Permission Issues in Open-source Android Apps: An Exploratory Study

Gian Luca Scoccia*, Anthony Peruma†, Virginia Pujols†, Ivano Malavolta‡, Daniel E. Krutz†
*Gran Sasso Science Institute, L'Aquila, Italy
gianluca.scoccia@gssi.it
†Rochester Institute of Technology, Rochester, NY, USA
{axp6201, vp2532, dxkvse}@rit.edu
‡Vrije University, Amsterdam, The Netherlands
i.malavolta@vu.nl

***Abstract*—Permissions are one of the most fundamental components for protecting an Android user's privacy and security. Unfortunately, developers frequently misuse permissions by requiring too many or too few permissions, or by not adhering to permission best practices. These permission-related issues can negatively impact users in a variety of ways, ranging from creating a poor user experience to severe privacy and security implications. To advance the understanding permission-related issues during the app's development process, we conducted an empirical study of 574 GitHub repositories of open-source Android apps. We analyzed the occurrences of four types of permission-related issues across the lifetime of the apps.**

Our findings reveal that (i) permission-related issues are a frequent phenomenon in Android apps, (ii) the majority of issues are fixed within a few days after their introduction, (iii) permission-related issues can frequently linger inside an app for an extended period of time, which can be as high as several years, before being fixed, and (iv) both project newcomers and regular contributors exhibit the same behaviour in terms of number of introduced and fixed permission-related issues per commit.

***Index Terms*—Mobile Permissions, Android, Mobile Software Engineering, Software Repository Mining**

I. Introduction

The *apps* on our mobile devices enable us to do everything from trade stocks to record vital health information. Although these apps provide immense amounts of power, they also present an unparalleled opportunity for security and privacy threats. Due to the magnitude of these threats, it is imperative that developers create apps that sufficiently protect our privacy and security [22].

The sensitive data and functionality used by an app is protected through permissions. Android apps use a permission-based system where an app requires specific permissions to carry out specific operations [8]. A developer must explicitly state the permissions an app may request, and the end-user can accept a subset of requested permissions that are deemed *dangerous* [3]. Example dangerous permissions include the ability to read SMS messages, record audio, and access the user's location. It is crucial for developers to make proper permission-related decisions since improperly used permissions (under and over-permissions) carry a wide range of ramifications. These include increased app susceptibility to malware and unwanted data leakage to ad libraries [23], [29], [26]. Additionally, not adhering to permissions best practices may have a wide range of implications. These may range from hurting the user experience, to creating functional defects and privacy and security-related issues [5], [20], [48], [52].

Unfortunately, developers do not always correctly use permissions for numerous reasons, including a lack of permissions-related knowledge [54] and even confusion over the permission's name [23]. There is substantial work examining the detrimental effects of permissions misuse [24], [25], [61] and tools to assist in the identification of a variety of *permission-related issues* (PRIs) [23], [11]. However, none of the existing works examine when, why and who is making permissions-related mistakes when developing apps.

In this paper, we provide a better understanding of how developers are creating and fixing permissions-related issues and the types of mistakes developers were making. To this aim, we analyzed the GitHub repositories of Android 574 apps. Using custom-built software along with the existing permission analysis tools M-Perm [16] and P-Lint [20], we identified a variety of PRIs ranging from not correctly adhering to permissions best practices to apps requesting too many permissions. This empirical information provides us with a history of the app's development life cycle including (i) When permissions and their related issues were introduced and fixed, (ii) who is making these decisions, (iii) file-change history that we could examine using permissions analysis tools, and (iv) all other commit information such as commit messages.

Our results reveal that (i) *PRIs are a frequent phenomenon* in Android apps (~50% of examined apps exhibit at least one PRI, with over-permissions being the most prevalent), (ii) the majority of issues are fixed in a timespan of a few days after their introduction, (iii) in many cases, permission-related issues can linger inside an app for an extended period of time, that can be as high as several years, before being fixed, and (iv) in total regular contributors introduce and fix a larger number of PRIs along the lifetime of Android apps, but this phenomenon is due to the fact that regular contributors commit more code changes.

To summarize, the main contributions of this study are:

2470-6892/19/$31.00 ©2019 IEEE
DOI 10.1109/SCAM.2019.00034

• a characterization of the *frequency* of PRIs and their *decay time* in the context of 574 open-source Android apps;
• an objective assessment of whether PRIs are introduced or fixed differently depending on the *status of the developer within the project*;
• the *replication package* of the study containing its results, raw data, and mining- and data analysis scripts [2].

The target audience of this paper includes both Android researchers and developers. Researchers are provided an *evidence-based* understanding of the phenomenon of permission-related issues in Android apps. Additionally, Android developers may use our findings to better plan their development activities (*e.g.,* planning refactoring sessions or assigning code reviews).

II. The Android Permission Model

Each Android app operates with a distinct Linux UID that is associated with a set of permissions. Services verify if the app's UID is permitted to access the requested functionality. An objective of this process is to ensure that the app adheres to the *principle of least privilege* – granting the least amount of privilege that the app needs to properly function [23]. For example, before an app may read SMS messages it must be granted the READ_SMS permission. To use the camera, the app would require the CAMERA permission. This is designed to limit the app from accessing unintended and non-user permitted functionality, and also to limit the effects that malware may have on a device [22]. Some permissions are considered to be less risky and are referred to as *normal* permissions. However, other permissions carry significantly more potentially hazardous risks and are known as *dangerous* permissions [3]. The *AndroidManifest.xml* file contains all permissions an app requests.

Deciding on the permissions an app should request is considered to be one of the most sensitive activities undertaken during development due to the potential security and privacy risks [23], and possible negative effects on the user's perception of the app [21], [48]. Studies have found that developers frequently misuse permissions by either not adding enough permissions to support requested functionality, or by adding unnecessary permissions that are not needed for any functionality in the app [23], [16]. Felt *et al.* [23] found that Android developers often mistakenly add unnecessary permissions in a counterproductive and futile attempt to make the app work correctly, or due to confusion over the permission name (*i.e.,* they add it incorrectly believing its functionality is necessary for their app). Developers should also ensure that they are using permissions correctly from various best practice perspectives [5]. Developers must also do their best to avoid permission smells [20] and user security fatigue [47]. Unfortunately, there is no permission enforcement mechanism that prevents developers from posting apps with improper permissions to Google Play or other app stores [12].

Example of permission-related issue. Listing 1 illustrates an example of PRI called *Missing Check* (MC) [20]. When the method showAppointments() is called by the app (line 3), the app is requesting permission to read the user's calendar (line 4). Beginning with Android 6.0, the call to requestPermissions() is necessary because users can revoke permissions at any time and developers cannot assume that the app currently has access to a specific permission, even if it previously had access to it [8]. However, each time requestPermissions() is called, a standard Android dialog is shown to the user for requesting the needed permission, even for permissions already granted [6]. Therefore, Android guidelines suggest that prior to running code that requires a specific permission, the method checkSelfPermission() should be called to determine if the user has already granted access to the needed permission [5]. Not adhering to this guideline can lead to a degradation of the user experience since the user is overwhelmed with messages requesting already granted permissions [6], [20].

```
1  // Method for listing appointments saved in the user's calendar
2  void showAppointments(){
3    ActivityCompat.requestPermissions(this, new String[ {Manifest.
         permission.READ_CALENDAR},
         PERMISSION_READ_CALENDAR);
4    // other tasks using information in the user's calendar
5  }
```

Listing 1: Apps should call CheckSelfPermission() to verify that it currently has access to the user's calendar.

