-WISE SERS.

Risk Category	Definition	Range	Risk Level
R_0	$[0, \bar{s}-\sigma]$	[0, 0.102]	Very Low
R_1	$[\bar{s}-\sigma, \bar{s}]$	[0.102, 0.354]	Low
R_2	$[\bar{s}, \bar{s}+\sigma]$	[0.354, 0.605]	Medium
R_3	$[\bar{s}+\sigma, \bar{s}+2\sigma]$	[0.605, 0.857]	High
R_4	$[\bar{s}+2\sigma, 1]$	[0.857, 1]	Very High

clusiveness Ratio (SER) as the proportion of seed-exclusive permissions out of all seed permissions in comparing with P_i:

$$\text{SER}(S, P_i) = \frac{|\mathbf{P_{S^*}}|}{|\mathbf{P_S}|} \tag{1}$$

$\text{SER} \in [0, 1]$ measures how the seed permissions are (not) used by a peer app with similar functions. A higher SER indicates a higher likelihood that S has requested unnecessary/excessive permissions in the context of P_i.

Next, we evaluate all the pair-wise SERs for each seed app against each peer app and show the distribution of the SERs in Figure 1. As shown in the figure, the pair-wise SERs in our dataset roughly follow a power-law distribution, with a mean ($\bar{s}$) of 0.354, and a standard deviation (σ) of 0.251. The results show that, on average, 35.4% of the seed permissions are not used in a peer app. This could be explained by the fact that the seed and peer apps still demonstrate differences in their functionalities, while the "similar app" designation in Play Store does not indicate perfect matching.

Finally, we categorize the SER values into five risk categories (RCs) based on four threshold values: $\bar{s}-\sigma$, $\bar{s}$, $\bar{s}+\sigma$, $\bar{s}+2\sigma$. The risk categories, $R_0, ..., R4$, and their implied risk levels are shown in Table III. In this way, each peer app "designates" a risk category label to the seed app. Please note that the categories are designed to be asymmetric. That is, we use three categories for $SER{>}s$, and two categories for $SER{<}s$, since we pay more attention to the high-risk cases.

C. Aggregate Permission Analysis and App Risk Scoring

For a seed app, each individual $\text{SER}(S, P_i)$ value or the corresponding risk category R_i only provides a partial view of the relative risk of the seed in the context of peer P_i. When we evaluate a seed app with all the apps in its peer group $\mathbf{A_S}$, we have the following observations: (1) when $\text{SER}(S, P_i)$ is *significantly* larger than $\bar{s}$, S demonstrates higher risk in comparing with P_i; and (2) when S demonstrates higher risk than a *significant portion* of apps in $\mathbf{A_S}$, S appears to be highly suspicious that warrants further manual examination.

In practice, we design a "voting" mechanism for aggregate permission analysis. For each app, we evaluate the pair-wise SER against each peer app in $\mathbf{A_S}$, and then aggregate the risk categorizations for each SER to generate the integrated permission risk score r_s for the seed app.

$$r_s = \frac{\sum_{i=0}^{4} i \times |\mathbf{R_i}|}{|\mathbf{A_s}|} \tag{2}$$

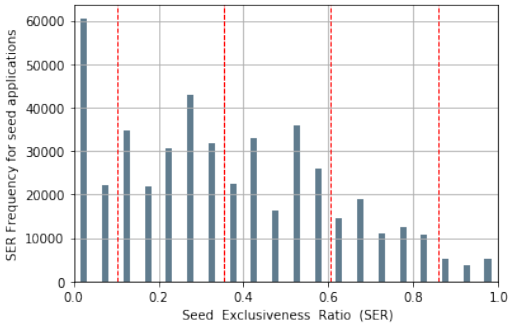

Fig. 1. Distribution of Seed Exclusive Ratio (SER)

where $|\mathbf{R_i}|$ denotes the number of peer apps that indicate risk category R_i for the seed, and $|\mathbf{A_s}|$ denotes the total number of peer apps.