III. Goal and Research Questions

The primary **goal of this study** is to provide a better understanding of permission-related issues introduced and fixed by developers in Android apps. To achieve this goal, we first collect 2,002 Android repositories from F-Droid [1] and then analyze these repositories using three existing open-source analysis tools: M-Perm [16], P-Lint [20], and oSARA [2]. Our research questions are as follows:

RQ1 – *What are the most common types of permission-related issues in Android apps?* By determining the most prevalent permission-related issues, Android developers can be made cognizant of these issues and devote appropriate efforts to avoid them in their apps. Answering RQ1 will also help researchers gain better insights into the prevalence of permission-related issues in Android apps. While previous work examines permissions-issues on the older install-time model [23], [11], [17], to our knowledge, this is the first study that examines permission-related issues on a large scale on the current Android run-time model.

RQ2 – *How long do permission-related issues tend to remain in Android apps across their lifetime?* Understanding how long permission-related issues typically exist in the code of Android apps can provide insight into how long introduced issues can be expected to impact the app. Indirectly, answering RQ2 provides an objective indication regarding the priority of developers to locate and address permission-related issues.

RQ3 – *How does developers' status within the project correlate with the introduction of permission-related issues?* By determining if a developer's status within a project significantly correlates with the introduction of PRIs provides

insight on who should be making permission-based decisions and modifications in Android apps. Answering RQ3 can also provide additional insight on whether regular contributors or project newcomers are introducing different amounts of permissions-related issues. This can create the foundation for improving the assignment of code reviews. For example, additional security-oriented reviews may be performed on code authored by developers whose status is more correlated with the introduction of PRIs.

RQ4 – *How does developers' status within the project correlate with fixes of permission-related issues?* Answering RQ4 provides insight on whether regular contributors or project newcomers fix different amounts PRIs. Here the underlying intuition is that developers with more experience in the project are more adept at fixing PRIs (see Section V-D).

Summarizing, identifying newcomer-specific effects in open-source projects is relatively new [14], [58], [38], [32] and it is specially important for better understanding the onboarding process in open-source Android projects [38] and for helping teams to deal with permission-related issues more efficiently, *e.g.,* via dedicated guidance in the decision making process, practices, and tools.

IV. Data Collection and Analysis

Our data collection and analysis process consists of 3 phases: *Repository Collection*, *Detection of PRIs*, and *Data Analysis*. We first mine the F-Droid catalog to obtain a list of open-source Android apps and perform a set of filtering steps on collected apps. In the second phase we execute the P-Lint and M-Perm tools for statically analyzing apps source code and project files. In the third phase, the results of the static analysis tools are statistically analyzed. Further details of this process can be found in Scoccia *et al.* [51].

A. Repository Collection

F-Droid is a catalog of FOSS apps for the Android platform. F-Droid contains links to Android app Github repositories. These projects range from small infrequently updated apps, to large popular apps. We chose F-Droid as our primary source for open-source Android projects due to the diversity of apps in its catalog and for its use in prior research works [39], [31], [10]. To retrieve the project repositories of the cataloged apps, we first cloned the F-Droid repository and then parsed the text files associated with each app to extract the apps' metadata. Extracted metadata includes name, description, category and repository URL of each app. We then clone the GitHub repository of all the apps. In order to avoid duplicates, we exclude apps from our dataset that were duplicated/forked by ensuring that all source URLs and commit log SHA's are unique. After cloning the repositories, we extract the following data from each of them:

- Commit Log Details. Using Git's *commit log*[1], we retrieve additional data associated to each commit, such as the author and committer of the commit, their respective timestamp, and the commit message.

[1]https://git-scm.com/docs/git-log

- Affected Files. For each commit of all apps, we examine the list of affected files and extract the revision of all the *.java and AndroidManifest.xml files.

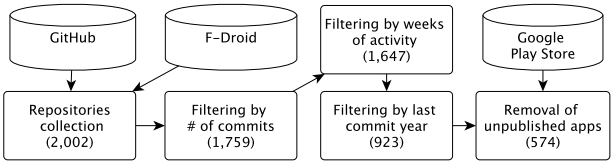

Fig. 1: Repositories collection and filtering process

As shown in Figure 1, we mined a total of 2,002 GitHub repositories. Since we used GitHub repositories, we ran the risk of including inactive or unmaintained repositories in our study [35]. To help mitigate this risk, we consider only repositories that (i) have a lifetime span[2] of at least 8 weeks, (ii) contain at least 10 commits, (iii) with at least one commit since January 2017 and (iv) also published on the Google Play store. The 10-commits threshold is derived from the fact that 90% of all considered repositories have more than 10 commits before this filtering step. The 8-weeks threshold is derived from the fact that 8 weeks is the average development time for an Android app [7] and has been used in a previous study on mining GitHub repositories of Android apps [42]. The January 2017 rule has been adopted to filter out unmaintained apps, without removing apps that are seldom updated. We excluded apps that were not published on the Google Play store to filter out unfinished or proof-of-concept apps. This filtering results in a final dataset of 574 active repositories, containing a total of 502,907 commits performed by 7,945 unique developers.

TABLE I: Demographics of apps included in the study (SD = standard deviation, IQR = inter-quartile range)

Metric	Min.	Max.	Median	Mean	SD	IQR
Rating	0	5	4.294	4.179	0.6681	0.4767
Installs[3]	1	100m	10k	926.1k	7,594k	99k
Commits	11	34,380	260	876.10	2246.97	707.5
Committers	1	486	7	16.67	32.71	13

Table I provides a summary of the demographics for apps included in the study. As demonstrated, apps in our dataset have a median rating on the Google Play store of 4.294 (out of a maximum of 5), while the median number of installs[3] is 10k. The median number of commits for apps in our study is 260, and the median number of committers per app is 7. Based on these numbers, we are reasonably confident that the apps considered in our study are of good quality and adequately representative of real-world projects.

[2]Lifetime span: the range between the first and last commits of a repository.

[3]Google Play does not provide the precise number of installs, but only a range (*i.e.,* 100-1000). We conservatively adopted the bottom of the range. Hence, all statistics on installs should be considered as a lower bound.

B. Detection of PRIs

We used the existing M-Perm [16] and P-Lint [20] tools to detect permission-based issues in Android apps. Although both tools have been used in foundational studies [16], [20], we decided to further evaluate them prior to including them in our own research. Other permission analysis tools, such as Stowaway [23] and PScout [11], have been used in existing literature to conduct permission analysis. However, a direct comparison with these tools was unfeasible, as both are several Android versions out of date and neither is compatible with the current run-time permission model.