Finally, $r_s \in [0, 4]$ indicates the peer-voted permission risk score of seed S. Using this approach, potentially malicious apps are identified based on being more aggressive on permission requests compared with its peer group. For better comprehension of the risk scores, we can further place the seed app into risk categories based on r_s, i.e., the app demonstrates very low risk when $r_s \in [0, 1]$, while the app demonstrates high risk when $r_s \in [3, 4]$.

D. Evaluation of Seed-to-Peer Differences

The app risk score r_s rely on the accurate identification of "similar apps". In case S and P_i are significantly different, $\mathbf{P_S}$ and $\mathbf{P_{P_i}}$ will be disjoint while $\text{SER}(S, P_i)$ will be large, which pushes the seed app towards "high risk". To partially mitigate this issue, we introduce another metric, the *Seed-to-Peer Exclusiveness Ratio* (SPER), which is the proportion of seed-exclusive permissions over seed- and peer-exclusive permissions:

$$\text{SPER}(S, P_{P_i}) = \frac{|\mathbf{P_{S^*}}|}{|\mathbf{P^*_S}| + \mathbf{P_{P^*_i}}|} \tag{3}$$

$\text{SPER} \in [0, 1]$ measures how the seed app permissions different from peer app permissions. A higher SPER indicates that the seed app uses significantly more exclusive permissions than the peer app, i.e., the seed app may be risky. We further employ the same approach in Section IV-C to obtain the distribution of SPER, and then apply the same voting mechanism on SPER to calculate the second risk score, r_{sp}, which serves as a complement of r_s. With r_{sp}, we have the following assessments: (1) The range of r_{sp} for all the seed apps is in $[0, 3]$. (2) r_{sp} is low if most of the peer permissions are different from the seed. (3) r_{sp} is high if the seed uses (significantly) more exclusive permissions than most of its peers, i.e., the seed appears to be risky.

r_{sp} serves as a complement of r_s. An app with high r_s and high r_{sp} is considered a candidate for permission abuse, which should be further examined. Meanwhile, an app with high r_s but low r_{sp} most likely indicate that the "similar apps" function performed poorly for the seed.

TABLE IV
CATEGORIZATION OF APP PERMISSION RISKS

r_s range	0	(0.0, 1.0]	(1.0, 2.0]	(2.0, 3.0]	(3.0, 4.0]
app count	597	2608	5902	2117	92

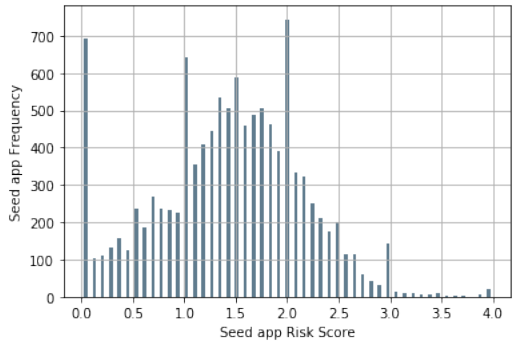

Fig. 2. Seed Application Distribution Based on Risk Score r_s.

V. EXPERIMENT RESULTS

The experiments are conducted with all 11,338 seed apps. We perform pair-wise permission evaluation for each seed app against each peer app to calculate the SERs (Eq. 1). We further aggregate the SERs for each seed to generate r_s (Eq. 2). There are several special cases: (1) There are 10 seed applications that do not use any special permission ($\mathbf{P_S} = \varnothing$), hence, they are considered not malicious ($r_s = 0$). (2) The permission requests of 12 apps are completely disjoint with all their peers ($\mathbf{P_S} \cap \mathbf{P_{P_i}} = \varnothing$). They are primarily caused by inaccurate similarity assessments by Play Store. We have generated r_s for all the remaining apps, and present the score distributions in Figure 2 and Table IV. As shown in the figure, the permission risk scores for all the apps roughly follow a normal distribution. r_s for the majority of the apps lie in the range of [1.0, 3.0], while there is a small portion of apps that demonstrate high risk. We also like to point out that 597 apps do not use any seed-exclusive permission compared with all the peers, i.e., $\mathbf{P_S} = \mathbf{P_C} \subseteq \mathbf{P_{P_i}}, \forall i$. That is, all the seed permissions are used by every peer app. In this case, all the SERs are 0 and the eventual risk scores are also 0. To further evaluate the validity of the app permission scoring mechanism, we employed Amazon Mechanical Turk (MTurk) for a user study, and also manually examined the high-risk apps.

User study using MTurk. To further validate the effectiveness of the proposed app permission risk score, we perform a small-scale user study. We randomly pick 12 apps and posted them to Amazon Mechanical Turk (MTurk). For each app, we displayed its app description and requested permissions, and asked the Turker to rate the permission requests as "excessive", "not sure", or "reasonable". Each MTurk task consists of 4 apps, with a compensation of $0.40. Each app is rated by 50 Turkers. We calculate a weighted sum of all the user inputs for each app, and denote it as the MTurk score. The scatter plot for r_s and the MTurk score is presented in Figure 3. The Pearson correlation coefficient is 0.664, indicating a strong correlation. The experiment result shows that the proposed r_s scores are highly consistent with users' perceptions.

TABLE V
MOST USED EXCESSIVE PERMISSIONS.

Permission	Seed count
Read the contents of your USB storage	39
Modify or delete the contents of your USB storage	37
Prevent device from sleeping	34
Receive data from Internet	29
View Wi-Fi connections	28
Control vibration	27
Full network access	26
Read phone status and identity	24
Run at startup	24
Google Play license check	24
Precise location (GPS and network-based)	23
Record audio	22
Take pictures and videos	22
View network connections	22
Approximate location (network-based)	20

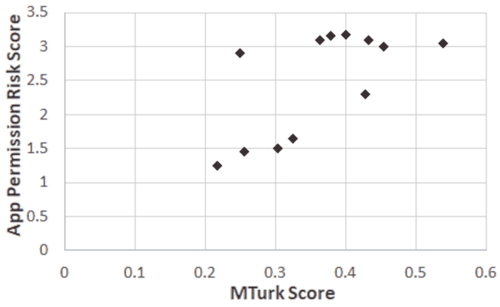

Fig. 3. Correlation between the app risk score r_s and MTurk user annotated score.

Manual examination of seed-exclusive permissions. As we have shown, 92 applications are categorized as potentially high-risk applications with $r_s \in (3.0, 4.4]$. We further manually examine all the apps in this category to validate the correctness of the risk assessment and to further investigate what/how excessive permissions are used. We read the function descriptions of these high-risk apps, and found that at least 30% of their total requested permissions cannot be explicitly or implicitly associated with any published functions. Moreover, the most popular seed-exclusive permissions are shown in Table V. For instance, permission "read the contents of your USB storage" appeared very frequently in the $\mathbf{P_{S^*}}$ of 39 seed apps, where it is rarely used by their peer apps.

VI. CASE STUDIES

In this section, we present four case studies, in which we manually examine randomly selected apps, including three apps from the "high risk" category and one from the "low risk" category. We present the app permission risk score of each app, qualitatively compare each app with several peer apps, and discuss the validity of the permission risk scoring.

TABLE VI
COMPARING EASY FLASHLIGHT WITH PEER APPLICATIONS.

EF: Easy Flashlight (com.jjapp.easyflash);
FP: Flashlight Pro: No Permissions (com.humberto.flashlightpro);
IT: Icon Torch - Flashlight (ru.irk.ang.balsan.shortcutled);
F: Flashlight (com.binghuo.torchlight.flashlight);
FL: Flash light-Bright LED (coocent.app.tools.light.flashlight);
.