Tool Evaluation. We evaluated M-Perm and P-Lint using several oracle Android apps. These include minimal calendar, camera, SMS messaging, contact storage and location recording apps. We then created multiple versions of these apps, and injected PRIs into them with the goal of covering numerous possible cases in which a PRI may occur. We then ran M-Perm and P-Lint on each of these app versions, identifying all TP, FP, FN and TN for PRIs. Both tools obtained a precision and recall value of 1.00. Although largely elementary, these results provided confidence in the ability of these tools in our study. The oracle apps are available on the project website [2]. Although both tools used in our study are able to decompile and analyze apk files [16], [20], decompilation was not performed as the source code for subject apps was readily available. In this analysis we created our own apps to provide a greater amount of confidence that we were aware of all PRIs in the this oracle, whereas manually identifying PRIs in existing apps would have been a time-consuming and largely imperfect task.

After the successful analysis of these tools, we used them to analyze all 502,907 commits belonging to the 574 apps in our dataset. These tools enabled us to identify a variety of permissions-based issues, ranging from not correctly adhering to the permission standards proposed by Google [5], to more severe issues such as over-permissions. Table II presents the PRIs considered in this study. **M-Perm** is able to detect occurrences of over and under-permission issues (*i.e., O* and *U* PRIs). An app is *over-privileged* if it requests too many permissions. Likewise, if it asks for too few permissions then it is *under-privileged* [23]. Apps that misuse permissions have an increased attack surface, making them more susceptible to a variety of security and privacy-related issues [23], [16]. M-Perm analyzes Android $\geq$ 23 apps and identifies instances of over and under-privileged permissions.

Similar to code smells, *permission smells* are symptoms of issues, but are not a definitive indication that a problem exists [20]. **P-Lint** analyzes Android $\geq$ 23 apps for proper permissions usage from a standards perspective. In this study we focused on the missing check (*i.e., MC*) and multiple requests in proximity (*i.e., MRP*) PRIs since (i) they were prevalent, occurring in a large number of apps and (ii) they were well-defined and had a clear negative impact. We focused on these four types of PRIs since they are (I) Impactful (II) Well-defined (III) Have been extensively analyzed in existing works (permission gap) [61], [23], [55]. Our study focuses on Android apps since we were able to easily collect and reverse engineer a large set of Android apps, something that would not be easily accomplished with iOS apps due to a lack of available tools and available apps.

TABLE II: Permission-related issues detected in this study

ID	Permission Issue	Quality	Security	Tool
O	*Over-permission:* too many permissions (violates the least privilege principle).		✓	M-Perm
U	*Under-permission:* not enough requested permissions.	✓		M-Perm
MC	*Missing Check:* checkSelfPermission() is not called when requesting a permission.	✓		P-Lint
MRP	*Multiple Requests in Proximity:* Multiple permission requested in close proximity, possibly overwhelming the user.	✓		P-Lint

After the detection of PRIs, we detect the commits that *introduced* and *fixed* each of them. This is a non-trivial task as identifying these issues involves much more analysis than merely examining each committed version with the static analysis tools. The following statuses define each PRI event:

- **New.** When a PRI is found, we check if it exists in the app at the time of the previous commit. If it does not, starting from the version containing the issue we examine each version of the app in a commit-by-commit fashion to determine the commit that introduced the PRI. Identifying this commit allows us to determine the committer responsible for introducing the PRI.
- **Exist.** If the detected PRI is also found to exist in the previous and subsequent versions of the app, then we record it as 'Exist' since the commit does not modify the state of the issue. These are expectedly observed quite frequently as developers often make a variety of changes to apps that are not permission-related.
- **Fix.** For every detected PRI, we check if the PRI exists in the subsequent committed version of the app. If it does not exist, we determine the commit that fixed the issue. This is accomplished by starting with the immediately subsequent commit after the version of the app exhibiting the detected PRI and examining its source code using the analysis tools. If the issue is not found, then we mark the current commit as the commit that fixed the issue. If the issue persists, we perform the same process on each subsequent commit until we find the commit that fixed the issue. This enables us to identify the committer responsible for fixing the permission issue. If we reach the last commit of the repository and no PRI fixing commit is found, then the PRI is marked as *unresolved*.

Demographics information about the detected PRIs and their related commits contextually to the discussion of the results of this study are provided in Section V-A.

oSARA Tool and Replication Package We leveraged the open Source Android Repository Analyzer (oSARA) tool [4] to perform the necessary data collection and analysis for our study. oSARA performs the following tasks: (I) Collects all relevant Android repository information from F-Droid; (II) Extracts all relevant permission information and versions from these repositories; (III) Analyzes each extracted version for PRIs using M-Perm and P-Lint; (IV) When PRIs are discovered, oSARA analyzes previous and subsequently committed files to determine the commit that either added or removed the PRI. Using this commit information, we are able to discern information about the developer performing the commit. Our project website [2] contains all code developed for the study, the raw dataset ($>$ 6 GB), the schema details of our collected data, and the oracle Apps used to verify P-Lint and M-perm.

C. Data Analysis

We will next describe the data analysis processes used to answer our research questions.

RQ1. We account for all occurrences of each type of PRI and provide an indication regarding their distributions by means of summary statistics. We employ the Fisher's exact test [9] to assess independence of observations among occurrences of the four PRIs types. We adopt the Fisher's test over alternatives (*e.g.,* χ^2-test [9]) due to its robustness when dealing with sparse, unbalanced data [44]. We employ the same test to perform post-hoc analysis, performing all tests for all pairs of populations and adjusting resulting p-values for inflation due to multiple comparisons via the Holm correction procedure [50]. The omnibus Friedman test [19] is then used to statistically determine if the four types of PRIs exhibit a significant difference. The Friedman test is a non-parametric test for one-way repeated measures analysis of variance by ranks. We use the Friedman test because (i) RQ1 is designed as a 1 factor – 4 treatments experiment, (ii) the collected data is not adhering to the assumptions of the ANOVA statistical test, and (iii) the Friedman test is a non-parametric alternative to ANOVA that does not assume independence of observations [19]. We apply the Conover's all-pairs comparison test as post-hoc analysis for performing pairwise comparisons among each pair of PRI types [18]. Since we are applying multiple statistical tests, we correct the obtained p-values via the Holm correction procedure [50]. We additionally compute the effect-size of the differences among PRIs distributions using the Cliff's delta (*d*) non-parametric effect size measure [30], which measures how often values in a distribution are larger than the values in a second distribution. Cliff's d ranges in the interval $[-1, 1]$ and is considered negligible for $d < 0.147$, small for $0.148 \leq d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$.

RQ2. In this phase of the study, we collect the *decay time* of each occurrence of PRI. The decay time of a PRI represents the number of days in which a PRI is present in the source code of an app. We compute the decay time of a PRI as the difference (in days) between the timestamp of the commit in which the PRI has been fixed and the timestamp of the commit in which it has been introduced in the GitHub repository of the app. In this phase of the study we exclusively consider the PRIs which have been fixed along the lifetime of the app, so that their decay time is meaningful (*i.e.,* the last commit of a PRI includes the actual fix of the PRI and it does not correspond to the last commit within the whole repository).

Summary statistics are used for providing an indication about how decay times vary across the four types of PRIs. The same statistical tests as in RQ1 (*i.e.,* Friedman, Conover, Holm correction and Cliff's delta) are used for statistically assessing the differences of decay times across PRIs.