Permission	EF	FP	IT	F	FL
Control flashlight	✓	✓	✓	✓	✓
Device and app history	✓				
Device ID and call information	✓				
Disable your screen lock	✓				
Draw over other apps	✓				
Expand/collapse status bar	✓				
Full network access	✓			✓	✓
Install shortcut	✓				
Precise location (GPS and network-based)	✓				
Prevent device from sleeping	✓		✓	✓	✓
Read the contents of your USB storage	✓				
Modify or delete the contents of your USB storage	✓				
Read phone status and identity	✓				
Read Home settings and shortcuts	✓				
Read sensitive log data	✓				
Retrieve running apps	✓				
Run at startup	✓			✓	
Take pictures and videos	✓	✓	✓	✓	✓
View network connections	✓		✓	✓	✓
View Wi-Fi connections	✓				
Uninstall shortcuts	✓				

TABLE VII
COMPARING BATTERY CHARGE SOUND ALERT WITH PEERS.

BC: Battery Charge Sound Alert - Yellow (ddolcatmaster.batterychargealertmanagement);
BA: Battery Alarm (simple.batttery.alarm);
FB: Full Battery Charge Alarm (pps.syrupy.fullbatterychargealarm);
B: Battery (com.macropinch.pearl);
BAP: Battery Alarm PRO(simple.battery.alarm);

Permission	BC	BA	FB	B	BAP
Access Bluetooth setting	✓				
Change network connectivity	✓				
Connect and disconnect from Wi-Fi	✓				
Control vibration	✓	✓	✓		✓
Full network access	✓		✓	✓	
Modify and delete the contents of your USB storage	✓				
Modify system settings	✓	✓			✓
Pair with Bluetooth devices	✓				
Prevent device from sleeping	✓	✓	✓	✓	✓
Read the contents of your USB storage	✓				
Read Home settings and shortcuts	✓				
Read sync settings	✓				
Receive data from Internet	✓			✓	
Run at startup	✓	✓	✓	✓	✓
Toggle sync on and off	✓				
View network connections	✓		✓	✓	
View Wi-Fi connections	✓				

A. Easy Flashlight

There are currently hundreds of flashlight apps in the Google Play Store. These apps offer simple interfaces and limited functions–controlling the flashlight LEDs on smart phones. In our experiments, the Easy Flashlight - Super Bright LED Flashlight app (com.jjapp.easyflash) has received an r_s score of 3.1 and an r_{sp} score of 2.9, which indicates highly malicious in terms of permission usage.

Table VI shows the comparison between the seed (Easy Flashlight) and some of its peer apps to highlight excessive permissions accessed. As per the functional requirements of the flashlight apps, it is reasonable to access flashlight-related settings. It still appears to be reasonable to access network-related settings, as is requested in most other flashlight apps. Further examination shows that network access is used for in-app advertisements. Whereas, Easy Flashlight has requested permissions that are dramatically more than its peer applications, while most of such permission requests cannot be explained by any of the published functionalities of the app, such as "accessing GPS" and "read sensitive log data", etc. In particular, this app requests 21 permissions in total, most of which are not used in any of its peer app. Hence, we confirm Easy Flashlight as potentially malicious based on the excessive request for permissions that are unexplained. Last, we also noticed that a negative Play Store review that is rated as "most helpful" also pointed out the excessive permission requests: "*Spy app? I could not understand, why does it need almost all permission? Is it a spy app?*".

B. Battery Charge Sound Alert

We examine the Battery Charge Sound Alert - Yellow (ddolcatmaster.batterychargealertmanagement) on Google Play Store for functionality details. Based on the description, this app enables the user to set a reminder song, which is played once the battery is fully charged. This app receives an r_s score of 3, and an r_{sp} score of 3, which indicates potentially malicious. A comparison of this app and some peers are shown in Table VII. In a manual evaluation, we found that most of the seed-exclusive permission requests cannot be explained by the app functions, e.g., permissions related to USB storage modification, Bluetooth settings, etc.