RQ3 – RQ4. Both research questions RQ3 and RQ4 are based on the concept of a developer's status. In existing literature, several repository-based metrics for proxying developer's status (or experience) have been proposed, such as (i) *Developer's Commit Ratio* (DCR), defined as the number of contributions made by a given developer for a repository divided by the number of all commits done by all repository's contributors [40], (ii) *maintainers* and *contributors* defined as those contributors with more than 30% and less than 10% of all repository's commits, respectively [57], and (iii) project *newcomers* defined as those contributors with less than 3 commits in a repository [38]. In this study we use the latter metric, as it has been defined in the literature [14], [58], [38], [32]. Specifically, the status of a developer d at a given commit c in a repository r as:

$$status(d, c, r) = \begin{cases} \text{Newcomer}, & \text{if } nCommits(d, c, r) \leq 3 \\ \text{Regular}, & \text{otherwise} \end{cases}$$

where c is the specific commit in r for which we want to calculate d's status and $nCommits(d, c, r)$ is the number of commits authored by developer d in repository r at the time in which commit c is performed. Intuitively, at a given time, a developer is a $\mathrm{newcomer}$ in the context of a given project if she performed no more than 3 commits in the repository, otherwise the developer is identified as a $\mathrm{regular}$ contributor. We opted for the $status(d, c, r)$ metric since it is (i) computationally lightweight, (ii) used in the literature, and (iii) independent from the size of the repository r.

To avoid the well-known *aliasing problem*, *i.e.,* the same developer having multiple identities in GitHub repositories [28], we apply the heuristic proposed by Kouters *et al.* for resolving developers using multiple identities when committing on the same repository [37]. This heuristic merges committers with the same email prefix, *i.e.,* the part before the @ symbol. We chose the heuristic proposed by Kouters *et al.* because, despite its apparent simplicity, there is empirical evidence that it provides a good enough trade-off between performance and simplicity of implementation w.r.t. other heuristics when considering long time frames as in our study. We refer the reader to [60] for a detailed evaluation of various heuristics for solving the aliasing problem.

To account for project contributors authoring more commits in potentially introducing and/or fix more PRIs, we com-

pute two additional metrics: $issuesPerCommit_i(d,t,r)$ and $issuesPerCommit_f(d,t,r)$ [58]. Both metrics are defined for each type of issue t, developer d, and repository r. The first metric is defined as $\frac{pris_i(d,r)}{nCommits(d,lastCommit(d,r),r)}$, where $pris_i(d,r)$ is the set of PRIs introduced by developer d in all commits they authored in r, and $lastCommit(d,r)$ is the last commit authored by d in r. Intuitively, $issuesPerCommit_i(d,t,r)$ represents the ratio between the total number of PRIs introduced by a developer in a repository and the total number of commits they authored in that repository. The $issuesPerCommit_f(d,t,r)$ metric is similar to $issuesPerCommit_i(d,t,r)$, but it focuses on the number of *fixed* PRIs.

For answering RQ3 and RQ4, we report and analyze the frequency of PRIs introduced (RQ3) and fixed (RQ4) in commits performed by project newcomers and regular contributors. Next, we build contingency tables with rows representing the types of PRIs and columns representing the developers' status; then, we compute the Cramer's V coefficient of each contingency table [50]. The Cramer's V coefficient is a well-known measure of association applicable to contingency tables involving two categorical variables and it is defined within the $[0,1]$ range, where 0 indicates no correlation and a value of 1 indicates perfect correlation.

For RQ3 and RQ4 we provide descriptive statistics for both the $issuePerCommit_i(d,t,r)$ and $issuePerCommit_f(d,t,r)$ metrics. For each metric, we apply the Mann-Whitney U test for statistically testing the following two-tailed null hypothesis: the distributions of the number of PRIs introduced (fixed) per commit are the same for both newcomers and regular contributors [62]. The effect-size of the differences among PRIs distributions is quantified by using the Cliff's delta (*d*) measure.

V. Results

A. RQ1 – What are the most common types of permission-related issues in Android apps?

Table III provides descriptive statistics for occurrences of PRIs, as well as counts of unique issues and affected apps. A total of 3,900 unique permission issues were identified. They are distributed across 402 distinct apps, with a median of 1 PRI per app. For all types of PRI, we can observe that the mean amount of occurrences is higher than the median, meaning that the average is influenced by apps in the upper part of the data that exhibit an especially high amount of PRIs. Furthermore, we can observe that over and under-permissions are the two most common issues, with 2,635 and 939 occurrences. The apps in our dataset have on average more than four over-permission issues and more than one under-permission issue. Diffusion of issue types *MC* and *MRP* appears to be on a lower scale, with 205 and 91 instances affecting 60 and 9 apps.

As a preliminary step prior to further analysis, we test for independence of observations among the four PRIs types. We statistically test this hypothesis by applying the Fisher's exact test. The results of the test ($p-value < 0.01$) allow us to reject the null hypothesis of independence among occurrences of PRI types. Likewise, the null hypothesis is always rejected ($p-value < 0.01$) for all post-hoc pairwise comparisons.

TABLE III: Descriptive statistics for occurrences of PRIs

PRI	#	Affected apps #	Min.	Max.	Median	Mean	SD	IQR
O	2,635	387	0	269	1	4.59	16.98	3
U	969	82	0	251	0	1.69	12.31	0
MC	205	60	0	32	0	0.36	1.76	0
MRP	91	9	0	67	0	0.16	2.84	0
Aggr.	3,900	402	0	377	1	6.79	26.28	4

Differences in means and standard deviation across the four types of PRIs suggest that the distribution of occurrences differs according to PRI type. We statistically test this hypothesis by applying the Friedman omnibus test. These results ($p-value < 0.01$) allow us to reject the null hypothesis that distributions of occurrences of PRI types are not statistically significantly different. Results of pairwise comparisons, using the Conover's test, reveal that the distribution of occurrences of each PRI type is statistically different from the others. Estimations of magnitude of differences, via pairwise applications of Cliff's *d*, reveal a *large* effect size for all pairs involving PRIs of type *O*, while it is *negligible* for all other pairs.

B. RQ2 – How long do permission-related issues tend to remain in Android apps across their lifetime?

Descriptive statistics of decay time for each type of PRI is summarized in Table IV. We can observe that for all PRI types the minimum decay time is equal to 1 day, while the maximum is close to 7, 3, 2 and 1.5 years for issues of type *O*, *U*, *MC*, and *MRP*, respectively. Median decay is quite similar for *O* and *U* issues, with a value of approximately one week, but significantly differs for issues *MC* and *MRP*, with a value of about 12 weeks for the former and 1 day for the latter. As expected, results of the application of the Friedman omnibus test ($p-value < 0.01$) allows us to reject the *null* hypothesis that distributions of decay times across the four types of PRIs does not significantly differ. Post-hoc analysis, performed via the Conover's test, reveals that the distribution of decay times for each PRI type is significantly different from the others, with the sole exception of the *U-MC* pair, for which we cannot reject the null hypothesis.

TABLE IV: Descriptive statistics for decay time of PRIs

PRI	Min.	Max.	Median	Mean	SD	IQR
O	1	2,784	6	187.6	419.11	105
U	1	1,066	5	45.25	102.35	28
MC	1	760	82.5	166.8	200.96	303.75
MRP	1	544	1	43.82	124.29	1.5
Aggr.	1	2,784	6	150.3	360.51	84

The mean of decay time is much higher than the median for all PRI types. This suggests that the average is greatly influenced by a subset of the data on the higher part of the scale. This observation is even more notable for *O* and *MC*

PRIs, that exhibit a comparatively significantly higher mean (187.6 and 166.8 days, against 45.25 and 43.82 days for issues *U* and *MRP*) and standard deviation (419.11 and 200.96 days, opposed to 102.35 and 124.29 days). We also observe that *MC* issues exhibit a relatively higher inter-quartile range (303.75 days), implying that decay time for *MC* issues is much more dispersed than for other PRIs. Results of applications of Cliff's *d* for effect size estimations reveal a *small* effect between *O* and *MC* and *negligible* for all other pairs.

C. RQ3 – How does developers' status within the project correlate with the introduction of permission-related issues?

Figure 2 shows the frequency of developers' status when introducing each type of PRI. We observe that PRIs are mostly introduced by regular contributors. Over-permissioning is the type of PRI which is introduced more frequently by project newcomers, however its frequency is still far below the one of regular contributors.

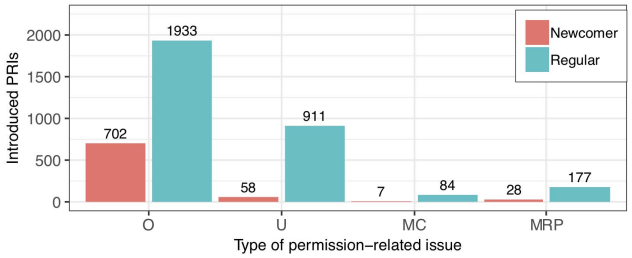

Fig. 2: Developers' status when introducing PRIs

In order to quantitatively assess if whether the introduction of PRIs among project newcomers and regular contributors depends on the type of PRI, we compute the Cramer's V coefficient, which measures the strength of association - varying between 0 to 1 - between two nominal variables. In this case, the computed Cramer's V coefficient value is 0.227 (which is *low*) meaning that there is a low association between developers' status and the types of PRIs being introduced.

TABLE V: Issues per commit over developers' status when introducing PRIs

PRI	Min.	Max.	Median	Mean	SD	IQR
Newcomer						
O	0	5.0	0	0.022	0.177	0
U	0	3.0	0	0.004	0.064	0
MC	0	1.667	0	0.0004	0.025	0
MRP	0	3.0	0	0.002	0.051	0
Aggr.	0	5.0	0	0.007	0.099	0
Regular						
O	0	1.714	0	0.005	0.051	0
U	0	0.75	0	0.001	0.051	0
MC	0	0.585	0	0.0002	0.021	0
MRP	0	0.667	0	0.001	0.008	0
Aggr.	0	1.714	0	0.002	0.029	0
Aggr.	0	5.0	0	0.004	0.073	0

As previously mentioned in Section IV, the results discussed above may depend on the total number of commits that each developer performs in a repository. Table V shows the number of PRIs introduced by each developer per repository, normalized by the total number of authored commits. Here we can observe that the number of PRIs per commit are generally very low both for project newcomers and regular contributors, with all medians equal to 0, very low averages, and extremely compact distributions (all standard deviations < 0.177).

After the application of the Mann-Whitney U test, we do not obtain a statistically significant measure of correlation between these two categorizations ($p\text{-}value = 0.29$)); this does not allow us to reject the null hypothesis that the distributions of the number of PRIs introduced per commit are the same for newcomers and regular contributors.

D. RQ4 – How does developers' status within the project correlate with fixes of permission-related issues?

We answer this research question by following the same procedure of RQ3; the only differences are that (i) now we are focusing on the commits in which PRIs have been fixed (as opposed to when they are firstly introduced) and (ii) we are considering exclusively the PRIs which have been fixed along the lifetime of the app, so that their PRI fixings commit is meaningful. Figure 3 shows the frequency of developers' statusses when fixing each type of PRI.

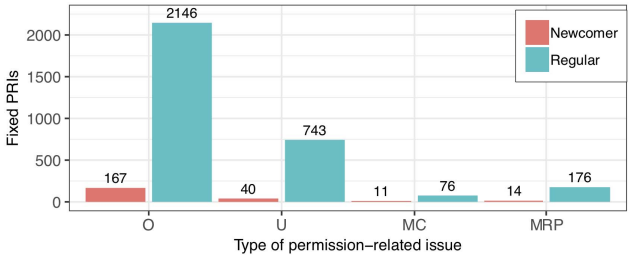

Fig. 3: Developers' status when fixing PRIs

The data demonstrates that in total, regular contributors fix more PRIs than project newcomers, specially when dealing with over-permissioning PRIs. Similarly to what happened also for RQ3, we also have a very *low* Cramer's V coefficient, *i.e.,* 0.051. Again, this confirms that there is a low association between developers' status and the types of PRIs being fixed.

Table VI presents descriptive statistics for the number of PRIs *fixed* by each developer in each repository, normalized by the total number of commits authored by each developer authored in each repository. Also in this case, the number of PRIs per commit is very low across all PRI types and developers' status, with an overall meanof 0.003 and all medians equal to zero. The application of the Mann-Whitney U test yields a statistically significant result with $p\text{-}value < 0.01$), allowing us to reject the null hypothesis that the distributions of the number of PRIs fixed per commit are the same for newcomers and regular contributors [62]. However, the effect size is *negligible* (Cliff's $d = -0.023$).

TABLE VI: Issues per commit over developers' status when introducing PRIs

PRI	Min.	Max.	Median	Mean	SD	IQR
Newcomer						
O	0	4.0	0	0.013	0.14	0
U	0	2.8	0	0.003	0.065	0
MC	0	1.0	0	0.001	0.031	0
MRP	0	1.0	0	0.001	0.035	0
Aggr.	0	4.0	0	0.005	0.081	0
Regular						
O	0	1.643	0	0.005	0.042	0
U	0	2.8	0	0.002	0.044	0
MC	0	0.333	0	0.0001	0.005	0
MRP	0	0.667	0	0.0009	0.017	0
Aggr.	0	2.8	0	0.002	0.032	0
Aggr.	0	4.0	0	0.003	0.061	0

VI. Discussion

RQ1. Permission-related issues are a frequent phenomenon in Android apps. The vast majority of the analyzed apps suffer from the presence of at least one PRI. Over and under-permissions are more prevalent than *MC* and *MRP* PRIs. Occurrences of PRIs appear to be dependent among PRI types. The distribution of occurrences significantly differs for each PRI type. By examining the number of issues identified for each PRI type, we can easily observe that the majority of issues is of types *O* and *U*. The mean amount of occurrences per app differs among the two, with a value $\mu = 4.59$ for the former and $\mu = 1.69$ for the latter. These results provide an initial notion of the prevalence of over- and under-permission phenomena in Android apps, as partially also confirmed by Felt *et al.* [23]. Moreover, by examining the counts of identified issues for all PRI types, we notice that issues of types *MC* and *MRP* amount to a comparatively small minority of the total. Although further research is required to fully determine the reason behind this imbalance, we believe that a primary factor is that *MC* and *MRP* issues are harder to introduce. In fact, in order to introduce *MC* or *MRP*, specific conditions must be met in the application code. However, types *O* or *U* may only require a mistake in the Android Manifest file. The dependence among occurrences of PRI types hints that whenever one type of PRI is found in the development history of an app, then also other kinds of PRIs are likely to be present. This is not overly surprising as developers who are not knowledgeable, or attentive about permissions are more likely to introduce multiple types of PRIs in their apps.