C. Fast Electric Pro

The Fast Electric Pro: electrical calculator (No Ads) (com.androny.egy.fast_electricPro) is a paid app that provides basic functions of a scientific calculator. It receives an r_s score of 3.38, which indicates highly risky. Meanwhile, its r_{sp} score is 2.1, which indicates that some of its peers may be less

TABLE VIII
COMPARING FAST ELECTRIC PRO WITH PEER APPS.

FEP: Fast Electric Pro (com.androny.egy.fast_electricPro);
EP: ElectriCalc Pro Calculator (com.calculated.carmencita);
ECP: Electrician Calculator Pro (com.rfxlabs.electriciancalculator);
EC: Electrical Calc Elite Electric (cyberprodigy.electrical.calc.elite);
IE: InstElectric Pro - Electricity(com.xlingenieria.instelectric.pro);

Permissions	FEP	EP	ECP	EC	IE
Approximate location (network-based)	✓				
Control vibration	✓	✓			✓
Full network access	✓	✓	✓		✓
Google Play license check	✓	✓		✓	
Pair with Bluetooth devices	✓				
Precise location (GPS and network-based)	✓				
Prevent device from sleeping	✓				✓
Receive data from Internet	✓				✓
Run at startup	✓				✓
View network connections	✓				✓
View Wi-Fi connections	✓				

relevant to the seed. A comparison between Fast Electric Pro and its peer apps is shown in Table VIII. We further manually examined the functions and permission requests of this app and found that some permissions that are exclusively requested by the seed cannot be explained with a published function, e.g., precise location (GPS) and Bluetooth.

D. WhatsApp Messenger

Most of the instant messaging and social networking apps need relatively more permissions to access user data and hardware, such as GPS, camera, storage. Latest messenger apps also added features such as video calling, broadcasting messages, video sharing, etc. Among the messaging apps, WhatsApp Messenger (com.whatsapp) received an app permission risk score of 2.2, which implies average risk.

Comparing with its peer applications, WhatsApp Messenger accesses a similar amount of user information, including contacts and SMS. However, the messaging and social networking apps are highly diversified, where each app implements a different set of features/functions and hence employs a different set of permissions. This is very different from the flashlight or calculator apps, where most of the peer apps share very similar functions. The WhatsApp Messenger app uses a total of 53 permissions, while its peer apps mostly use similar numbers of permissions. However, each peer app uses a different set of permissions that partially overlaps with WhatsApp Messenger. As a result, WhatsApp Messenger is rated as moderately risky in comparison with its peers, which we think is reasonable.

VII. DISCUSSIONS

In this section, we compare the proposed app permission risk assessment mechanism with code analysis approaches and further discuss the limitations of our current design.

Compare with code analysis approaches. Code analysis is the most popular method in mobile/Android security research. Existing solutions have shown to be very effective in detecting malicious behaviors, such as permission abuse and private information leakage [5]. In this paper, we take a different perspective, in which we focus on the legitimacy and necessity of the permission requests. That is, the fact that a permission is explicitly requested by the developer and consented by the user does not make the request appropriate. As we have shown in Sections V and VI, many permission requests cannot be explained by the legitimate functionalities of the app.

We further compare our experimental results with code analysis tools. MobSF is an automated code analysis (static and dynamic analysis) and security assessment tool for Android apps [26]. To demonstrate the differences between code analysis and our approach, we employ MobSF to examine the four apps we used in case studies (Section VI). MobSF reports Whatsapp Messenger as high risk, Easy Flashlight, Battery Charge Sound Alert as medium risk, and Fast Electric Pro as not risky. In particular, MobSF reports the permission of displaying system-level alerts as dangerous because it may be exploited by malicious users. It also considers camera access as risky. However, from our perspective, it is reasonable for the Messenger app to access the camera for video calls or to invoke system-level alerts for incoming calls. These permissions are also employed by its peer apps, i.e., other instant messaging and video conferencing apps. On the other hand, while the other three apps do not request or use high-risk permissions such as cameras, most of their requested permissions are not used by their peer apps, and they cannot be justified by their published functions.