The most common types of PRIs occurring in Android apps are of types *O* and *U*. This indicates that even if issues and their consequences are well-known and have been studied in-depth by the academic community [23], [17], [11], that they are still a common occurrence, even in apps developed for the newer versions of Android. As a consequence, advise developers to pay more attention to these PRI types and also advocate for the adoption of permission analysis tools (such as M-Perm [16] and P-Lint [20]) during app development.

RQ2. Results indicate that *the majority of PRIs are fixed in a timespan of a few days after their introduction*. Nonetheless, in many cases *PRIs can linger inside an app for an extended period of time*, that can be as high as several years, before being fixed. The PRIs considered in this study can impact the end-users opinion of the app [48], [21], and can result in security problems [23]. Therefore, understanding characteristics and reasons for the persistence of these longer-living PRIs represents a relevant research question that demands further investigation. Of particular interest are the higher median values of issues *MC*. As previously mentioned in RQ1, specific conditions must be met inside an app's source code to introduce one of these issues, meaning that the issue cannot solely exist in the AndroidManifest file. We speculate that this greater specificity of necessary conditions is also the reason behind the greater median decay time, *i.e.,* once introduced, more non-trivial changes in the source code must be carried out to fix such issues. In other words, *MC issues are harder to introduce but also harder to fix once introduced*.

Given the fact that PRIs of all kinds can linger inside an app for an extended period of time we encourage developers and organizations to pay increased attention to code that has been written during early project life, during quality assurance activities (*i.e.,* code review sessions). Moreover, since *MC* issues tend to persist a long time once introduced, extra attention should be paid by developers and organizations to both ensure that they are not introduced, but to also regularly check their apps for these types of issues. Further work is needed to understand precisely why *MC*s tend to last longer compared with other PRIs.

RQ3. Overall, regular contributors introduce more PRIs in Android apps w.r.t. project newcomers across all types of PRIs. This result is (i) quite expected since the number of commits authored by regular contributors is much higher than the number of commits authored by project newcomers (216,069 vs 10,383 commits in total) and (ii) confirmed by Tufano and colleagues [58], another study involving code smells introduced by newcomers or regular contributors in Java-based open-source projects. This observation is further confirmed when analyzing the number of introduced issues per commits; indeed, even though in average project newcomers tend to introduce more issues across all types of PRIs (mean number of introduced PRIs = 0.005 for newcomers vs 0.002 for regular contributors), such a difference is not statistically significant. This finding may be an indication that *both project newcomers and regular contributors actually risk to introduce PRIs when working on their Android apps*. We suggest organizations and project maintainers to take special care of PRIs (*e.g.,* by planning dedicated code review sessions), independently of the experience of the developer performing the commit.

The results discussed above demonstrate that even regular contributors need to be cognizant of PRIs. This strengthens the case for adopting permission analysis tools during app development, as discussed in Section V-A. In addition, we suggest organizations and project maintainers to be cognizant

of over- and under-permission issues during activities that might require changes to app permissions, even when regular contributors are involved.

RQ4. The frequencies of PRI fixes across project newcomers and regular contributors tend to follow the same trends as the ones related to the introduction of PRIs (see Section V-C), but with one main difference: *project newcomers fixed fewer PRIs in total, specially for over-permissioning issues*. We may expect this observation since we can speculate that PRIs are non-trivial issues and are managed (and fixed) by developers who are more familiar with the internals of the app being developed. The obtained results confirm the intuition that since PRIs are non-trivial issues in an Android they tend to be fixed by developers who are not newcomers in the project.

Newcomers and regular contributors are exhibiting a statistically-significant difference of the number of issues fixed per commit, but with a negligible effect size. This means that, despite the fact that regular contributors tend to fix more PRIs per commit, such a difference is extremely small. It is interesting to mention that a recent study on code smells in Android apps [32] further confirmed that *developers with few contributions, like newcomers, do not forcibly introduce more or remove less code smells*. Overall, those observations lead us to conjecture that PRIs (and generic code smells as emerged in the work by Habchi *et al.* [32]) in open-source Android projects are managed by contributors belonging to different groups and that developers' experience does not seem to be a good predictor of the introduction or fixing of PRIs/code smells in the source code of the app. As future work, we will perform a more in-depth analysis in order to better characterize this phenomenon, *e.g.,* by investigating on the specific activities performed by developers when introducing/fixing PRIs, interviewing developers to better understand the context in which PRIs are introduced and fixed, and to assess if integrating the automated detection of PRIs in the development workflow (*e.g.,* in a continuous-integration pipeline) may help in having less PRIs in today's Android apps.

VII. Threats To Validity

Although our research led to several interesting results, there are several threats to validity.

Internal Validity In this study we rely upon the M-Perm and P-Lint tools. While these tools have been published in peer-reviewed venues, they are both still reasonably new. Like with all static analysis tools, they are not perfect, and tool imperfections have the capability to skew research results. In particular, obfuscated code is known to be particularly challenging for tools of this kind [45]. As described in Section IV-B, we validated both tools on a set of benchmark applications, thus making us reasonably confident about their accuracy. In order to foster independent checks and verification, all evaluation data is available in the replication package [2] of this study.

We examined 'commit ownership' and not 'code ownership' in our study. While 'code ownership' is a general term used to describe whether one person is primarily responsible for a software component [13], commit ownership is merely the author who made the commit to the repository. Due to our empirical examination of existing repositories, it would have been impossible to examine code ownership in our study. Since we only knew the committing author, we were unable to account for other developers who may have contributed to the commit, for example in the case of pair programming. We, therefore, considered 'code ownership' out of scope for this study and focused on 'commit ownership'. However, future work could also include code ownership to provide a possible alternative view on the results.

We relied on the $status(d, c, r)$ metric to proxy developer's experience in a project. Although reasons for this choice were described in Section IV-C, it is important to notice that the metric we adopted does not consider factors such as commit scale, quality of the work done or frequency among developer commits. Therefore in some cases, the metric might not properly represent a developer experience.

We utilized Git user names to identify developers. An inherent limitation of using this process is that developers could use different user names throughout the project, and the researcher would only be able to assume that these are two different developers. An additional limitation of many empirical studies is if developers are following a pair programming process, then the committer of the code will be assumed to be the sole developer. The study would not be able to account for the efforts of the non-committing developer.