False positives. The proposed app permission risk scoring mechanism may rate legitimate permission requests as unnecessary/excessive, which implies false positives. False positives are mostly caused by three circumstances: (1) the seed app supports significantly more functions than many of its peer apps and hence requires more permissions, (2) the "similar app" function by Play Store mistakenly place less relevant apps as similar, (3) the same functions may be implemented in slightly different ways and thus requires different permissions. In practice, case (3) is rare since SER (Eq. 1) could tolerate a small number of inconsistent permissions, while the risk scoring mechanism (Eq 2) could also tolerate a few outlier peers. Meanwhile, cases (1) and (2) appear to be the root causes of most of the false positives generated in our approach. For example, when a health tracking app (seed app) that tracks all user health information is considered similar to a large number of pedometer apps (peer apps), all these pedometer apps will generate relatively high SERs for the seed app since the pedometer apps are significantly simpler so that they require fewer permissions.

To tackle this issue, we have designed the SPER and r_{sp} measurements to partially mitigate the problem of irrelevant apps in the peer group, i.e., to handle case (2). Meanwhile, we have planned to further examine app functions instead of relying on Play Store's assessment. We will design fine-grained app similarity measurements based on the public information

that we can collect, e.g., app descriptions, reviews, privacy policies, etc. As we add the app similarity measurements to Play Store's similarity assessment, we will also introduce a weighting mechanism in app risk scoring (Eq. 2). That is, SERs generated from highly similar peers will carry higher weights than less similar peers.

Risk level of permissions. As shown in [26], different permissions may imply different levels of risk, e.g., if an app employs an unnecessary permission of camera access, it implies a higher risk than an unnecessary permission of disable screen lock. While we currently treat all the unnecessary permission requests as equally risky, it is our future plan to add permission risk levels to our app risk scoring.

VIII. Conclusion and Future Work

In this paper, we propose a unique approach to identify potentially excessive permission usage by Android apps based on comparison with their peer applications. This approach runs on the assumption that permissions are used to support app functions so that apps with similar functionalities are expected to request/use similar sets of permissions. A statistical analysis approach is designed to examine the shared and exclusive permissions between pairs of apps. All seed applications that do not stand with their peer applications are considered to be potentially malicious. With experiments using more than 11,000 apps, we assess the app permission risk score r_s and demonstrate the effectiveness of the risk scores. The current approach can be improved with a more accurate measurement of application similarities. We also plan to further enhance the expressiveness of the proposed mechanism by adding permission risk level and improve the scoring mechanism.

Acknowledgements

The authors were sponsored in part by NSF DGE-1565570, IIS-2014552, and the Ripple University Blockchain Research Initiative. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] J. Clement, "Mobile app usage – statistics & facts," https://www.statista.com/topics/1002/mobile-app-usage/, online; accessed April 2021.

[2] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri, "A study of android application security." in *USENIX security symposium*, vol. 2, no. 2, 2011.

[3] 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*, 2012, pp. 1–14.

[4] J. Lin, S. Amini, J. I. Hong, N. Sadeh, J. Lindqvist, and J. Zhang, "Expectation and purpose: understanding users' mental models of mobile app privacy through crowdsourcing," in *Proceedings of the 2012 ACM conference on ubiquitous computing*, 2012, pp. 501–510.

[5] P. Bhat and K. Dutta, "A survey on various threats and current state of security in android platform," *ACM Computing Surveys (CSUR)*, vol. 52, no. 1, pp. 1–35, 2019.

[6] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev, "Google android: A state-of-the-art review of security mechanisms," *arXiv preprint arXiv:0912.5101*, 2009.