In some cases, due to licensing reasons, open-source app repositories might not contain parts of the app code that is added at a later stage, before publication in app stores (*e.g.,* ad libraries). In these cases, the app manifest file might include some permissions currently not used in its code but added in anticipation of additions. Our analysis of over-permissions might have been influenced by these instances.

External Validity For our study, we empirically analyzed the version control repositories of open-source apps. While we analyzed a large number of open-source Android applications, we only examined a small subset of the millions of available Android apps, and hence our results might not generalize.

Other permission analysis tools such as PScout [11] could also have been included to examine apps that rely on the install-time permission system, in use until Android API versions 5.1. In our study, we did not include other tools as we focused on the current Android permission model and for consistency reasons. M-Perm uses a call graph to determine the reachability of the app's source code. However, during our analysis, we did not evaluate M-Perm's ability to reach dynamically loaded code. We may, therefore, consider this a potential limitation to our study.

Our work is empirical in nature, enabling us to analyze a large number of apps. Future work could conduct a laboratory study and include developer interviews to further understand developer permissions-decisions and mistakes.

VIII. Related Work

Previous works have analyzed Android permissions from a variety of perspectives. Stowaway [23] combines callback

directed API with the app behavior to identify the necessary permissions for the app's runtime. PScout [11] parses the examined code to build its syntax tree and then used it to link between active API calls and invoked permissions. Krutz *et al.* [40] did not target permissions-misuse in their study, but did find that developers who revert permission-related decisions typically had a higher level of code ownership than the developer who added the permission. Calciati *et al.* [15] conducted a preliminary study to understand how permission requests apps evolve over several releases. They found that apps typically request more permissions over time and that the removal of permissions does not typically imply the loss of functionality. This work differs from ours in that we primarily focused on developer tendencies and who was actually making the permissions-based decisions, and mistakes, in the app development and maintenance process.

Researchers have studied the prevalence and effects of permission-misuse in Android apps. Tang [55] examined 10,710 apps and found that 76% of the apps contain at least one over-privilege. This work found a much higher occurrence of over-privileges as opposed to prior studies using Stowaway (36%) [17] and PScout (53%) [11]. This work differs in that we examined permissions and permission-based issues during the development process (not merely from a topical perspective), analyzed only Android 6.0+ for permissions issues, and we utilized more than merely an NLP-based technique to discover permission-based mistakes. Jha *et al.* [34] report on the different types of mistakes committed by developers in writing Android manifest files. Their results highlight that developers often commit mistakes while performing this activity, which can translate into security, reliability, and availability issues. By analyzing file-change history over the app's development life cycle, we were able to track a different set of PRIs and investigate developers' roles in introducing and fixing these issues. Watanabe *et al.* [59] focused on text description that accompanies an app on app stores, comparing resource mentioned in descriptions to those used in the app source code. From their analysis, they identify some common reasons that lead to overpermissioning issues in Android apps, including usages of third party libraries and frameworks that require unnecessary permissions. In our work, instead we focused more on ocurrences of PRIs in an app development history, cross referencing information extracted from apps source code and its commit histories. Mujahid *et al.* [46] investigated the occurrences of specific PRIs that can affect Android wearable apps. Their findings highlight that a considerable amount of apps are affected by these issues and that overpermission issues are quite common, to comply with requirements specific for Android wearable apps. They hint that developers lack of knowledge and lack of support from existing tools are the main reasons for the proliferation of such issues.

Previous works have analyzed effects of permissions on the user's perception of the app [48], [21]. Lin *et al.* [41] examined user comfort levels when using permissions they did not fully understand, or when they did not comprehend why the app needed the permission. They found that users generally felt uncomfortable and may even delete applications when they did not understand why it requested a permission they deemed unnecessary. Scoccia *et al.* [52] found that users tend not to understand why they are being asked for certain permissions, and frequently complain about this in their reviews.

Taylor *et al.* [56] examined the evolution of app permission usage with each app release. This was accomplished by taking snapshots of requested app permissions in the Google Play store. They found apps were requesting more permissions over time. This work differs from ours in that they examined permission requests at a higher app level, while we examined who was making permission requests cause permission issues.

Numerous other works have explored the impact of code smells on a variety of factors including code quality and software security. Rahman *et al.* [49] conducted a qualitative analysis of 1,726 infrastructure as code (IaC) scripts to identify seven common security smells. This work also created a static analysis tool that located security smells and identified software vulnerabilities. Other works have even explored code and security smells in Android [27], [43], [33], [36]. While we also examine smells in Android, we focus specifically on PRIs, and permission smells. Code smells have even been used to predict software issues. Soltanifar *et al.* [53] utilized code smells to create a defect prediction model. This work found that code smells are a strong indicator of possible defects in a software product. While these previous works have examined code smells, their causes, and impacts from a variety of perspectives, our work is the first to examine when *permission* smells in mobile apps appear, and who creates them.

IX. Conclusions and Future Work

The results of the study provide evidence-based insights for better understanding and managing permission-related issues in Android apps. Specifically: (i) permission-related issues is a frequent phenomenon in Android apps, with a strong prevalence of over-and under- permissions; (ii) the majority of permission-related issues are fixed in a span of a few days, even though in many cases some issues can plague the app for an extended period of time (*i.e.,* years) before being fixed; (iii) regular contributors introduce and fix a considerably larger number of PRIs along the lifetime of Android apps, but this phenomenon is related to the fact that regular contributors commit more code changes.

Future work will investigate if PRIs accumulate-diminish over the lifetime of an app, potentially revealing interesting patterns about their evolution. We will also perform a more in-depth study to understand what developers do when they introduce or fix PRIs; this study will involve a qualitative analysis of (i) the changes performed in the PRI introducing/fixing commits, (ii) their corresponding commit messages, and (iii) the discussions around their related pull requests.

Our work benefits both developers and researchers to better understand permission-related issues. For researchers, this paper create the foundation for future work in the area of permissions-related issues. For developers, this work provides insight on how teams can better plan development activities.

REFERENCES

[1] Fdroid repository. https://f-droid.org/.

[2] Mobile permissions. https://mobileevolution.github.io.

[3] Normal and dangerous permissions. https://developer.android.com/guide/topics/permissions/requesting.html#normal-dangerous.

[4] Osara: Open source android repository analyzer. https://github.com/MobileEvolution/oSARA/.

[5] Permissions best practices. https://developer.android.com/training/permissions/best-practices.html.

[6] Requesting permissions at run time. https://developer.android.com/training/permissions/requesting.html.

[7] How long does it take to build a mobile app? http://www.kinvey.com/how-long-to-build-an-app-infographic, 2017.

[8] Permissions Overview – Android developer guidelines. https://developer.android.com/guide/topics/permissions/, 2018. [Online; accessed 16-May-2019].

[9] A. Agresti and M. Kateri. *Categorical data analysis*. Springer, 2011.

[10] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 468–471. IEEE, 2016.

[11] 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*, CCS '12, pages 217–228, New York, NY, USA, 2012.

[12] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot. Understanding and improving app installation security mechanisms through empirical analysis of android. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 81–92, New York, NY, USA, 2012. ACM.

[13] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14, New York, NY, USA, 2011. ACM.

[14] M. Boussaa, W. Kessentini, M. Kessentini, S. Bechikh, and S. B. Chikha. Competitive coevolutionary code-smells detection. In *International Symposium on Search Based Software Engineering*, pages 50–65. Springer, 2013.

[15] P. Calciati and A. Gorla. How do apps evolve in their permission requests?: A preliminary study. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 37–41. IEEE Press, 2017.

[16] P. Chester, C. Jones, M. W. Mkaouer, and D. E. Krutz. M-perm: A lightweight detector for android permission gaps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 217–218, 2017.

[17] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[18] W. J. Conover and W. J. Conover. Practical nonparametric statistics. 1980.

[19] W. W. Daniel et al. *Applied nonparametric statistics*. Houghton Mifflin, 1978.

[20] C. Dennis, D. E. Krutz, and M. W. Mkaouer. P-lint: A permission smell detector for android applications. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 219–220, May 2017.

[21] S. Egelman, A. P. Felt, and D. Wagner. Choice architecture and smartphone privacy: There's a price for that. In *In Workshop on the Economics of Information Security (WEIS)*, 2012.

[22] 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*, 17(2):998–1022, 2015.

[23] 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.

[24] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*, WebApps'11, pages 7–7, 2011.

[25] 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*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[26] X. Gao, D. Liu, H. Wang, and K. Sun. Pmdroid: Permission supervision for android advertising. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 120–129, Sept 2015.

[27] M. Ghafari, P. Gadient, and O. Nierstrasz. Security smells in android. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 121–130, Sep. 2017.

[28] M. Goeminne and T. Mens. A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971–986, 2013.

[29] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 101–112, New York, NY, USA, 2012. ACM.

[30] R. J. Grissom and J. J. Kim. Effect sizes for research. *A broad practical approach. Mah*, 2005.

[31] S. Habchi, G. Hecht, R. Rouvoy, and N. Moha. Code smells in ios apps: How do they compare to android? In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 110–121, 2017.

[32] S. Habchi, N. Moha, and R. Rouvoy. The rise of android code smells: Who is to blame? In *MSR 2019-Proceedings of the 16th International Conference on Mining Software Repositories*, 2019.

[33] G. Hecht, N. Moha, and R. Rouvoy. An empirical study of the performance impacts of android code smells. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 59–69, May 2016.

[34] A. K. Jha, S. Lee, and W. J. Lee. Developer mistakes in writing android manifests: An empirical study of configuration errors. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, pages 25–36. IEEE, 2017.

[35] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. An in-depth study of the promises and perils of mining github. *Empirical Software Engineering*, 21(5):2035–2071, 2016.

[36] M. Kessentini and A. Ouni. Detecting android smells using multi-objective genetic programming. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 122–132, May 2017.

[37] E. Kouters, B. Vasilescu, A. Serebrenik, and M. G. van den Brand. Who's who in gnome: Using lsa to merge software repository identities. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 592–595.

[38] V. Kovalenko and A. Bacchelli. Code review for newcomers: is it different? In *2018 IEEE/ACM 11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 29–32. IEEE, 2018.

[39] D. E. Krutz, M. Mirakhorli, S. A. Malachowsky, A. Ruiz, J. Peterson, A. Filipski, and J. Smith. A dataset of open-source android applications. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 522–525, 2015.

[40] 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 *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '17, pages 165–169, Piscataway, NJ, USA, 2017. IEEE Press.

[41] 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*, UbiComp '12, pages 501–510, New York, NY, USA, 2012. ACM.

[42] I. Malavolta, R. Verdecchia, B. Filipovic, M. Bruntink, and P. Lago. How maintainability issues of android apps evolve. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, Sep. 2018.

[43] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen. Understanding code smells in android applications. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 225–236, May 2016.

[44] C. R. Mehta, N. R. Patel, and A. A. Tsiatis. Exact significance testing to establish treatment equivalence with ordered categorical data. *Biometrics*, pages 819–825, 1984.

[45] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, pages 421–430. IEEE, 2007.

[46] S. Mujahid, R. Abdalkareem, and E. Shihab. Studying permission related issues in android wearable apps. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 345–356. IEEE, 2018.

[47] S. Parkin, K. Krol, I. Becker, and M. A. Sasse. Applying cognitive control modes to identify security fatigue hotspots. In *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. USENIX Association, 2016.

[48] A. Peruma, J. Palmerino, and D. E. Krutz. Investigating user perception and comprehension of android permission models. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft '18, pages 56–66, New York, NY, USA, 2018. ACM.

[49] A. Rahman, C. Parnin, and L. Williams. The seven sins: Security smells in infrastructure as code scripts. In *Proceedings of the 41st International Conference on Software Engineering*, 2019.

[50] J. Rosenberg. Statistical methods and measurement. In *Guide to Advanced Empirical Software Engineering*, pages 155–184. Springer, 2008.

[51] G. L. Scoccia, A. Peruma, V. Pujols, B. Christians, and D. E. Krutz. An empirical history of permission requests and mistakes in open source android apps. In *Proceedings of the 16th International Conference on Mining Software Repositories*, MSR '19, pages 597–601, Piscataway, NJ, USA, 2019. IEEE Press.

[52] G. L. Scoccia, S. Ruberto, I. Malavolta, M. Autili, and P. Inverardi. An investigation into android run-time permissions from the end users' perspective. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft@ICSE 2018, Gothenburg, Sweden, May 27 - 28, 2018*, pages 45–55.

[53] B. Soltanifar, S. Akbarinasaji, B. Caglayan, A. B. Bener, A. Filiz, and B. M. Kramer. Software analytics in practice: A defect prediction model using code smells. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, IDEAS '16, pages 148–155, New York, NY, USA, 2016. ACM.

[54] R. Stevens, J. Ganz, V. Filkov, P. Devanbu, and H. Chen. Asking for (and about) permissions used by android apps. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 31–40, 2013.

[55] J. Tang, R. Li, H. Han, H. Zhang, and X. Gu. Detecting permission overclaim of android applications with static and semantic analysis approach. In *2017 IEEE Trustcom/BigDataSE/ICESS*, pages 706–713, Aug 2017.

[56] V. F. Taylor and I. Martinovic. To update or not to update: Insights from a two-year study of android app evolution. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* ASIA CCS '17, pages 45–57, New York, NY, USA, 2017. ACM.

[57] A. Trockman. Adding sparkle to social coding: An empirical study of repository badges in the npm ecosystem. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, pages 524–526, New York, NY, USA, 2018. ACM.

[58] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, 2017.

[59] T. Watanabe, M. Akiyama, T. Sakai, and T. Mori. Understanding the inconsistencies between text descriptions and the use of privacy-sensitive resources of mobile apps. In *Eleventh Symposium On Usable Privacy and Security ({SOUPS} 2015)*, pages 241–255, 2015.

[60] I. S. Wiese, J. T. da Silva, I. Steinmacher, C. Treude, and M. A. Gerosa. Who is who in the mailing list? comparing six disambiguation heuristics to identify multiple addresses of a participant. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 345–355. IEEE, 2016.

[61] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. Wagner, and K. Beznosov. Android permissions remystified: A field study on contextual integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 499–514, 2015.

[62] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Computer Science. Springer, 2012.