[7] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE communications surveys & tutorials*, vol. 17, no. 2, pp. 998–1022, 2014.

[8] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *International Conference on Trust and Trustworthy Computing*, 2012, pp. 291–307.

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[10] A. Acar, H. Fereidooni, T. Abera, A. K. Sikder, M. Miettinen, H. Aksu, M. Conti, A.-R. Sadeghi, and A. S. Uluagac, "Peek-a-boo: I see your smart home activities, even encrypted!" *arXiv preprint arXiv:1808.02741*, 2018.

[11] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen, "Detecting third-party libraries in android applications with high precision and recall," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 141–152.

[12] Y. Nan, Z. Yang, X. Wang, Y. Zhang, D. Zhu, and M. Yang, "Finding clues for your secrets: Semantics-driven, learning-based privacy discovery in mobile apps." in *NDSS*, 2018.

[13] C. Zuo, Z. Lin, and Y. Zhang, "Why does your data leak? uncovering the data leakage in cloud from mobile apps," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1296–1310.

[14] O. Alrawi, C. Zuo, R. Duan, R. P. Kasturi, Z. Lin, and B. Saltaformaggio, "The betrayal at cloud city: an empirical analysis of cloud-based mobile backends," in *USENIX Security Symposium*, 2019.

[15] D. E. Krutz, N. Munaiah, A. Peruma, and M. W. Mkaouer, "Who added that permission to my app? an analysis of developer permission changes in open source android apps," in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, 2017, pp. 165–169.

[16] M. Zheng, M. Sun, and J. C. Lui, "Droidtrace: A ptrace based android dynamic analysis system with forward execution capability," in *2014 international wireless communications and mobile computing conference (IWCMC)*, 2014, pp. 128–133.

[17] D. Geneiatakis, I. N. Fovino, I. Kounelis, and P. Stirparo, "A permission verification approach for android mobile applications," *Computers & Security*, vol. 49, pp. 192–205, 2015.

[18] X. Wang, A. Continella, Y. Yang, Y. He, and S. Zhu, "Leakdoctor: Toward automatically diagnosing privacy leaks in mobile applications," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 3, no. 1, pp. 1–25, 2019.

[19] L. Wei, W. Luo, J. Weng, Y. Zhong, X. Zhang, and Z. Yan, "Machine learning-based malicious application detection of android," *IEEE Access*, vol. 5, pp. 25 591–25 601, 2017.

[20] P. Story, S. Zimmeck, A. Ravichander, D. Smullen, Z. Wang, J. Reidenberg, N. C. Russell, and N. Sadeh, "Natural language processing for mobile app privacy compliance," in *AAAI Spring Symposium on Privacy-Enhancing Artificial Intelligence and Language Technologies*, 2019.

[21] M. Wei, X. Gong, and W. Wang, "Claim what you need: a text-mining approach on android permission request authorization," in *2015 IEEE Global Communications Conference (GLOBECOM)*, 2015, pp. 1–6.

[22] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "Autocog: Measuring the description-to-permission fidelity in android applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 1354–1365.

[23] B. Liu, M. S. Andersen, F. Schaub, H. Almuhimedi, S. A. Zhang, N. Sadeh, Y. Agarwal, and A. Acquisti, "Follow my recommendations: A personalized privacy assistant for mobile app permissions," in *Symposium on Usable Privacy and Security ({SOUPS})*, 2016.

[24] M. Backes, S. Bugiel, E. Derr, P. McDaniel, D. Octeau, and S. Weisgerber, "On demystifying the android application framework: Re-visiting android permission specification analysis," in *25th {USENIX} security symposium ({USENIX} security 16)*, 2016, pp. 1101–1118.

[25] 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*, 2012, pp. 31–40.

[26] A. Abraham *et al.*, "Mobile Security Framework (MobSF)," https://github.com/ajinabraham/Mobile-Security-Framework-MobSF, online; accessed 29 January 2014.